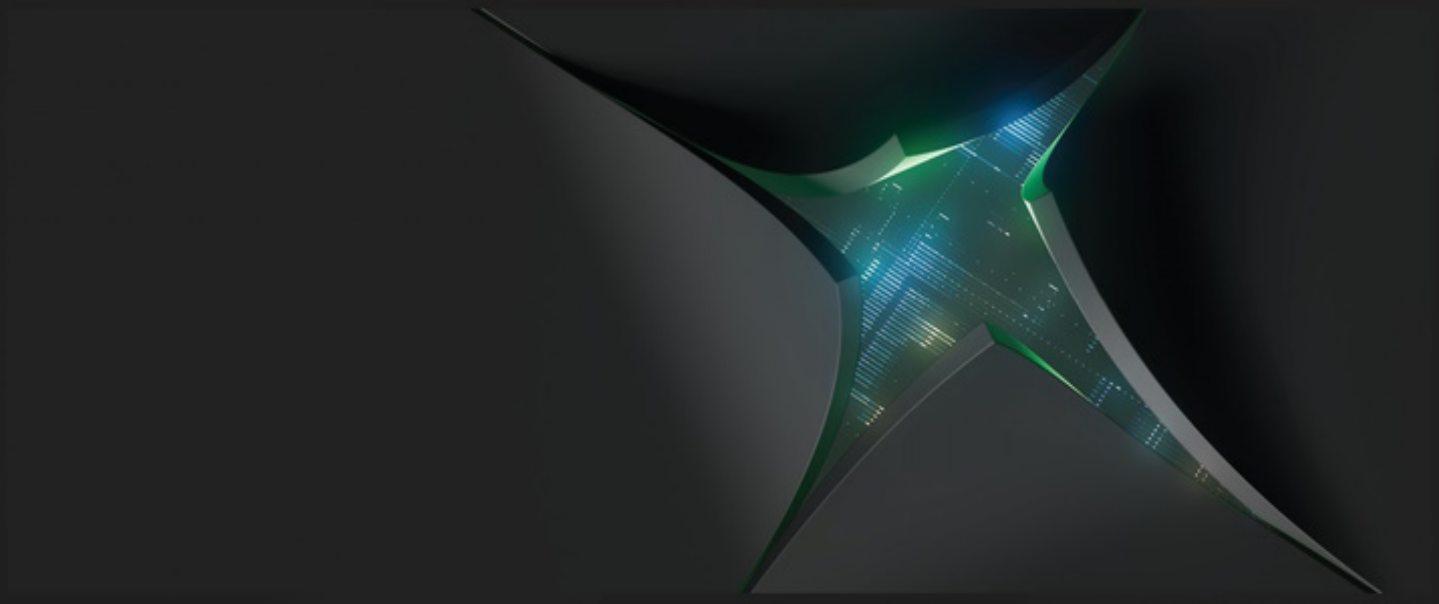

2ND EDITION

Mastering Linux Administration

Take your sysadmin skills to the next level by configuring and maintaining Linux systems

ALEXANDRU CALCATINGE
JULIAN BALOG

Mastering Linux Administration

Group Product Manager: Pavan Ramchandani

Publishing Product Manager: Prachi Sawant

Book Project Manager: Ashwini Gowda

Senior Editors: Shruti Menon and Adrija Mitra

Technical Editor: Irfa Ansari

Copy Editor: Safis Editing

Proofreader: Safis Editing

Indexer: Rekha Nair

Production Designer: Jyoti Kadam

Senior DevRel Marketing Coordinator: Marylou De Mello

First published: June 2021

Second edition: March 2024

Production reference: 1230224

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul’s Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83763-069-1

www.packtpub.com

This is to everyone who did not believe in me, and to everyone who believed in me.

To my mother and father, my twin sister, my beloved Uca, and my truthful friends.

In memory of my late grandfather, Miron, who would have loved to see this new book finished before he passed away.

Contributors

About the author

Alexandru Calcatinge is an open-minded architect with a background in computer science and mathematics. He is a senior university lecturer with a PhD in urban planning from Ion Mincu University of Architecture and Urban Planning and a postgraduate degree in DevOps from Caltech's Center for Technology and Management Education (CTME). He teaches students about architectural programming and development and open source technologies. He has authored five books on architecture and urban planning and numerous scientific articles on urban and rural development. Alex was certified as a Linux trainer in 2017. He loves the DevOps philosophy and the possibilities that cloud technologies bring for the future. He is also a certified programming analyst, computer network administrator, trainer, designer, and life coach.

I want to thank the people who have been close to me and supported me unconditionally, especially my parents, my twin sister, and my friends. I want to express my deepest gratitude to my one and only Uca for sticking by my side during a time of turmoil, and for her constant support and trust in me.

Julian Balog is a senior software engineer with more than 15 years of experience in the industry. Currently, his work primarily focuses on application delivery controllers, containerized workflows, networking, and security. With a never-ending passion for Linux and open-source technologies, Julian is always in pursuit of learning new things while solving problems and making things work through simple, efficient, and practical engineering. He lives with his wife, two children, and an Aussie-doodle in the greater Seattle area, Washington.

The authors would like to thank the wonderful editorial and production team at Packt for their professional leadership, dedication, and guidance throughout the writing of this book. We are indebted to them for many helpful suggestions and the comprehensive revision of the drafts. We are also grateful to the reviewers for their critical comments. We could not have hoped for a better team and support.

About the reviewers

Himanshu Sharma has nearly 18 years of experience in designing, architecting, and developing cloud and network software. He previously worked for some of the biggest companies such as Brocade and Juniper and start-ups such as Netskope. Currently, he is working with Netskope as a Principal Engineer, responsible for Netskope's security service offering. He designed, architected, and developed Netskope's advanced threat protection services from the ground up. He has a keen interest and experience in developing scalable cloud services with cutting-edge technologies. His favorite hobbies are skiing and playing video games.

I would like to thank my wife, Puja, who gave me all her support, and my two loving and beautiful daughters, Navya and Jaanvi.

Also, I want to thank my brother, Sudhanshu, for always having my back, and my parents for all their sacrifices to get me where I am today.

Dennis Salamanca is a passionate technology enthusiast with a solid track record of over 12 years in the IT industry. Throughout his career, he has had the privilege of working with renowned industry leaders such as Amazon, VMware, Microsoft, and Hewlett-Packard Enterprise. His dedication to continuous learning is reflected in an extensive collection of over 15 technical certifications spanning various domains including cloud, storage, Linux, Kubernetes, and virtualization. Notably, he is actively involved in the development team for Linux+ and Cloud+ certifications and proudly contributes as a member of the esteemed CompTIA Linux and Cloud Subject Matter Experts and Technical Advisory Committee.

I would like to acknowledge my wife and family for all their support over these years. Without their motivation and support, nothing would've been possible.

Table of Contents

The Linux Shell and Filesystem

3

Linux Software Management

8

Linux Shell Scripting

Further reading

Part 3: Server Administration

11

Working with Virtual Machines

Part 4: Cloud Administration

14

Short Introduction to Cloud Computing

Preface

Mastering Linux Administration provides the ultimate coverage of modern server and cloud administration technologies.

Technology evolves at an unprecedented speed, and Linux and related technologies are at the forefront of innovation. This makes it really hard to keep up and learn new things. Present Linux administrators need to know about more than just Linux, thus containerization and cloud technologies are essential for the future DevOps expert.

Linux is the operating system that powers almost everything, from IoT to personal computers to servers, and is the foundation for all cloud technologies. It enables you to master the cloud through the power of the command line.

You will begin by learning about the command line, working with files, processes, users, packages, and filesystems, then you will begin administering network services and hardening security, and finally, you will learn about cloud computing, containers, and orchestration. You will learn how to work at the command line, learn about the most important Linux commands, and master users, processes, and services administration. You will also learn how to harden Linux security using iptables. At the end, you will work with containers, hypervisors, virtual machines, Ansible, and Kubernetes and learn how to deploy Linux on AWS and Azure. By the end of this book, you will have mastered Linux and you will be confident in working with Linux from bare metal to the cloud, in a pure DevOps fashion.

Who this book is for

This book is for Linux administrators who want to understand the fundamentals, as well as modern concepts of Linux system administration. Windows system administrators looking to extend their knowledge to the Linux OS will also benefit from this book.

What this book covers

Chapter 1, *Installing Linux*, shows you how to install Linux on physical hardware (bare-metal) and inside a virtual machine in Windows. As we are targeting future Linux system administrators, the command line will be used most of the time, with little reference to the GUI. The future Linux professional will learn how to install Linux and how the boot process works.

Chapter 2, *The Linux Shell and Filesystem*, teaches you how to use the command line and introduces you to the most widely used commands in Linux. You will learn about the structure of a basic command, how the Linux filesystem is organized, the structure of the Linux operating system, and the structure of a file. By the end of the chapter, you will also know how to use VI/VIM, one of the widely used command-line text editors in Linux.

Chapter 3, *Linux Software Management*, explains how to use specific software management commands, how software packages work depending on the distribution of choice, and how to build your own packages.

Chapter 4, *Managing Users and Groups*, shows you how to manage user accounts in Linux. This is one of the most important tasks a Linux system administrator should master. You will be introduced to the general concepts, the specific files for user administration, and how to manage accounts. By the end of the chapter, you will know how to work with permissions and how to change them, and you will understand the special permissions and attributes.

Chapter 5, *Working with Processes, Daemons, and Signals*, explores processes, signals, and services in Linux. You will learn how to manage them, how to use them, and what the differences are between them.

Chapter 6, *Working with Disks and Filesystems*, teaches you how to manage disks and filesystems, understand storage in Linux, use **Logical Volume Management** (**LVM**) systems, and how to mount and partition.

Chapter 7, *Networking with Linux*, discusses how networking works in Linux, including the key concepts and how to configure your network from the command line and GUI.

Chapter 8, *Linux Shell Scripting*, shows you how to create and use Bash shell scripts for task automation in Linux. This will prove an invaluable asset for any system administrator.

Chapter 9, *Securing Linux*, delves into advanced topics of Linux security. You will learn how to work with SELinux and AppArmor.

Chapter 10, *Disaster Recovery, Diagnostics, and Troubleshooting*, shows you how to do a system backup and restore in a disaster recovery scenario. Also, you will learn how to diagnose and troubleshoot a common array of problems.

Chapter 11, *Working with Virtual Machines*, describes how to set up and work with KVM virtual machines on Linux.

Chapter 12, *Managing Containers with Docker*, introduces containers and discusses how to use Docker-specific tools to deploy your applications.

Chapter 13, *Configuring Linux Servers*, shows you how to configure different types of Linux servers, from **Domain Name System** (**DNS**), **Dynamic Host Configuration Protocol** (**DHCP**), **Secure Shell** (**SSH**), **Samba** file-sharing servers, and **Network File System** (**NFS**). This is one of the core foundations for any good Linux system administrator.

Chapter 14, *Short Introduction to Computing*, covers the basics of cloud computing. You will be presented with core technologies such as **infrastructure-as-a-service** (**IaaS**), **platform-as-a-service** (**PaaS**), **containers-as-a-service** (**CaaS**), **DevOps**, and cloud management tools.

Chapter 15, *Deploying to the Cloud with AWS and Azure*, explains how to deploy Linux to AWS and Azure.

Chapter 16, *Deploying Applications with Kubernetes*, teaches you how to use Kubernetes to monitor and secure your deployments and how to manage your containers and networks. You will learn what Kubernetes is and how to use its diverse community approaches.

Chapter 17, *Infrastructure and Automation with Ansible*, introduces Ansible, including how to configure it, and how to manage playbooks, modules, and servers. At the end of this chapter, you will be a master of automation.

To get the most out of this book

You will need Ubuntu Linux LTS or Debian Linux to perform the examples in this book. No prior knowledge of Linux is necessary.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/PacktPublishing/Mastering-Linux-Administration-Second-Edition. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "To check the contents of a binary `deb` package, you can use the `ar` command."

A block of code is set as follows:

```
spec:
     replicas: 1
```

Any command-line input or output is written as follows:

```
$ sudo zypper search nmap
```

When we wish to draw your attention to a particular part of a code block or command, the relevant lines or items are set in bold:

```
uid=1004(alex2) gid=1100(admin)
groups=1100(admin),1200(developers),1300(devops),1400(managers)
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Navigate to **Virtual Machines**, select your instance, and click **Size** under **Availability + Scale**."

TIPS OR IMPORTANT NOTES

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Mastering Linux Administration*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/9781837630691

2. Submit your proof of purchase

3. That’s it! We’ll send your free PDF and other benefits to your email directly

Part 1:Basic Linux Administration

In this first part, you will master the Linux command line and basic administrative tasks, such as managing users, packages, files, services, processes, signals, and disks.

This part has the following chapters:

1

Installing Linux

Recent years have been marked by a significant rise in the adoption of **Linux** as the operating system of choice for both server and desktop computing platforms. From enterprise-grade servers and large-scale cloud infrastructures to individual workstations and small-factor home appliances, Linux has become an ever-present platform for a wide range of applications.

The prevalence of Linux, perhaps now more than ever, brings into the spotlight much-needed administration skills for a growing community of system administrators and developers. In this book, we take a practical approach to Linux administration essentials, with the modern-day system administrator, DevOps team member, and developer in mind.

In this second edition, we will adopt a slightly different approach to installing Linux. As this is a book meant for more advanced readers, we will no longer discuss the basic aspects of installing the operating system in such detail as in the first edition. The information has been updated to the most relevant aspects available as of the beginning of 2023 with regard to operating system versioning.

In this first chapter, we'll guide you through the Linux installation process, either on physical hardware (bare metal) or using a **Virtual Machine** (**VM**).

Here are the topics we cover in this chapter:

- Introducing the Linux operating system
- Installing Linux – the basics
- Enabling the Windows Subsystem for Linux
- Installing Linux – the advanced stages
- Linux distributions – a practical guide

Technical requirements

We will use the following platforms and technologies in this chapter:

- Linux distributions: Ubuntu
- VM hypervisors: Oracle VM VirtualBox, VMware Workstation Player, and Hyper-V
- VM host platforms: Windows 11 (equally applicable on macOS)

Introducing the Linux operating system

Linux is a relatively modern operating system created in 1991 by Linus Torvalds, a Finnish computer science student from Helsinki. Originally released as a free and open source platform prohibiting commercial redistribution, Linux eventually adopted the GNU **General Public Licensing** (**GPL**) model in 1992. This move played a significant role in its wide adoption by the developer community and commercial enterprises alike. It is important to note that the Free Software Foundation community distinctly refers to Linux operating systems (or distributions) as **GNU/Linux** to emphasize the importance of GNU for free software.

Initially made for Intel x86 processor-based computer architectures, Linux has since been ported to a wide variety of platforms, becoming one of the most popular operating systems currently in use. The genesis of Linux could be considered the origin of an open source alternative to its mighty predecessor, Unix. This system was a commercial-grade operating system developed at the AT&T Bell Labs research center by Ken Thompson and Dennis Ritchie in 1969.

Exploring Linux distributions

A Linux operating system is typically referred to as a **distribution**. A Linux distribution, or **distro**, is the installation bundle (usually an ISO image) of an operating system that has a collection of tools, libraries, and additional software packages installed on top of the Linux **kernel**. A kernel is the core interface between a computer's hardware and its processes, controlling the communication between the two and managing the underlying resources as efficiently as possible.

The software collection bundled with the Linux kernel typically consists of a bootloader, shell, package management system, graphical user interface, and various software utilities and applications.

The following diagram is a simplified illustration of a generic Linux distribution architecture:

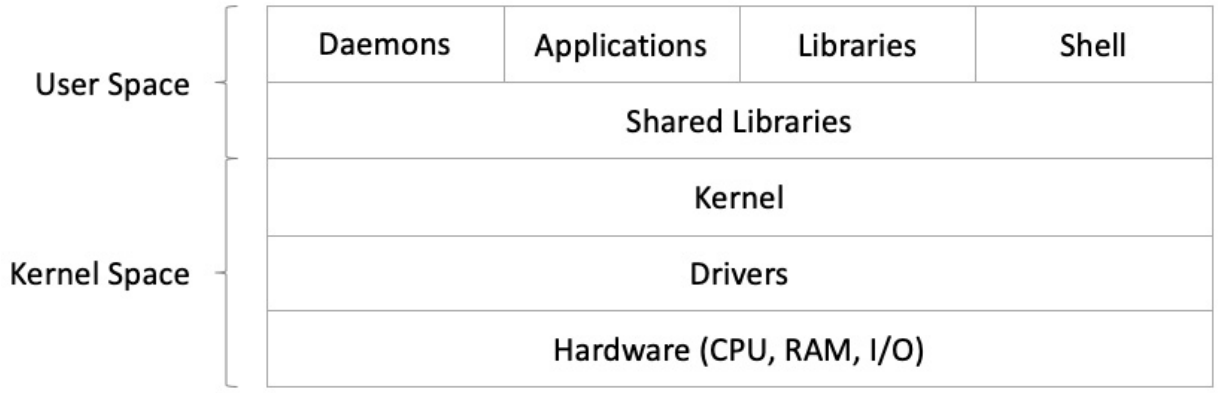

Figure 1.1 – Simplified view of a generic Linux architecture

There are hundreds of Linux distributions currently available. Among the oldest and arguably most popular ones are **Debian**, **Fedora**, **openSUSE**, **Arch Linux**, and **Slackware**, with many other Linux distributions either based upon or derived from them. Some of these distros are divided into commercial and community-supported platforms.

IMPORTANT NOTE

*As writing this second edition, CentOS became a rolling release and is the base from which future **Red Hat Enterprise Linux** (**RHEL**) versions are derived. Its place was taken by other free community distributions that use the RHEL binaries. Among those, **Rocky Linux** is a good example, and we will reference it throughout this book. One other community distribution based on RHEL is **AlmaLinux**.*

One of the key differences among Linux distributions is the package management system they use and the related Linux package format. We'll get into more detail on this topic in *Chapter 3*. For now, the focus is on choosing the right Linux distribution based on our needs. But before being able to decide, you should first know a little about some of the most widely used distributions. Therefore, in the next section, we will briefly present to you some Linux distros.

Common Linux distributions

This section summarizes the most popular and common Linux distributions at the time of writing this edition, with emphasis on their package manager type. Most of these distros are free and open source platforms. Their commercial-grade variations, if any, are noted:

- **Fedora, CentOS Stream and RHEL**: CentOS and its derivatives use **Red Hat Package Manager** (**RPM**) as their package manager. **CentOS Stream**, now a rolling release distribution, is based on the open source Fedora project. It is suited to both servers and workstations. RHEL is a commercial-grade version derived from CentOS Stream, designed to be a stable platform with long-term support. The community distribution that uses RHEL binaries is Rocky Linux.
- **Debian**: The package manager for Debian and most of its derivatives is **Debian Package** (**DPKG**). Debian is releasing at a much slower pace than other Linux distributions, such as Linux Mint or Ubuntu, for example, but it's relatively more stable.
- **Ubuntu**: Ubuntu uses **Advanced Package Tool** (**APT**) and DKPG as package managers. Ubuntu is one of the most popular Linux distributions, releasing every 6 months, with more stable **Long Term Support** (**LTS**) releases every other year.
- **Linux Mint**: Linux Mint also uses APT as its package manager. Built on top of Ubuntu, Linux Mint is mostly suitable for desktop use, with a lower memory usage than Ubuntu (with the Cinnamon desktop environment, compared to Ubuntu's GNOME). There's also a version of Linux Mint built directly on top of Debian, called **Linux Mint Debian Edition** (**LMDE**).
- **openSUSE**: openSUSE uses **RPM**, **Yet another Setup Tool** (**YaST**), and **Zypper** as package managers. openSUSE has two versions available: one is called Tumbleweed and is a rolling release, a leading-edge Linux distribution; the other is Leap, a regular release version, which uses the same code base as SUSE Linux Enterprise. Both versions are suited to desktop and server environments. SUSE Linux Enterprise Server is a commercial-grade platform. openSUSE was regarded as one of the most user-friendly desktop Linux distributions before the days of Ubuntu.

IMPORTANT NOTE

*In this book, our focus is mainly on the Linux distributions that are widely used in both community and commercial deployments, such as **Ubuntu**, **Fedora/Rocky Linux**, and **openSUSE**. Most of the examples in this book are equally appliable to any Linux distro. We will specify which one we use for given examples or scenarios.*

Now that you know a fair amount of information about the most common Linux distros, in the next section we will give you some hints on how to choose a Linux distribution.

Choosing a Linux distribution

There are many aspects involved in selecting a Linux distribution, based on various functional requirements. A comprehensive analysis would be far beyond the scope of this chapter. However, considering a few essential points may help with making the right decision:

- **Platform**: The choice between a server, a desktop, or an embedded platform is probably one of the top decisions in selecting a Linux distribution. Linux server platforms and embedded systems are usually configured with the core operating system services and essential components required for specific applications (such as networking, HTTP, FTP, SSH, and email), mainly for performance and optimization considerations. On the other hand, Linux desktop workstations are loaded (or pre-loaded) with a relatively large number of software packages, including a graphical user interface for a more user-friendly experience. Some Linux distributions come with server and desktop flavors (such as **Ubuntu**, **Fedora**, and **openSUSE**), but most distros have a minimal operating system, with further configuration needed (such as **Rocky Linux**, and **Debian**). Usually, such distributions would be good candidates for Linux server platforms. There are also Linux distributions specifically designed for desktop use, such as **elementary OS**, **Pop!_OS**, or **Deepin**. For embedded systems, we have highly optimized Linux distros, such as **Raspbian** and **OpenWRT**, to accommodate small-form factor devices with limited hardware resources.
- **Infrastructure**: Today we see a vast array of application and server platform deployments spanning from hardware and local (on-premises) data centers to hypervisors, containers, and cloud infrastructures. Weighing a Linux distribution against any of these types of deployments should take into consideration the resources and costs involved. For example, a multi-CPU, large-memory, and generally high-footprint Linux instance may cost more to run in the cloud or a **Virtual Private Server** (**VPS**) hosting infrastructure. Lightweight Linux distributions take fewer resources and are easier to scale in environments with containerized workloads and services (for instance, with Kubernetes and Docker). Most Linux distributions now have their cloud images available for all major public cloud providers (for instance, Amazon AWS, Microsoft Azure, and Google Compute Engine). Docker container images for various Linux distributions are available for download on Docker Hub (https://hub.docker.com). Some Docker images are larger (heavier) than others. For example, the **Ubuntu Server** Docker image outweighs the **Alpine Linux** Docker image considerably, and this may tip the balance when choosing one distribution over the other. Also, to address the relatively new shift to containerized workflows and services, some Linux distributions offer a streamlined or more optimized version of their operating system to support the underlying application infrastructure. For example, Fedora features the **Fedora CoreOS** (for containerized workflows) and **Fedora IoT** (for Internet of Things ecosystems).
- **Performance**: Arguably, all Linux distributions can be tweaked to high-performance benchmarks in terms of CPU, GPU, memory, and storage. Performance should be considered very closely with the platform and the application of choice. An email backend won't perform very well on a Raspberry Pi, while a media streaming server would do just fine (with some external storage attached). The configuration effort for tuning the performance should also be taken into consideration. **Rocky Linux**, **Debian**, **openSUSE**, **Fedora** and **Ubuntu** all come with server and desktop versions reasonably optimized for their use. The server versions can be easily customized for a particular application or service, by only limiting the software packages to those that are essential for the application. To further boost performance, some would go to the extent of recompiling a lightweight Linux distro (for instance, **Gentoo**) to benefit from compiler-level optimizations in the kernel for specific subsystems (for instance, the networking stack or user permissions). As with any other criteria, choosing a Linux distribution based on some application or platform performance is a balancing act, and most of the time, common Linux distros will perform exceptionally well.

- **Security**: When considering security, we have to keep in mind that a system is only as secure as its weakest link. An insecure application or system component would put the entire system at risk. Therefore, the security of a Linux distribution should be scrutinized as it pertains to the related application and platform environment. We can talk about *desktop security* for a Linux distro serving as a desktop workstation, for example, with the user browsing the internet, downloading media, installing various software packages, and running different applications. The safe handling of all these operations (against malware, viruses, and intrusions) would make for a good indicator of how secure a system can be. There are Linux distros that are highly specialized in application security and isolation and are well suited for desktop use: **Qubes OS**, **Kali Linux**, **Whonix**, **Tails**, and **Parrot Security OS**. Some of these distributions have been developed for penetration testing and security research.

On the other hand, we may consider the *server security* aspect of Linux server distributions. In this case, regular operating system updates with the latest repositories, packages, and components would go a long way to securing the system. Removing unused network-facing services and configuring stricter firewall rules are further steps for reducing the possible attack surface. Most Linux distributions are well equipped with the required tools and services to accommodate this reconfiguration. Opting for a distro with *frequent* and *stable* upgrades or release cycles is generally the first prerequisite for a secure platform (for instance, **Rocky Linux**, **RHEL**, **Ubuntu LTS**, or **SUSE Enterprise Linux**).

- **Reliability**: Linux distributions with aggressive release cycles and a relatively large amount of new code added in each release are usually less stable. For such distros, it's essential to choose a *stable* version. **Fedora**, for example, has rapid releases, being one of the fastest-progressing Linux platforms. Yet, we should not heed the myths claiming that Fedora or other similar fast-evolving Linux distros, such as openSUSE Tumbleweed, are less reliable. Don't forget, some of the most reliable Linux distributions out there, **RHEL** and **SUSE Linux Enterprise (SLE)**, are derived from Fedora and openSUSE, respectively.

There's no magic formula for deciding on a Linux distribution. In most cases, the choice of platform (be it server, desktop or IoT) combined with your own personal preferences is what determines the Linux distribution to go for. With production-grade environments, most of the previously enumerated criteria become critical, and the available options for our Linux platform of choice would be reduced to a few industry-proven solutions.

IMPORTANT NOTE

*In this book, our focus is mainly on the Linux distributions that are widely used in both community and commercial deployments, such as **Ubuntu**, **Fedora/Rocky Linux**, and **openSUSE**. That said, most of the examples in this book are equally applicable to any Linux distro. We will specify which one we use for given examples or scenarios.*

Now that you know a bit about what a Linux distribution is, along with the most commonly used ones and their use cases, in the following two sections we will present the basic and advanced aspects of Linux installation.

Installing Linux – the basics

This section serves as a quick guide for the basic installation of an arbitrary Linux distribution. For hands-on examples and specific guidelines, we use Ubuntu. We also take a brief look at different

environments hosting a Linux installation. There is an emerging trend of hybrid cloud infrastructures, with a mix of on-premises data center and public cloud deployments, where a Linux host can either be a bare-metal system, a hypervisor, a VM, or a Docker container.

In most of these cases, the same principles apply when performing a Linux installation. For detail on Docker-containerized Linux deployments, see *Chapter 13*.

In the following sections, we will show you how to install Linux on bare metal and on a Windows 11 host using different VM hypervisors, and using WSL. Installing on a macOS host is basically the same as installing on Windows using a VM hypervisor, and we will not cover that.

How to install Linux on bare metal

This section describes the essential steps required for a Linux installation on **bare metal**. We use this term when referring to hardware such as laptops, desktops, workstations, and servers. In a nutshell, the main steps are downloading the ISO image, creating bootable media, trying out the live mode, and finally, doing the installation.

The steps used here are equally applicable to virtual machine installations, as you will see in the following sections.

Step 1 – Download

We start by downloading our Linux distribution of choice. Most distributions are typically available in ISO format on the distribution's website. For example, we can download Ubuntu Desktop at https://ubuntu.com/download/desktop.

Using the ISO image, in the next step we can create the bootable media required for the Linux installation. We can also use the ISO image to install Linux in a VM, as shown in the next section.

Step 2 – Create the bootable media

As we install Linux on a PC desktop or workstation (*bare-metal*) system, the bootable Linux media is generally a CD/DVD or a USB device. With a DVD-writable optical drive at hand, we can simply burn a DVD with our Linux distribution ISO. But because modern-day computers, especially laptops, rarely come equipped with a CD or a DVD unit of any kind, the more common choice for bootable media is a USB drive.

IMPORTANT NOTE

*There's also a third possibility of using a **Preboot eXecution Environment** (**PXE**) boot server. PXE (pronounced pixie) is a client-server environment where a PXE-enabled client (PC/BIOS) loads and boots a software package over a local or wide area network from a PXE-enabled server. PXE eliminates the need for physical boot devices (CD/DVD, USB) and reduces the installation overhead, especially for a large number of clients and operating systems. Probing the depths of*

PXE internals is beyond the scope of this chapter, but we will give you a short introduction on how it works for Linux installations by the end of this chapter. A good starting point to learn more about PXE is https://en.wikipedia.org/wiki/Preboot_Execution_Environment.

A relatively straightforward way to produce a bootable USB drive with a Linux distribution of our choice is via using tools such a **UNetbootin** (https://unetbootin.github.io) or **Balena Etcher** (https://www.balena.io/etcher). Both UNetbootin and Etcher are cross-platform utilities, running on Windows, Linux, and macOS.

We will use Balena Etcher for this example of creating a bootable USB drive in Windows:

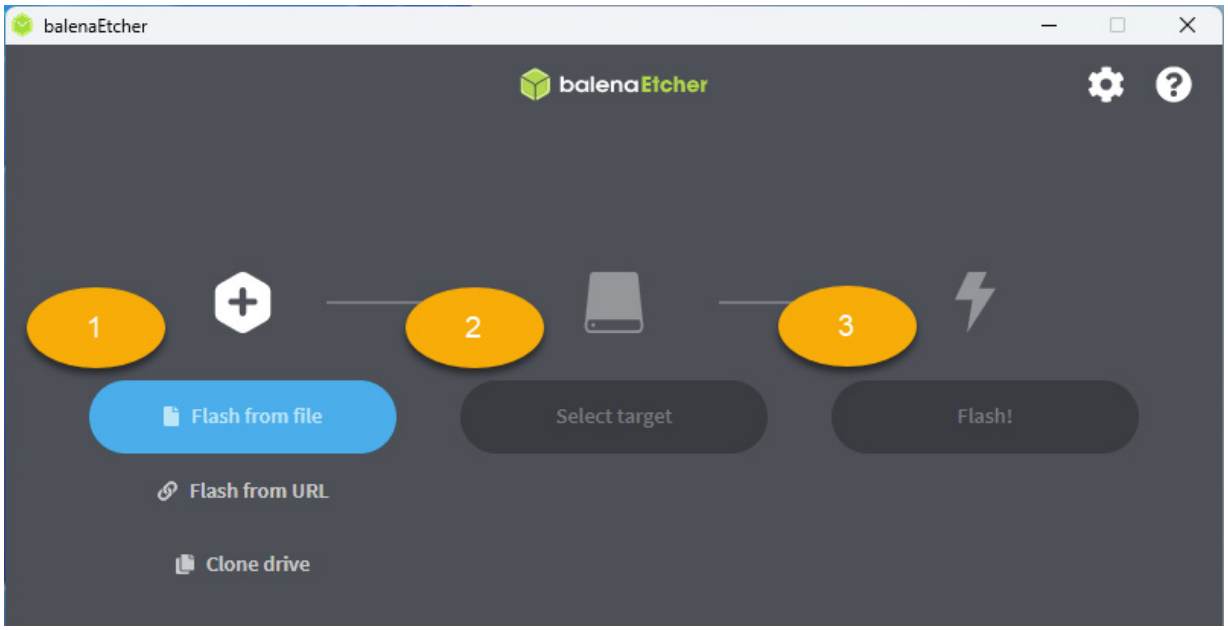

Figure 1.2 – Create a bootable USB drive with Balena Etcher

Here are the basic steps for creating a bootable USB drive with Ubuntu Desktop using Balena Etcher. We assume the Ubuntu Desktop ISO image has been downloaded and Etcher is installed (in our case on Windows 11):

1. Choose the ISO file with your Linux distribution of choice.
2. Select the USB target destination disk.
3. Flash the previously selected disk with the ISO of your choice.

The process should take a couple of minutes and the USB drive will be ready. Now, let's look at how we can take the bootable media for a spin.

Step 3 – Try it out in live mode

This step is optional.

Most Linux distributions have their ISO image available for download as *live* media. We say most because not all of them offer this option, at least not by default. Nevertheless, among those who do offer live media by default are **Ubuntu** and **Fedora**.

Once we have the bootable media created with our Linux distribution of choice, we can run a live environment of our Linux platform without actually installing it. In other words, we can evaluate and test the Linux distribution before deciding whether we want to install it or not. The live Linux operating system is loaded in the system memory (RAM) of our PC, without using any disk storage. We should make sure the PC has enough RAM to accommodate the minimum required memory of our Linux distribution.

When booting the PC from a bootable media, we need to make sure the boot order in the BIOS is set to read our drive with the highest priority. On a Mac, we need to press the *Option* key immediately after the reboot start-up chime and select our USB drive to boot from. When on a PC, make sure you access your BIOS interface and select the appropriate device for boot. Depending on your system, you will either have to press one of the *F2*, *F10*, *F12*, or *F1* keys after hitting *Enter*, or the *Delete* key, as a general rule. In some specific cases, there could be another *Function* key assigned for this. The keys that you need to press are usually specified at the bottom of the initial bootup screen.

Upon reboot, the first splash screen of our Linux distribution should give us the option of running in live mode, as seen in the following illustration for Ubuntu Desktop (**Try Ubuntu**):

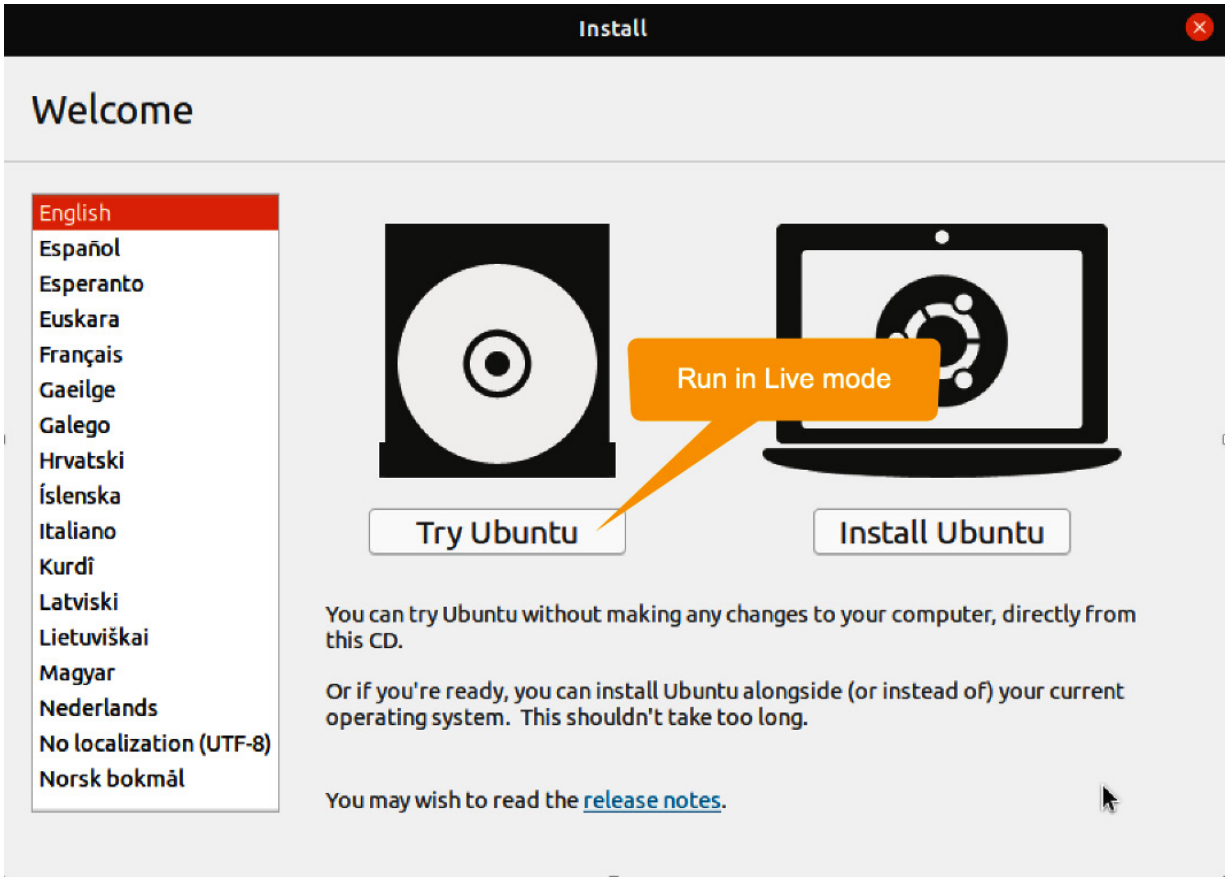

Figure 1.3 – Choosing live mode for Ubuntu Desktop

Next, let's take a look at the installation procedure of our Linux distro, using the bootable media.

Step 4 – Perform the installation

We start the installation of our Linux distribution by booting the PC from the bootable media created in *step 2*. To ensure the system can boot from our external device, we are sometimes required to change the boot order in the BIOS, especially if we boot from a USB drive. Do as specified in the previous paragraph to select the right boot drive.

In the following sections, we showcase the installation process of Ubuntu using its ISO images. We choose the Desktop and Server versions for Ubuntu and highlight the main differences between them. As a comparison, Rocky Linux and CentOS Stream come in a single flavor, in essence, a server platform with an optional graphical user interface. Similar to those, openSUSE offers one installation medium for both desktop and server installs. Fedora, on the other hand, has different installation mediums for desktop and server.

We will now walk you through the process of installing Linux inside a VM.

Linux in a VM

In each of the subsections in the *Installing Linux* section, we will also provide a brief guide on how to prepare a VM environment for the related Linux platform.

A VM is an isolated software abstraction of a physical machine. VMs are deployed on top of a **hypervisor**. A hypervisor provides the runtime provisioning and resource management of VMs. A couple of general-purpose hypervisors used are the following:

- **Oracle VM VirtualBox** (https://www.virtualbox.org)
- **VMware Workstation** (https://www.vmware.com/products/workstation-pro.html)
- **Hyper-V** (available only in Windows Pro, Enterprise, or Education)

The first two of these hypervisors are cross-platform virtualization applications and run on both Intel and AMD processor architectures on Windows, macOS, and Linux. The latter is only available on Windows Pro, versions 10 and 11.

IMPORTANT NOTE

*At the time of writing this book, hypervisors for the Apple silicon Macs are provided only by VMware Player and Parallels. Oracle VirtualBox is still in preview for the **Advanced RISC Machines** (**ARM**) architecture. Both solutions from VMware and Parallels are paid-for software on macOS, so you will need to purchase them in order to use them.*

The difference between installing Linux on a VM compared to a physical machine is minor. The notable distinction is related to the VM sizing and configuration steps, making sure the minimum system requirements of the Linux distribution are met. Thus, in the following sections we will install Ubuntu on VMware Workstation under Windows.

Please take into account that installing Linux on VMware Player under macOS is very similar, and we will not duplicate the process in this edition of the book. macOS functionality was discussed in the first edition of the book, but given the limited availability of hypervisors for the Apple silicon platform, we have decided to skip it in this edition. Regarding Linux availability on bare-metal Apple silicon Macs, you could visit Asahi Linux, a project that aims to bring a fully functional Linux distribution to Apple silicon computers. Asahi Linux is available at https://asahilinux.org/.

In the next section, we briefly illustrate the installation of Ubuntu Server LTS. If we plan to install Ubuntu in a VM, there are some preliminary steps required for provisioning the VM environment. Otherwise, we proceed directly to the *Installation* section.

VM provisioning using VMware Workstation

In the following steps, we will create a VM based on Ubuntu Server using VMware Workstation on Windows 11. At the time of writing, version 17 of the software is available for both free and commercial use.

1. The first step after initializing the hypervisor is to click on **Create a New Virtual Machine**. This will open a new window with the new virtual machine wizard, where you can select the ISO image for the Linux distribution you want to install.

2. Click **Browse** and then open the image from your hard drive or download destination.
3. Click **Next** and you will have to give a name to the new VM and choose a location on your disk for installation. We will leave the default destination as provided by the hypervisor and name the VM `Ubuntu Server 22.04.1`.
4. Click **Next**. In the following window that appears, you have to give the maximum disk size for the VM. By default, it is set to 20 GB as the recommended size for Ubuntu Server. We will leave it as is.
5. By clicking **Next** once more, a window with the VM settings is provided. By default, the hypervisor provides 2 CPU cores and 4 GB of RAM to the VM. You can click on the **Customize hardware** button to change the defaults, depending on your hardware availability. As a rule, we recommend having at least 16 GB of RAM on your system and an 8-core CPU to be able to create reasonable-sized VMs. When everything is set up as you want, click on the **Close** button on the lower right side of the window. You are now back to the main wizard window.
6. Click **Finish** to complete the setup and create and initialize the VM. In the following screenshot you can see the newly created VM, running inside VMware

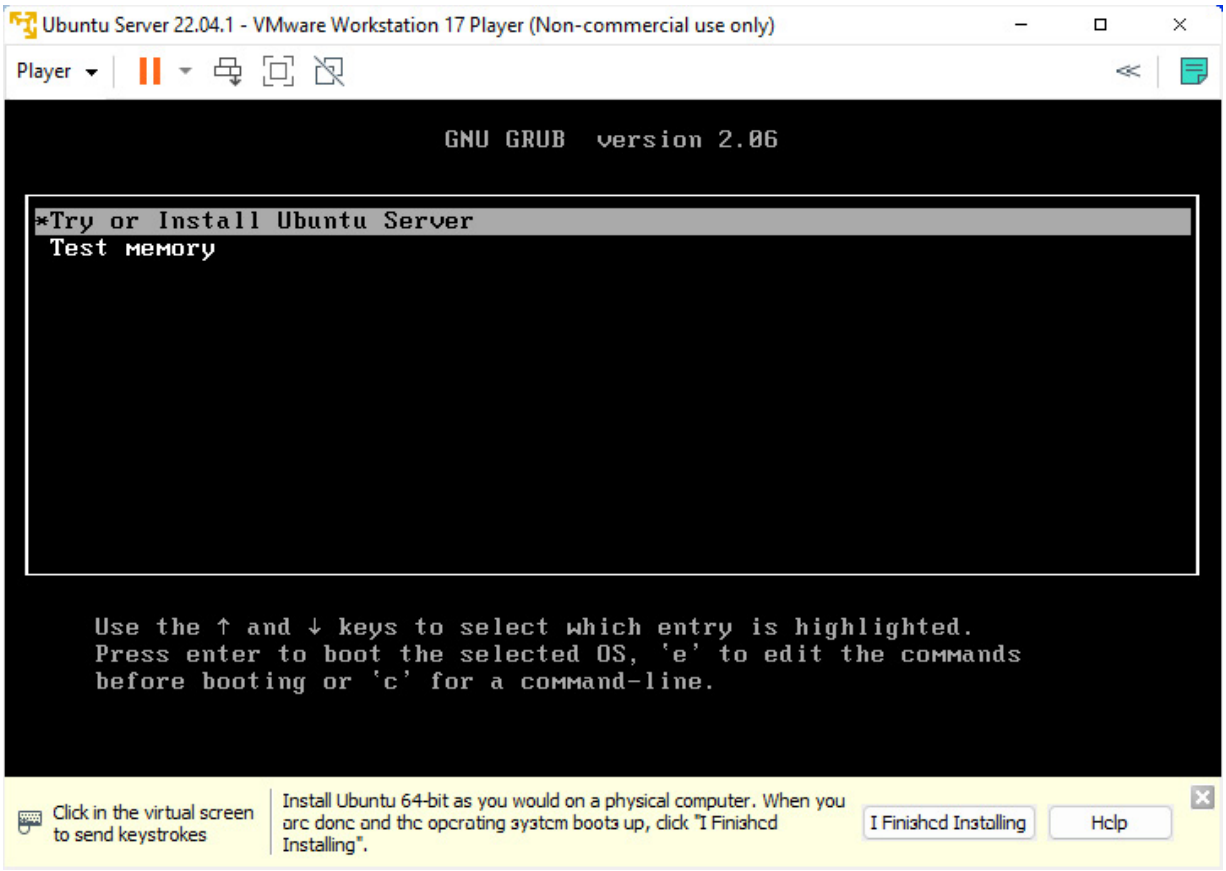

Figure 1.4 – Power-up and Linux installation on a new VM

Installation process

Here's the normal installation process for Ubuntu Server LTS, following the initial boot into setup mode:

1. The initial welcome screen prompts for the **language** of your choice. Select the one you prefer and press *Enter* on your keyboard.

2. You might be prompted to apply an *installer update* if available. You have the options to update the installer or continue without updating. We choose to update the installer if prompted.
3. If there is no update available, the next screen prompts you to select your **keyboard layout**. Select accordingly. In our case, it is English. Select **Done** and press *Enter*.
4. The next prompt asks you to choose the **base for the installation** from the following options: **Ubuntu Server** and **Ubuntu Server (minimized)**. You also have the option to search for **third-party drivers**. We choose **Ubuntu Server** and select the third-party driver option. You can make your way around the options on the screen using either *Tab* or the *Arrow* keys. To select an option, press the *Space* key. Select **Done** and press *Enter*.
5. The next screen will show you the **network connections**. If the defaults work for you, hit *Enter* to go to the next setup screen.
6. You will be asked about **proxy configuration**. If you don't require this, just hit *Enter* to go to the next screen.
7. You will now be asked to configure the default Ubuntu **mirror** for the repository archives. Edit this according to your location, or just leave the defaults provided by the installer. Press *Enter*.
8. The next screen prompts you to configure **storage** and **partitioning**. We will just use the entire 20 GB disk with the default settings, so select **Done** and press *Enter*.
9. A storage **configuration summary** is provided. If everything is according to your requirements, just hit *Enter*.
10. A **warning** will pop up, asking whether you are definitely happy with the settings and willing to continue with the installation. Hit *Enter*.
11. The next screen asks for your **profile information**, including your name, the server's name, the username, and password. Set those up and go to the next screen.
12. You will be asked to choose whether to **install an openSSH server** or not. Select the option to install openSSH. If you have any SSH key(s) you would like to import, you can provide them here. Once finished, go to the next screen.
13. You will be prompted to select and install **specific snap packages** for your new server installation. Depending on your requirements, you can install these now, or choose to manually install them later on. Among the provided packages are **`docker`**, **`microk8s`**, **`powershell`**, **`nextcloud`**, and **`livepatch`**. Select what you need to meet your requirements and continue to the next screen.
14. The **installation process** begins. This could take a couple of minutes. Be patient and wait for the reboot option to appear once the operating system is installed.

After you reboot, the login screen appears and you will be able to use your new Ubuntu Server VM from inside Windows 11 using VMware Workstation. We have now completed the Ubuntu Server installation.

Installing any other distribution is very similar to installing Ubuntu. When installing desktop variants, a graphical user interface will be available. In the preceding example, as we installed a server-specific operating system, the graphical user interface was missing, having just a minimal text-based interface.

We will not walk you through the installation process of any other distribution, but we will show you the Rocky Linux installation interface:

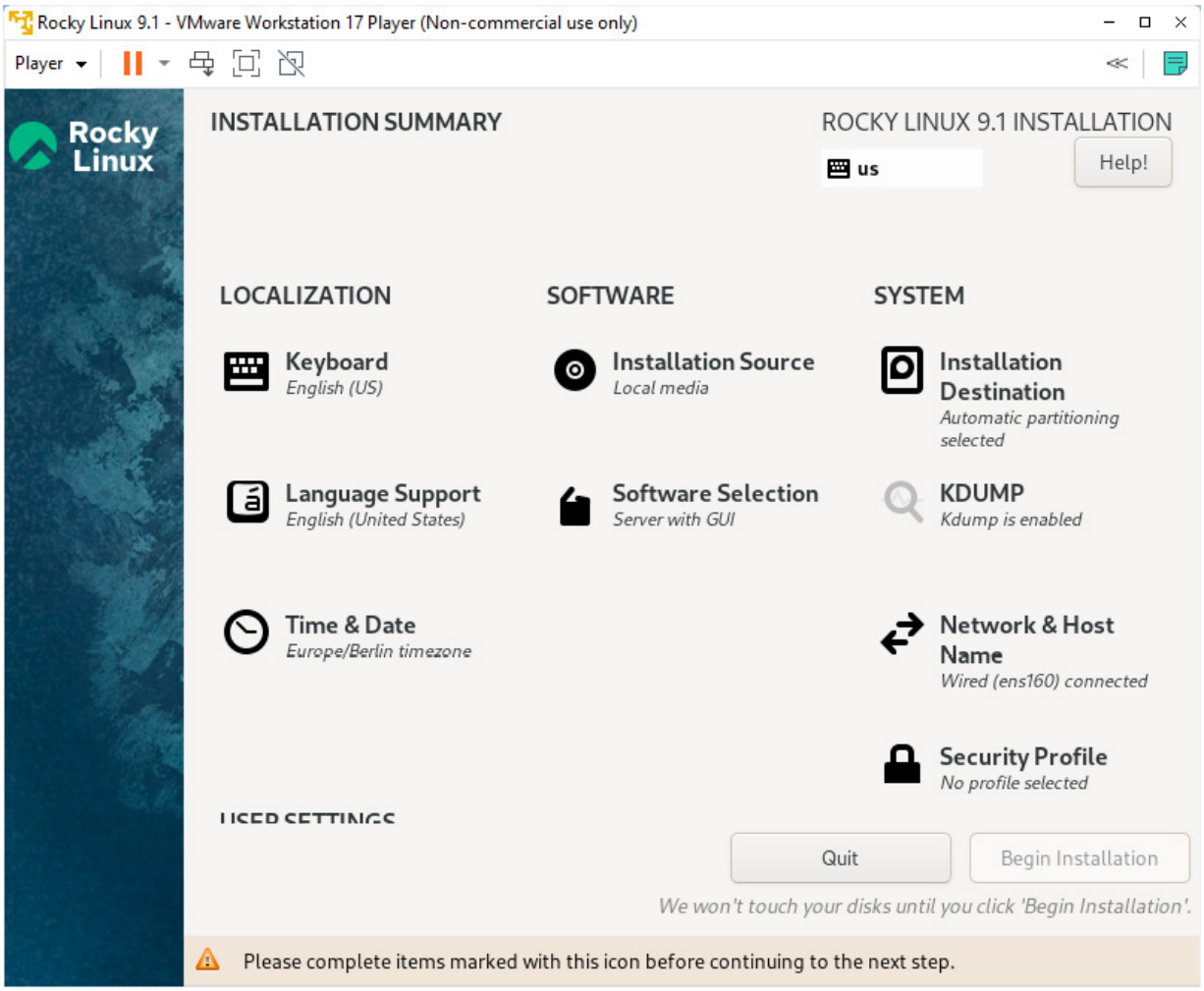

Figure 1.5 – The Rocky Linux installation GUI

So far, we have learned how to perform a basic installation of Linux. Along the way, we created a bootable USB flash drive for our installation media, most commonly used for Linux PC platform installations. We briefly covered VM-specific Linux environments using the VMware Workstation hypervisor for Windows 11.

In the following section, we'll learn how to install and run a Linux distribution on the Windows platform without the use of a standalone hypervisor by using the Windows Subsystem for Linux.

VM provisioning using Hyper-V

In the following steps, we will create a VM based on Ubuntu Server, using Microsoft's Hyper-V solution available on Windows 11 Pro.

The first step is to activate the Hyper-V hypervisor, as it is not activated by default. For this, we will need to go to **Windows Features** and select the **Hyper-V** checkbox, as shown in the following

figure. After activation, a restart is required.

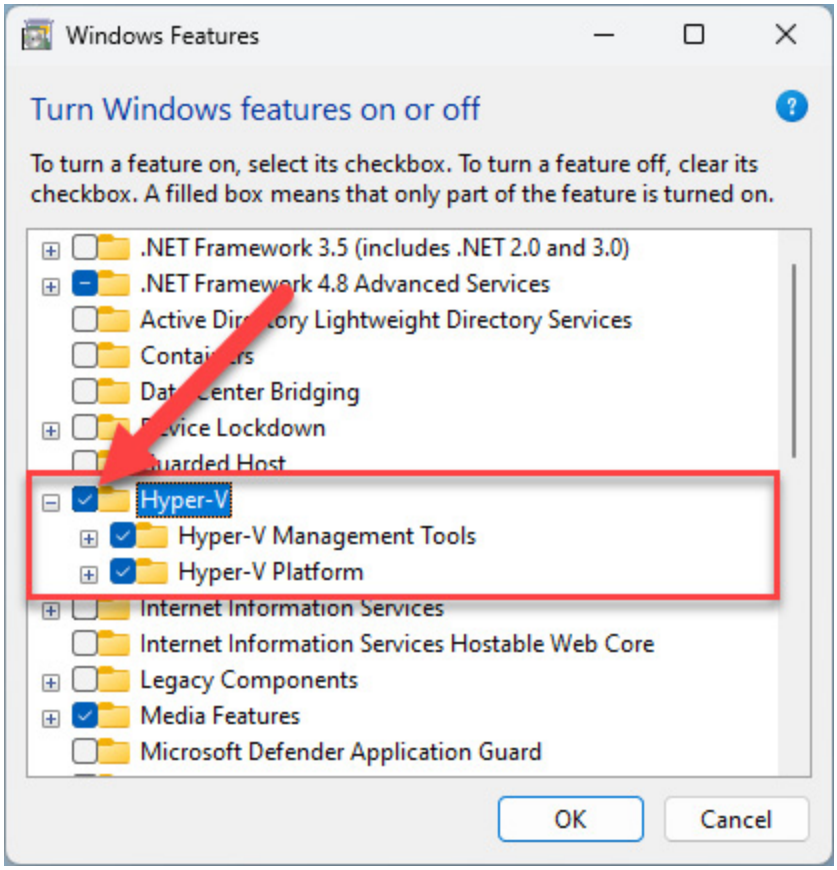

Figure 1.6 – Activating Hyper-V on Windows 11 Pro

To create a new VM, you will have to start **Hyper-V Manager**. The application has a three-pane interface. In the **Actions** pane on the right, you should see the **New** option. Click on it and select the **Virtual Machine…** option. This will open a new window where you can configure the new VM with the following steps:

1. Set the name and location; we will give it the name `Ubuntu` and leave the default location as is. Click **Next**.
2. Set the generation of the VM. You have two options, **Generation 1** and **Generation 2**. The second option will be suitable for UEFI-based BIOS and Network installation (PXE). We will select **Generation 1** and click **Next**.
3. Specify the amount of RAM. By default, this is set to 4 GB minimum, with the option of dynamic memory selected. We will leave the default as is. Click **Next**.
4. Configure networking by selecting the appropriate option from the dropdown. You have three options: **Not Connected**, **Default Switch**, **WSL**; we will select **Default Switch** and click **Next**.
5. Configure a virtual hard drive by setting the size and location. Click **Next**.
6. In the following window, you have the option to install an operating system now or at a later time. We will select the Ubuntu Desktop ISO image from our location of choice and click **Next**.

7. The following window shows the summary of the VM's configuration. You can change any of it by going back from here. Once done, click the **Finish** button and the VM will be created.

The following screenshot shows the new Ubuntu VM running inside Hyper-V:

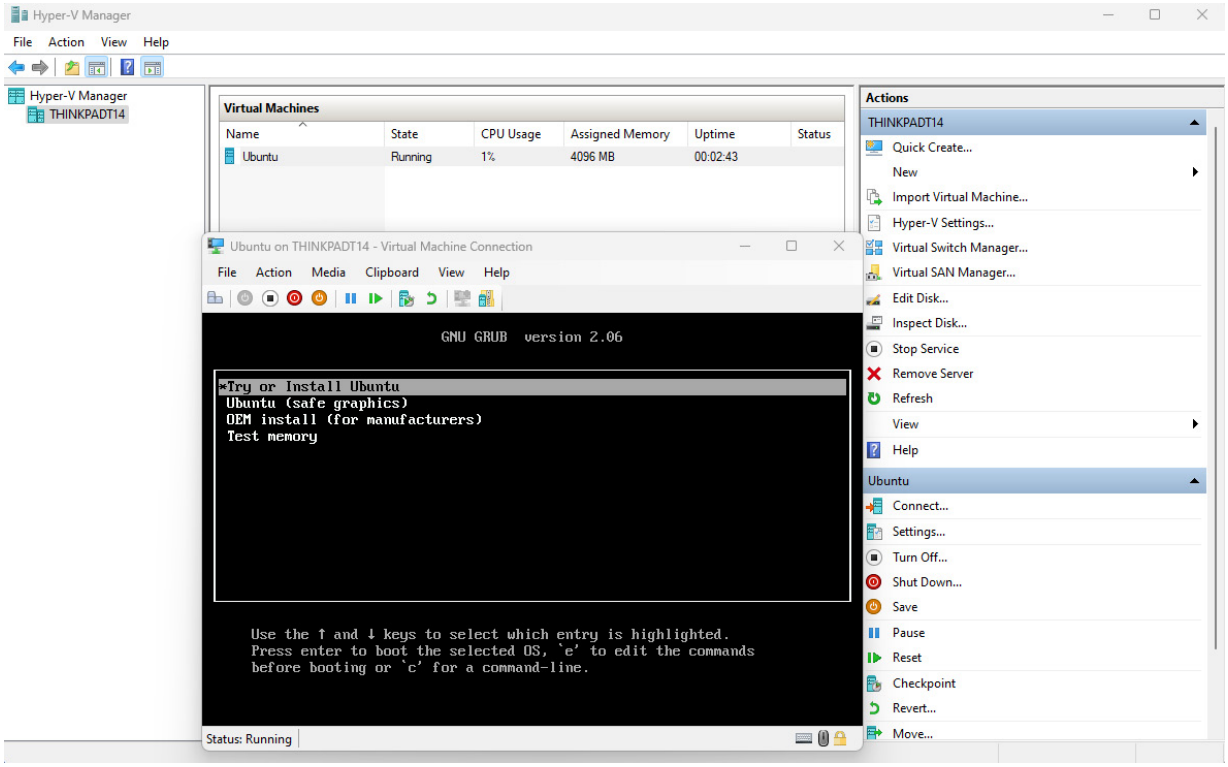

Figure 1.7 – New Ubuntu VM using Hyper-V

The installation process is similar to the one shown in the previous section, thus we will not reproduce it here again. In the next section, we will use another hypervisor, this time from Oracle.

VM provisioning using Oracle's VirtualBox

Oracle's **VirtualBox** is a free-to-use piece of software that is multi-platform, available on Windows, macOS and Linux. We will show you how to create a Linux VM from Windows 11. We assume that you have VirtualBox already installed. Once you start it, a user-friendly interface is available. The following steps are used to create a new VM. We will use Fedora Workstation for our example:

1. Click on the **New** icon to start the process of creating a new VM. This will open a new window where information about the VM's name and operating system type and ISO location need to be provided. This new window is in **Guided Mode** by default. You have the option on the lower right side of the window to choose the **Expert Mode**. This will give you more control over the creation process.

2. Provide all the needed information. In our case, we will use Fedora, so we will give it the name **`Fedora`**. Point to the ISO file's location and the type of the operating system will change automatically. If you are in **Expert Mode**, you will have some more auto-hidden sections for **Unattended Install**, **Hardware**, and **Hard Disk** options.

3. Because we are installing Fedora, the **Unattended Install** section is grayed out (see *Figure 1.11*). This option is supported by only a few operating systems (Ubuntu, RHEL, Oracle Linux, and Windows).
4. In the **Hardware** section, we will provide the amount of system memory and processors we want the new VM to have. Choose according to your hardware resources, but keep in mind that each operating system has specific system requirements. In our case, we will choose 4 GB of RAM and 2 vCPUs.
5. The **Hard Disk** section is where you choose the amount of disk space for the VM's hard disk. Again, choose according to your resources, but keep in mind that a minimal amount is required to meet specific system requirements. In our opinion, a minimum of 20 GB of hard disk space should be provided. Choose the location of the virtual hard drive and click **Finish**.
6. The VM will be created and the window will close, bringing you back to the initial VirtualBox window. Here, you will see all the relevant information about the VM. To power it on, just click on the **Start** button (the one with a big green right arrow).
7. A new window with the VM will appear.

The following screenshot shows the VM creation window inside VirtualBox:

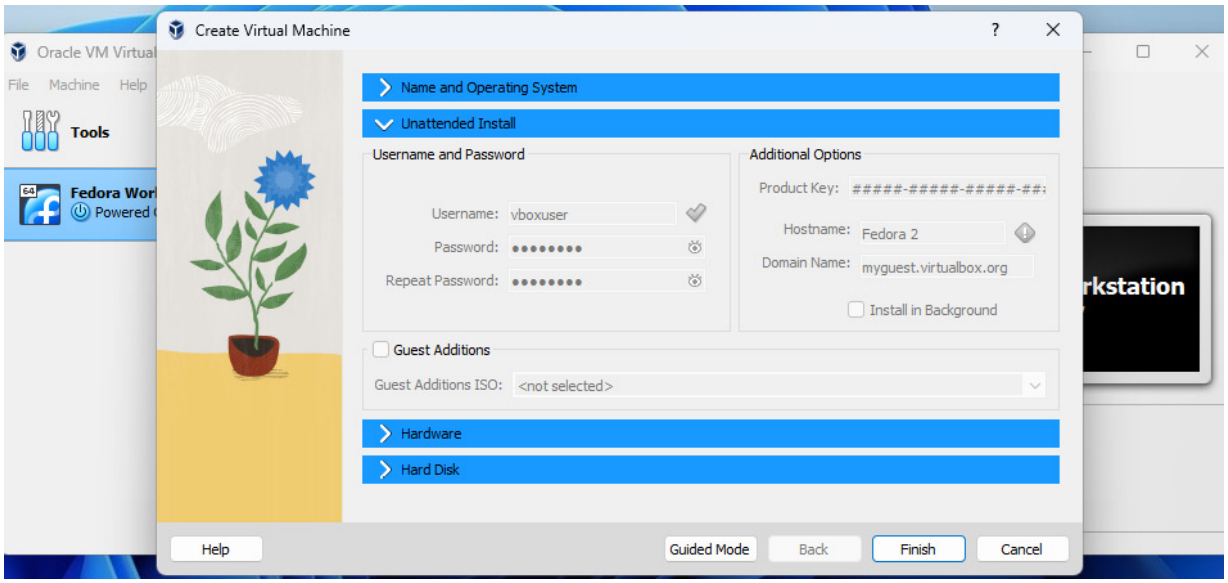

Figure 1.8 – VirtualBox interface

As you can see, creating Linux VMs with all three major hypervisors available (from VMware, Oracle, and Microsoft) is very straightforward and relatively easy to do. No matter which solution you use, the process of installing Linux in a VM is the same.

Aside from VM provisioning, Microsoft Windows offers a relatively new way to run Linux, and this is by using the Windows Subsystem for Linux. We will show you how in the next section.

Enabling Windows Subsystem for Linux

Software developers and system administrators often face a tough decision in choosing the appropriate hardware and operating system platform for the specific requirements of their work or

environment. In the past, Windows professionals frequently discovered that some standard development tools, frameworks, or server components were available on the Linux or macOS platforms while lacking native support on Windows. **Windows Subsystem for Linux** (**WSL**) attempts to close this gap.

WSL is a Windows platform feature that provides a native GNU/Linux runtime along with the Windows desktop environment available for both versions 10 and 11 of Windows. WSL enables the seamless deployment and integration of select Linux distributions on top of the Windows kernel, without the need for a dedicated hypervisor. With WSL enabled, you can easily install and run Linux as a native Windows application.

IMPORTANT NOTE

Without WSL, we could only deploy and run a Linux distribution on a Windows platform by using a standalone hypervisor, such as Hyper-V, Oracle VM VirtualBox, or VMware Workstation. WSL eliminates the need for a dedicated hypervisor. At the time of writing, WSL is a Windows kernel extension with a hypervisor embedded.

In this section, we provide the steps required to enable WSL and run an Ubuntu distribution on Windows. Since Windows 11 version 21H2 and Windows 10 versions 21H2 and 22H2, WSL is available by default from the Windows Store, so there is no need to use the command line to install and set it up. Go to the Microsoft Store and search for `wsl`. From the list shown, select the application shown in the following figure:

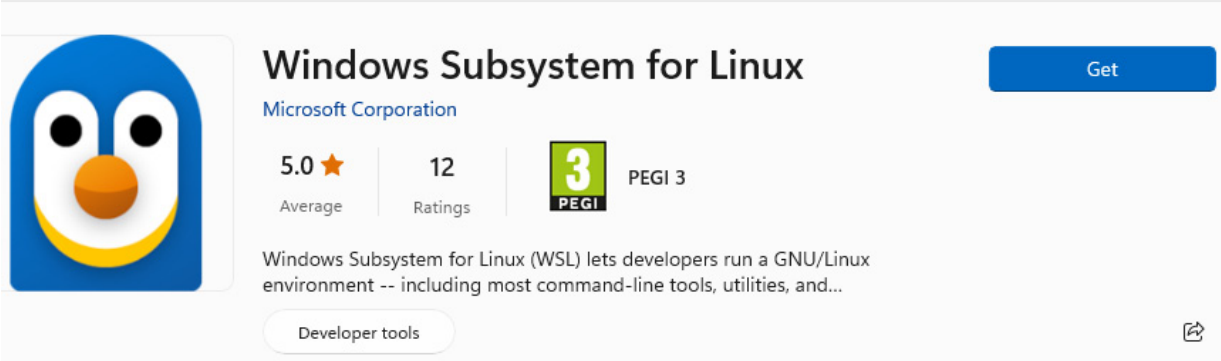

Figure 1.9 – WSL application from the Windows Store

After installing WSL, all you need to do is to install any Linux distributions available for it. If you attempt to open the freshly installed WSL application, you will get a terminal message saying that the WSL has no distribution installed. This means that you will need to install a distro by going back to Microsoft Store and searching for one. For example, if you search for `Linux` inside Microsoft Store, you will get results including SUSE Linux Enterprise Server, Oracle Linux, Kali Linux, Ubuntu LTS, Debian, and openSUSE Leap, among others.

IMPORTANT NOTE

*Make sure you have Hyper-V enabled in Windows, as it is the service responsible for running WSL. To enable it, go to **Windows Features**, and select **Hyper-V** from the list, then click on **OK**. After installing the necessary components, a restart is required. Hyper-V is available by default on Windows 11 Pro, Enterprise and Education, but NOT on Home edition.*

Now you can install a Linux distribution from Microsoft Store. We will try doing this with Ubuntu for our demonstration. After installation, you can open the application, create a user, and start using it inside the command line – it is that easy. To open the new Linux distribution, enter its name in the search bar, hit *Enter*, and a new terminal window with the Linux distribution is opened directly in Windows Terminal application, as shown in the following screenshot:

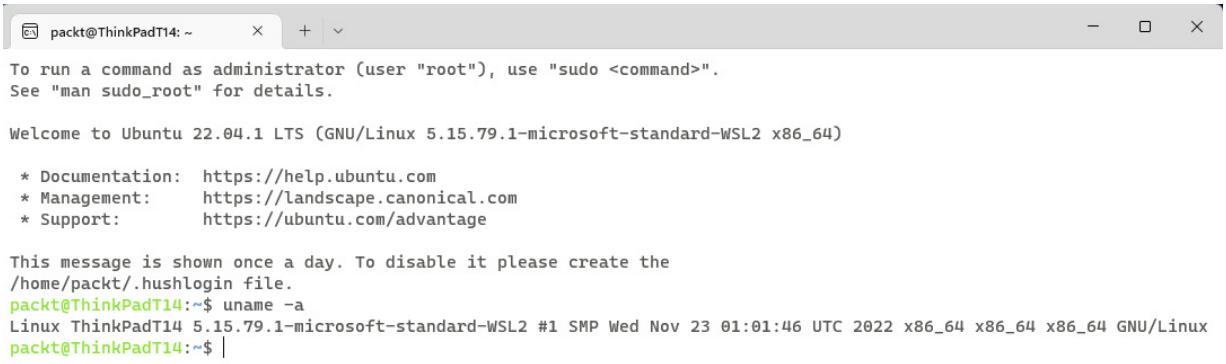

Figure 1.10 – Ubuntu inside a Windows Terminal window using WSL

Furthermore, you will have access to the distribution's filesystem directly from **File Explorer** inside Windows. The following screenshot shows the Ubuntu filesystem accessible from **File Explorer**:

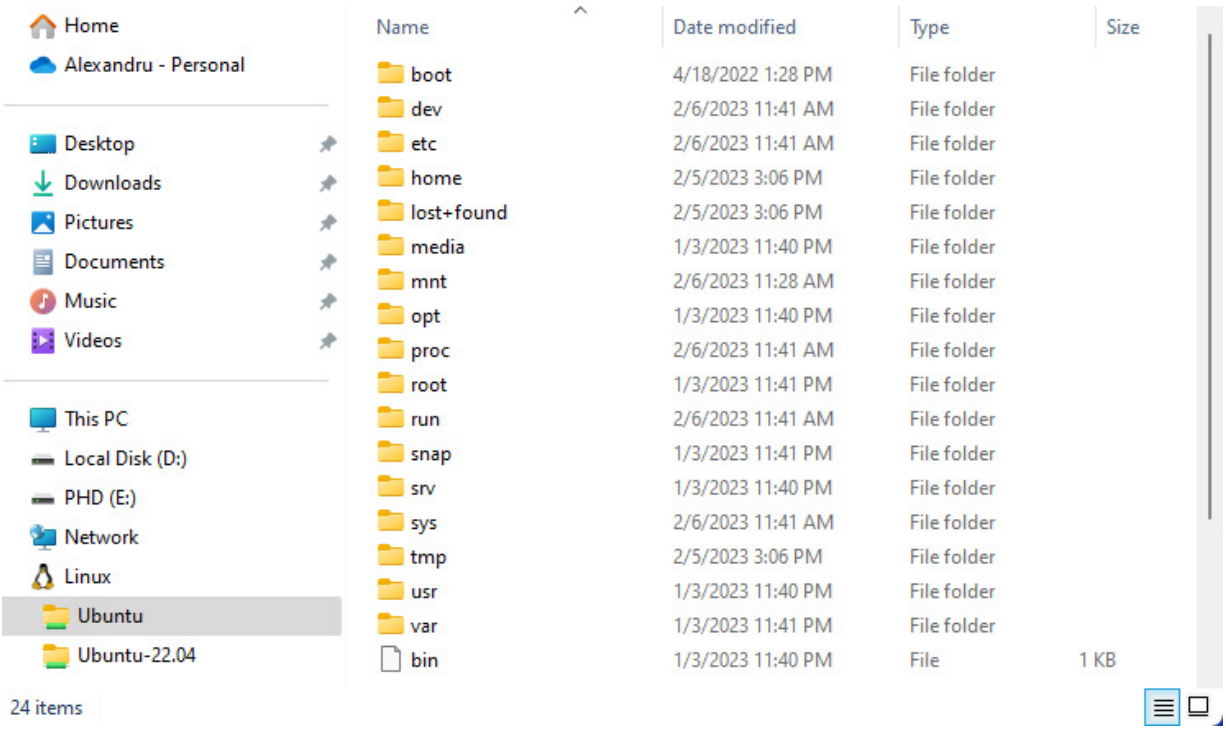

Figure 1.11 – The Ubuntu filesystem in the File Explorer in Windows 11

WSL enables a swift adoption of Linux for a growing number of Windows professionals. As shown in this section, WSL is relatively easy to configure, and with WSL, there's no need for a dedicated hypervisor to run a Linux instance.

By now, you have learned how to install Linux on a VM inside Windows using three different hypervisors, the VMware Workstation, Microsoft's own Hyper-V, and Oracle's VirtualBox. As we stated earlier, the installation process on macOS is very similar and there is no need for us to spend any more space covering it, as it would only duplicate the output. The interface of VMware Fusion on macOS is similar to the one used in Windows, with minor changes.

Installing on bare metal is similar; the only difference is that you require physical access to the destination machines. As stated at the beginning of this chapter, there is one more way to install Linux, and this is over the network. This is a more advanced task that requires more attention to detail along with basic networking knowledge. Also, understanding the Linux boot process is mandatory.

In the next section, we will provide some details about the network installation process.

Installing Linux – the advanced stages

In this section, we will cover the more advanced aspects of installing Linux. As we saw in the previous sections, installation on bare metal and VM requires direct access to the given machines. But what if we do not have access to the location? Or there are so many machines that need to be set up, that completing the task manually would be tedious at best, and infeasible at worst?

Installing Linux in an enterprise environment with tens or hundreds of machines in use can be done using an automated environment by booting through the network. As we stated earlier, a detailed overview of the network boot technique is out of the scope of this book; nevertheless, we will describe the process and show you the most important aspects of it, as none of the prominent books out there discuss this.

But first, to better understand how network booting works, let us take a short look at the Linux boot process.

The Linux boot process

How does Linux boot? We will give you a comprehensive view of the process without getting into too many details.

When you first start your Linux-powered computer or virtual machine, the **BIOS** (or boot firmware) starts loading and initiates a bootloader. The BIOS has a specific configuration and is loaded by the manufacturer onto a memory chip on the motherboard (in the case of physical computers, not VMs). The BIOS has information about the hardware and capabilities of controlling peripherals such as keyboards and monitors. It also has information about the operating system and the location of the bootloader. Some of this information is user controlled and can be changed according to the user's needs, such as the boot sequence for example, or password protection. The BIOS also has control over the **network interface controllers** (**NICs**) and all the external ports, including USB and display ports. But this is about all it can do, as it requires a bootloader to further initiate any operating system existing on the disk.

A newer version of the BIOS is the **Unified Extensible Firmware Interface** (**UEFI**). It has the same advantages as the older BIOS, but offers more interactive interfaces and better support for newer operating systems. The drawback, however, is the lack of software support from third-party vendors.

There is also **Secure Boot**, a feature introduced to offer an extra layer of security for the operating system and the software that runs. Some Linux distributions support it, but not all of them. Secure Boot uses a digital signature that proves the authenticity of the operating system. In order to support Secure Boot, the operating system developer must obtain a valid certificate for the software that can be verified on boot to prove that the system is valid and has not been tampered with.

Now that we know what the BIOS, UEFI and Secure Boot are, let us learn about the bootloader. Once **Power-On Self Test** (**POST**) is finished, the **bootloader** is accessed to load the operating system. POST is a series of tests that are conducted upon startup to ensure that the hardware is fully functional. What is a bootloader? It is the bridge between the hardware and the operating system. It is stored in the boot sector of the bootable storage. It can be either a partition or the very first block of the storage medium.

The bootloader used on Linux is the **Grand Unified Bootloader** (**GRUB**). It is responsible for loading the kernel of the operating system. The kernel is the central component of Linux, responsible for all the software components, drivers, services, and hardware integration. All of this forms what we call user space. It is the GRUB that has the capacity to support network booting.

The information provided in this section is sufficient to get a grasp of the Linux boot process. We will now detail the use of network boot to install Linux in the following section.

PXE network boot explained

Earlier in this chapter we mentioned the **PXE** (pronounced *pixie*) boot option. What exactly is PXE? It is a service that uses different networking protocols for booting over the network. It is based on different protocols and standards that were introduced forty years ago to define the much-needed network boot interoperability, also known as the **Network Bootstrap Program** (**NBP**).

The protocols that PXE is based on are **Trivial File Transfer Protocol** (**TFTP**), **Dynamic Host Configuration Protocol** (**DHCP**), and the UDP/IP stack using the **Hyper Text Transfer Protocol** (**HTTP**). These three are the base for PXE's application programming interface. Nowadays, most network cards available on the market already have the PXE firmware installed. This makes PXE the standard for network boot on many architectures. For more information on the latest PXE version 2.1, visit the following link: https://web.archive.org/web/20110524083740/http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf.

For PXE to work, we need to have a PXE server on the network. This machine will provide the necessary bootable files in response to client requests on the network. For this, at least a DHCP and a TFTP server need to be installed on the PXE server. In addition, a **Network File System** (**NFS**) server must also be installed, as this protocol is required for network file sharing and is used in modern Linux operating systems.

But before we go into further detail, let us discuss how network boot works. PXE relies on a client/server environment where different machines are equipped with PXE-enabled NICs. The network configuration of the PXE environment was developed so that it does not interfere with the existing network configuration. As DHCP and TFTP are needed, the PXE environment makes sure that it does not interfere with the existing DHCP configuration of the non-PXE router from the local network. This is a well-thought-out design for corporate environments.

In a basic scenario, after all clients are set up for PXE boot (an option available from the BIOS on almost every computer), the NICs send DHCP requests over the network in order to find the local PXE server. In order to be able to correctly respond to those requests, PXE uses a sort of proxy DHCP that sends IP and mask information of the TFTP server back to the PXE-enabled clients. This way, it does not interfere with the local network's DHCP server.

Setting up a PXE server is beyond the scope of this chapter, but useful information about what this is and how it works is relevant and can be found at https://ubuntu.com/server/docs/install/netboot-amd64 and https://www.redhat.com/sysadmin/pxe-boot-uefi. However, further details, such as how to practically set up a DHCP server, will be found in *Chapter 13*.

For a PXE server to work, there are some specific steps to take, depending on the installation root you follow. There are several options available, as you can use iPXE (an open source network boot

firmware), `cloud-init` (specific to Ubuntu), or kickstart (for Fedora-based systems). Nonetheless, setting up DHCP, TFTP, and NFS servers is required, with the DNS server being optional (details on setting up these servers are available in *Chapter 13*).

As you'll see these details later, we will not include them here. This is the introductory chapter, intended to make you comfortable with different ways to install Linux, and we will slowly build upon this foundation throughout the book to get you ready for the more advanced stuff as you go through the chapters.

In the next section, we will give you some scenarios of using certain Linux flavors depending on specific needs. We will present to you what we consider to be the appropriate distributions and applications to use in different case studies. Please keep in mind that installing applications and working with package managers will be discussed in more detail in *Chapter 3*.

Linux distributions – a practical guide

The following use cases are inspired by real-world problems, taken mostly from the authors' own experience in the system administration and software engineering field. Each of these scenarios presents the challenge of choosing the right Linux distribution for the job.

Case study – development workstation

This case study is based on the following scenario made from the perspective of a software developer:

> *I'm a backend/frontend developer, writing mostly in Java, Node.js, Python, and Golang, and using mostly IntelliJ and VS Code as my primary IDE. My development environment makes heavy use of Docker containers (both building and deploying) and I occasionally use VMs (with VirtualBox) to deploy and test my code locally. I need a robust and versatile development platform.*

Let's look at the functional and system requirements before deciding which Linux distribution is fit for the job:

- **Functional requirements**: The requirements suggest a relatively powerful day-to-day development platform, either a PC/desktop or a laptop computer. The developer relies on local resources to deploy and test the code (for instance, Docker containers and VMs), perhaps frequently in an offline (airplane mode) environment if on the go.
- **System requirements**: The system will primarily be using the Linux desktop environment and window manager, with frequent context switching between the **Integrated Development Environment** (**IDE**) and Terminal windows. The required software packages for the IDE, Docker, hypervisor (VirtualBox), and tools should be readily available from open source or commercial vendors, ideally always being up to date and requiring minimal installation and customization effort.

Choosing the Linux distribution

The choice of Linux distribution here would be the **Ubuntu Desktop Long Term Support** (**LTS**) platform. Ubuntu LTS is relatively stable, runs on virtually any hardware platform, and is mostly up to date with hardware drivers. Software packages for the required applications and tools are generally available and stable, with frequent updates. Ubuntu LTS is an enterprise-grade, cost-effective, and secure operating system suitable for organizations and home users alike.

Besides Ubuntu, **Fedora** and **openSUSE** are equally suitable for a developer's workstation. Choosing between them depends on whether you need a **Debian-** or **Red Hat**/**SUSE**-based ecosystem, and whether you need more up-to-date packages or not.

Case study – secure web server

This case study is based on the following scenario made from the perspective of a DevOps engineer:

> *I'm looking for a robust platform running a secure, relatively lightweight, and enterprise-grade web server. This web server handles HTTP/SSL requests, offloading SSL before routing requests to other backend web servers, websites, and API endpoints. No load-balancing features are needed.*

Let's look at the **functional requirements** in this case study. When it comes to open source, secure, and enterprise-grade web servers, the top choices are usually NGINX, Apache HTTP Server, Node.js, Apache Tomcat, and lighttpd. Without going into the details of selecting one web server over another, let's just assume we pick Apache HTTP Server. It has state-of-the-art SSL/TLS support, excellent performance, and is relatively easy to configure.

We can deploy this web server in VPS environments, in local (*on-premises*) data centers, or the public cloud. The deployment form factor is either a VM or a Docker container. We are looking for a relatively low-footprint, enterprise-grade Linux platform.

Choosing the Linux distribution

Our choice of Linux distribution is **Rocky Linux** or **AlmaLinux**. Most of the time, those two distributions are a perfect match for Apache HTTP Server. They are relatively lightweight, coming only with bare-bones server components and an operating system networking stack. Both Rocky and Alma are widely available as VPS deployment template from both private and public cloud vendors. Our Apache HTTP Server can run as a Docker container on top of Rocky Linux or AlmaLinux, as we may need to horizontally scale to multiple web server instances. More details on setting up a web server are provided in *Chapter 13*.

Use case – personal blog

This case study is based on the following scenario made from the perspective of a software engineer and blogger:

> *I want to create a software engineering blog. I'll be using the Ghost blogging platform, running on top of Node.js, with MySQL as the backend database. I'm looking for a hosted Virtual Private Server (VPS) solution by one of the major cloud providers. I'll be installing, maintaining, and managing the related platform myself. Which Linux distribution should I use?*

Let's discuss the **functional requirements** for this use case. We are looking for a self-managed publicly hosted **Virtual Private Server** (**VPS**) solution. The related hosting cost is a sensitive matter. Also, the maintenance of the required software packages should be relatively easy. We foresee frequent updates, including the Linux platform itself.

Choosing the Linux distribution

Our picks for the Linux distribution in this scenario would be either **Debian Stable** or **Ubuntu Server LTS**. As previously highlighted, Ubuntu is a robust, secure, and enterprise-class Linux distribution. **Debian** is equally stable and offers good options for applications. The platform maintenance and administration efforts are not demanding. The required software packages – Node.js, Ghost, and MySQL – are easily available and are well maintained. Ubuntu Server has a relatively small footprint. We can run our required software stack for blogging easily within the Ubuntu system requirements so the hosting costs would be reasonable.

Use case – media server

This case study is based on the following scenario made from the perspective of a home theater aficionado:

I have a moderately large collection of movies (personal DVD/Blu-ray backups), videos, photos, and other media, stored on Network Attached Storage (NAS). The NAS has its own media server incorporated, but the streaming performance is rather poor. I'm using Plex as a media player system, with Plex Media Server as the backend. What Linux platform should I use?

Based on this description, let's identify the **system requirements** for this use case. The critical system requirements of a media server are speed (for a high-quality and smooth streaming experience), security, and stability. The related software packages and streaming codecs are subject to frequent updates, so platform maintenance tasks and upgrades are quite frequent. The platform is hosted locally, on a PC desktop system, with plenty of memory and computing power in general. The media is being streamed from the NAS, over the in-house **Local Area Network** (**LAN**), where the content is available via an NFS share.

Choosing the Linux distribution

Both **Debian** and **Ubuntu** would be excellent choices for a good media server platform. Debian's *stable* release is regarded as rock solid and very reliable by the Linux community, although it's somewhat outdated. Both feature advanced networking and security, but what may come as a decisive factor in choosing between the two is that Plex Media Server has an ARM-compatible package for Debian. The media server package for Ubuntu is only available for Intel/AMD platforms. If we owned a small-factor ARM-processor-based appliance, Debian would be the right choice. Otherwise, **Ubuntu LTS** would meet our needs here just as well.

Now that you know about different use cases, it is time to pick your Linux distribution and start playing with it. In this chapter, we provided you with a plethora of information that will prove invaluable as you start your journey with Linux.

Summary

In this chapter, we learned about Linux distributions, with a practical emphasis on choosing the right platform for our needs and performing the related installation procedures.

Throughout the chapter, the main emphasis was on the Ubuntu distribution. In the spirit of a practical approach, we covered VM environments running Linux. We also took a short route through the Windows realm, where we touched upon WSL, a modern-day abstraction of Linux as a native Windows application.

With the skills learned in this chapter, we hope you'll have a better understanding of how to choose different flavors of Linux distros based on your needs. You've learned how to install and configure Linux on a variety of platforms. You will use some of these skills throughout the rest of the book, but

most importantly, you'll now be comfortable quickly deploying the Linux distribution of your choice and testing with it.

Starting with the next chapter, we'll take a closer look at the various Linux subsystems, components, services, and applications. *Chapter 2*, *The Linux Shell and Filesystem*, will familiarize you with the Linux filesystem internals and related tools.

Questions

Here are a few questions and thought experiments that you may ponder, some based on the skills you learned in this chapter, and others revealed in later parts of the book:

1. If we have a relatively large number of Linux VM instances or distros deployed and running at the same time, how could we make it easier to manage them?

Hint: Use **Vagrant**, a tool for building and managing VM environments.

2. Can we run multiple Linux instances in WSL?

Hint: We can.

Further reading

Here are a few Packt titles that can help you with the task of Linux installation:

- *Fundamentals of Linux*, by Oliver Pelz
- *Mastering Ubuntu Server – Fourth Edition*, by Jay LaCroix
- *Mastering Linux Administration – First Edition*, by Alexandru Calcatinge and Julian Balog

2

The Linux Shell and Filesystem

Understanding the **Linux filesystem**, **file management** fundamentals, and the basics of the **Linux shell** and **command-line interface** (**CLI**) is essential for a modern-day Linux professional.

In this chapter, you will learn how to use the Linux shell and some of the most common commands in Linux. You will learn about the structure of a basic Linux command and how the Linux filesystem is organized. We'll explore various commands for working with files and directories. Along the way, we'll introduce you to the most common command-line text editors. We hope that by the end of this chapter, you'll be comfortable using the Linux CLI and be ready for future, more advanced explorations. This chapter will set the foundation for using the Linux shell, and for more information about the shell, go to *Chapter 8*, *Linux Shell Scripting*.

We're going to cover the following main topics:

- Introducing the Linux shell
- The Linux filesystem
- Working with files and directories
- Using text editors to create and edit files

Technical requirements

This chapter requires a working installation of a standard Linux distribution, on either server, desktop, PC, or **Virtual Machine** (**VM**). Our examples and case studies use the Ubuntu and Fedora platforms, but the commands and examples explored are equally suitable for any other Linux distribution.

Introducing the Linux shell

Linux has its roots in the Unix operating system, and one of its main strengths is the command-line interface. In the old days, this was called *the shell*. In **UNIX**, the shell is invoked with the `sh` command. The shell is a program that has two streams: an *input stream* and an *output stream*. The input is a command given by the user, and the output is the result of that command, or an interpretation of it. In other words, the shell is the primary interface between the user and the machine.

The main shell in major Linux distributions is called **Bash**, which is an acronym for **Bourne Again Shell**, named after Steve Bourne, the original creator of the shell in UNIX. Alongside Bash, there are other shells available in Linux, such as **ksh**, **tcsh**, and **zsh**. In this chapter and throughout the book, we will cover the Bash shell, as it is the most widely used shell in modern Linux distributions.

IMPORTANT NOTE

Distributions such as Debian, Ubuntu, Fedora, CentOS Stream, RHEL, openSUSE, SLE, and Linux Mint, just to name a few, use the Bash shell by default. Other distributions, such as Kali Linux, have switched to zsh by default. Manjaro offers zsh on some editions. For those who use macOS, you should know that zsh has been the default shell for some years now. Nevertheless, you can install any shell you want on Linux and make it your default one. In general, shells are pretty similar, as they do the same thing, but they add different extras to usability and features. If you are interested in a specific shell, feel free to use it and test out the differences between others.

One shell can be assigned to each user. Users on the same system can use different shells. One way to check the default shell is by accessing the **`/etc/passwd`** file. More details about this file and user accounts will be discussed in *Chapter 4*, *Managing Users and Groups*. For now, it is important to know where to look for the default shell. In this file, the last characters from each line represent the user's default shell. The **`/etc/passwd`** file has the users listed on each line, with details about their **process identification number** (**PID**), **group identification number** (**GID**), username, home directory, and basic shell.

To see the default shell for each user, execute the following command by using your user's name (in our case, it is **`packt`**):

```
cat /etc/passwd | grep packt
```

The output should be a list of the contents of the **`/etc/passwd`** file. Depending on the number of users you have on your system, you will see all of them, each one on a separate line. An easier way to see the *current shell* is by running the following command:

```
echo $0
```

This shows what exactly is running your command, which in the case of the CLI is the shell. The **`$0`** part is a **bash special parameter** that refers to the currently running process. In the following screenshot you will see the output of the previous two commands we used to discover the shell and a comparison between the output of the **`echo $0`** command on Ubuntu and on Debian:

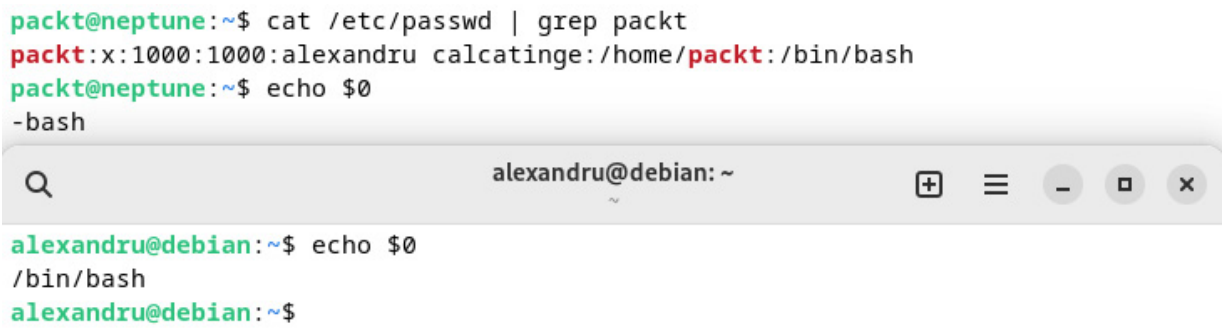

Figure 2.1 – Commands used to see the running shell

As you can see, running the `echo $0` command gives us different outputs but with the same message: the running shell is Bash. If you have other shells that you prefer, you can easily assign another shell to your user, if you already have it installed. However, if you know Bash, you will be comfortable with all the other available shells.

IMPORTANT NOTE

The Linux shell is case-sensitive. This means that everything you type inside the command line should respect this. For example, the `cat` command used earlier used lowercase. If you type `Cat` or `CAT`, the shell will not recognize it as being a command. The same rule applies to file paths. You will notice that default directories in your home directory use uppercase for the first letter, as in `~/Documents`, `~/Downloads`, and so on. Those names are different from `~/documents` or `~/downloads`.

In this chapter, you will learn how to use Linux commands and the shell in addition to learning about its filesystem. You will learn about software management in *Chapter 3*, thus showing you how to install another shell now would mean that we will get ahead of ourselves. We want you to slowly but steadily build your Linux knowledge, so we will show you in the next chapter how to install a new shell. For now, Bash is sufficient, and we will use it throughout the book.

If you want to see all the shells that are installed on your system, you can run the following command:

```
cat /etc/shells
```

In our case, the output with all the installed shells (by default) in Ubuntu Server 22.04.2 LTS is shown in the following image:

```
packt@neptune:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/usr/bin/sh
/bin/dash
/usr/bin/dash
/usr/bin/tmux
/usr/bin/screen
packt@neptune:~$
```

Figure 2.2 – The shells available by default in Ubuntu

This will show you all the shells installed. You can use any of those or can install new ones as we will show you in *Chapter 3*. Also, in *Chapter 4*, when we work with user accounts, you will get to learn how you can change a user's shell.

The following section will introduce you to shell connection types.

Establishing the shell connection

We can make two different types of connections to the shell: `tty` and `pts`. The name `tty` stands for **teletypewriter**, which was a type of terminal used at the beginning of computing. This connection is considered a native one, with ports that are direct connections to your computer. The link between the user and the computer is mainly found to be through a keyboard, which is considered to be a native terminal device.

The `pts` connection is generated by SSH or Telnet types of links. Its name stands for **pseudo terminal slave**, and it is an emulated connection made by a program, in most cases `ssh` or `xterm`. It is the slave of the **pseudo-terminal device**, which is represented as `pty`.

In the next section, we will further explore the connections to virtual terminals available in Linux.

Virtual consoles/terminals

The terminal was thought of as a device that manages the input strings (which are commands) between a process and other I/O devices such as a keyboard and a screen. There are also **pseudo terminals**, which are emulated terminals that behave the same way as a **classical terminal**. The difference is that it does not interact with devices directly, as it is all emulated by the Linux kernel, which transmits the I/O to a program called the shell.

Virtual consoles are accessible and run in the background, even though there is no open terminal. To access those virtual consoles, you can use the commands *Ctrl* + *Alt* + *F1*, *Ctrl* + *Alt* + *F2*, *Ctrl* + *Alt* + *F3* , *Ctrl* + *Alt* + *F4* , *Ctrl* + *Alt* + *F5*, and *Ctrl* + *Alt* + *F6*. These will open **`tty1`**, **`tty2`**, **`tty3`**, **`tty4`**, **`tty5`**, and **`tty6`**, respectively, on your computer.

We will explain this using an Ubuntu 22.04.2 LTS Server VM installation, but it is identical in Rocky Linux too. After starting the VM and being prompted to log in with your username and password, the first line on the screen will be something similar to the following output:

```
Ubuntu 22.04.2 LTS neptune tty1
```

If you press any of the preceding key combinations, you will see your terminal change from **`tty1`** to any of the other **`tty`** instances. For example, if you press *Ctrl* + *Alt* + *F6*, you will see this:

```
Ubuntu 22.04.2 LTS neptune tty6
```

As we were using the server edition of Ubuntu, we did not have the GUI installed. But if you were to use a desktop edition, you will be able to use *Ctrl* + *Alt* + *F7* to enter **`x graphical`** mode, for example. The **`neptune`** string is the name we gave to our virtual machine.

If you are not able to use the preceding keyboard combinations, there is a dedicated command for changing virtual terminals. The command is called **`chvt`** and has the syntax **`chvt N`**. Even though we have not discussed shell commands yet, we will show you an example of how to use them and other related commands. This action can only be performed by an administrator account or by using **`sudo`**. Briefly, **`sudo`** stands for *superuser do* and allows any user to run programs with administrative privileges or with the privileges of another user (more details about this in *Chapter 4*, *Managing Users and Groups*).

In the following example, we will use Ubuntu to show you how to change virtual terminals. First, we will see which virtual terminal we are currently using to change it to another one without using the *Ctrl* + *Alt* + *Fn* keys.

The **`who`** command will show you information about the users currently logged in to the computer. In our case, as we are connected through SSH to our virtual machine, it will show that the user **`packt`** is currently using pseudo-terminal zero (**`pts/0`**):

```
packt    pts/0        2023-02-28 10:45 (192.168.122.1)
```

If we were to run the same command in the console of the virtual machine directly, we would have the following output:

```
packt    pts/0        2023-02-28 10:45 (192.168.122.1)
packt    tty1         2023-02-28 10:50
```

It shows that the user is connected to both the virtual terminal 1 (`tty1`) and also through SSH from our host operating system to the virtual machine (`pts/0`).

Now, by using the `chvt` command, we will show you how to change to the sixth virtual terminal. After running `sudo chvt 6`, you will be prompted to provide your password and immediately be switched to virtual terminal number six. Running `who` once more will show you all logged-in users and the virtual terminals they use. In our case will be `pts/0`, `tty2`, and `tty6`. Please take into consideration that your output could be different, as in different virtual terminal numbers.

Now that we know what types of shell connections are established, let us learn about the shell's prompt in the next section.

The command-line prompt

The **command-line prompt** or **shell prompt** is the place where you type in the commands. Usually, the command prompt will show the username, hostname, present working directory, and a symbol that indicates the type of user running the shell.

Here is an example from the Ubuntu 22.04.2 LTS Server edition (similar to Debian):

```
packt@saturn:~$
```

Here is an example from the Fedora 37 server (similar to Rocky Linux, RHEL, or AlmaLinux):

```
[packt@localhost ~]$
```

Here is a short explanation of the prompt:

- **`packt`** is the name of the user currently logged in
- **`saturn`** and **`localhost`** are the hostnames
- **`~`** represents the home directory (it is called a tilde)
- **`$`** shows that the user is a regular user (when you are logged in as an administrator, the sign changes into a hashtag, **`#`**)

Also, when using openSUSE, you will notice that the prompt is different than the ones in Ubuntu/Debian and Fedora/RHEL. The following is an example of the prompt while running the Leap 15.4 server edition:

```
packt@localhost:~>
```

As you can see, there is no dollar sign (`$`) or hashtag (`#`), only a greater than sign (`>`). This might be confusing at first, but when you will use the root user, the sign will eventually change to the hashtag (`#`). The following is an example:

```
localhost:/home/packt #
```

Let’s look at the shell command types next.

Shell command types

Shells work with **commands**, and there are two types that they use: internal ones and external ones. **Internal commands** are built inside the shell. **External commands** are installed separately. If you want to check the type of command you are using, there is the `type` command. For example, you can check what type of command `cd` (change directory) is:

```
packt@neptune:~$ type cd
cd is a shell builtin
```

The output shows that the `cd` command is an internal one, built inside the shell. If you are curious, you could find out the types of other commands that we will show you in the following sections by writing type in front of the command’s name. Let us see some more examples in the following image:

```
packt@neptune:~$ type date
date is /usr/bin/date
packt@neptune:~$ type ls
ls is aliased to `ls --color=auto'
packt@neptune:~$ type man
man is /usr/bin/man
packt@neptune:~$ type grep
grep is aliased to `grep --color=auto'
packt@neptune:~$ type echo
echo is a shell builtin
packt@neptune:~$ type touch
touch is /usr/bin/touch
packt@neptune:~$ type pwd
pwd is a shell builtin
packt@neptune:~$ type elif
elif is a shell keyword
packt@neptune:~$
```

Figure 2.3 – Different types of commands in Linux

Now that you know some of the types of Linux commands, let us dissect the command’s structure and learn about its components.

Explaining the command structure

We have already used some commands, but we did not explain the structure of a Linux command. We will do that now for you to be able to understand how to use commands. In a nutshell, Unix and

Linux commands have the following form:

- The command's name
- The command's options
- The command's arguments

Inside the shell, you will have a general structure such as the following:

```
command [-option(s)] [argument(s)]
```

A suitable example would be the use of the `ls` command (`ls` comes from a *list*). This command is one of the most-used commands in Linux. It lists files and directories and can be used with both options and arguments.

We can use `ls` in its simplest form, without options or arguments. It lists the contents of your present working directory (`pwd`). In our case, it is the home directory, indicated by the `~` tilde character in the shell's prompt (see *Figure 2.10*).

The `ls` command with the `-l` option (lowercase L) uses a long listing format, giving you extra information about files and directories from your present working directory (`pwd`):

```
packt@saturn:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  snap  Templates  Videos
packt@saturn:~$ ls -l
total 36
drwxr-xr-x 2 packt packt 4096 feb 28 12:32 Desktop
drwxr-xr-x 2 packt packt 4096 mar  1 21:05 Documents
drwxr-xr-x 2 packt packt 4096 feb 28 12:32 Downloads
drwxr-xr-x 2 packt packt 4096 feb 28 12:32 Music
drwxr-xr-x 2 packt packt 4096 feb 28 12:32 Pictures
drwxr-xr-x 2 packt packt 4096 feb 28 12:32 Public
drwx------ 3 packt packt 4096 feb 28 12:32 snap
drwxr-xr-x 2 packt packt 4096 feb 28 12:32 Templates
drwxr-xr-x 2 packt packt 4096 feb 28 12:32 Videos
packt@saturn:~$ ls -l ~/Documents
total 0
-rw-rw-r-- 1 packt packt 0 mar  1 21:04 file1
-rw-rw-r-- 1 packt packt 0 mar  1 21:04 file2
-rw-rw-r-- 1 packt packt 0 mar  1 21:04 file3
-rw-rw-r-- 1 packt packt 0 mar  1 21:05 report1
-rw-rw-r-- 1 packt packt 0 mar  1 21:05 report2
-rw-rw-r-- 1 packt packt 0 mar  1 21:05 report3
-rw-rw-r-- 1 packt packt 0 mar  1 21:05 report4
-rw-rw-r-- 1 packt packt 0 mar  1 21:05 report5
```

Figure 2.4 – Using the ls command with both options and attributes

In the preceding example, we used **`ls -l ~/Documents/`** to show the contents of the **`~/Documents`** directory. Shown here is a way to use the command with both options and attributes, without changing our present working directory to **`~/Documents`**.

In the following section, we will show you how to use the manual pages available by default in Linux.

Consulting the manual

Any Linux system administrator's best friend is the manual. Each command in Linux has a manual page that gives the user detailed information about its use, options, and attributes. If you know the command you want to learn more about, simply use the **`man`** command to explore. For the **`ls`** command, for example, you use **`man ls`**.

The manual organizes its command information into different sections, with each section being named by convention to be the same on all distributions. Briefly, those sections are **`name`**, **`synopsis`**, **`configuration`**, **`description`**, **`options`**, **`exit status`**, **`return value`**, **`errors`**, **`environment`**, **`files`**, **`versions`**, **`conforming to`**, **`notes`**, **`bugs`**, **`example`**, **`authors`**, **`copyright`**, and **`see also`**.

Similar to the manual pages, almost all commands in Linux have a **`-help`** option. You can use this for quick reference.

For more information about the **`help`** and **`man`** pages, you can check each command's help or manual page. Try the following commands:

```
$ man man
$ help help
```

When you use the manual, keep in mind that it is not a step-by-step how-to guide. It is technical documentation that might be confusing at first. Our advice is to use the **`man`** pages as much as you can. Before you search for anything on the internet, try to read the manual first. This will be a good exercise, and you will become proficient with Linux commands in no time.

Consider the manual your friend, similar to the textbooks you used in high school or college. It will give you first-hand information when you most need it. If you take into consideration situations where outside internet access is limited, with no access to search engines, the built-in manual will be your best companion. Learn to use its powers to your advantage.

In the following section, you will learn about the Linux filesystem.

The Linux filesystem

The **Linux filesystem** consists of a logical collection of files that are stored on a partition or a disk. Your hard drive can have one or many partitions. Those partitions usually contain only one filesystem, and they can extend to an entire disk. One filesystem could be the `/ (root)` filesystem and another the `/home` filesystem. Or, there can be just one that contains all filesystems.

Generally, using one filesystem per partition is considered to be good practice by allowing for logical maintenance and management. As everything in Linux is a file, physical devices such as hard drives, DVD drives, USB devices, and floppy drives are treated as files too. In this section, you will learn about the directory structure, how to work with files, and some very useful editing techniques from the command line.

Directory structure

Linux uses a **hierarchical** filesystem structure. It is similar to an upside-down tree, with the root (`/`) at the base of the filesystem. From that point, all the branches (directories) spread throughout the filesystem.

The **Filesystem Hierarchy Standard** (**FHS**) defines the structure of Unix-like filesystems. However, Linux filesystems also contain some directories that aren't yet defined by the standard.

Exploring the Linux filesystem from the command line

Feel free to explore the filesystem yourself by using the `tree` command. In Fedora Linux, it is already installed, but if you use Ubuntu, you will have to install it by using the following command:

```
$ sudo apt install tree
```

Do not be afraid to explore the filesystem, because no harm will be done just by looking around. You can use the `ls` command to list the contents of directories, but `tree` offers different graphics. The following image shows you the differences between the outputs:

```
packt@neptune:~$ ls -la
total 36
drwxr-x--- 4 packt packt 4096 Feb 28 10:37 .
drwxr-xr-x 3 root  root  4096 Feb 27 08:58 ..
-rw------- 1 packt packt 1039 Mar  1 18:56 .bash_history
-rw-r--r-- 1 packt packt  220 Jan  6  2022 .bash_logout
-rw-r--r-- 1 packt packt 3771 Jan  6  2022 .bashrc
drwx------ 2 packt packt 4096 Feb 27 08:59 .cache
-rw------- 1 packt packt    0 Feb 27 10:52 .lesshst
-rw-r--r-- 1 packt packt  807 Jan  6  2022 .profile
drwx------ 2 packt packt 4096 Feb 27 08:58 .ssh
-rw-r--r-- 1 packt packt    0 Feb 27 08:59 .sudo_as_admin_successful
-rw-rw-r-- 1 packt packt   67 Feb 27 14:33 users
packt@neptune:~$ tree -a
.
├── .bash_history
├── .bash_logout
├── .bashrc
├── .cache
│   └── motd.legal-displayed
├── .lesshst
├── .profile
├── .ssh
│   └── authorized_keys
├── .sudo_as_admin_successful
└── users

2 directories, 9 files
packt@neptune:~$
```

Figure 2.5 – Comparing the output of ls -la commands and tree -a commands

The **`tree`** command has different options available, and you can learn about them by using the manual. Let us use the **`tree`** command by invoking the **`-L`** option, which tells the command how many levels down to go, and the last attribute states which directory to start with. In our example, the command will go down one level, starting from the **`root`** directory, represented by the forward slash as an argument (see *Figure 2.12*):

```
$ tree -L 1 /
```

Start exploring the directories from the structure by using the **`tree`** command, as shown here:

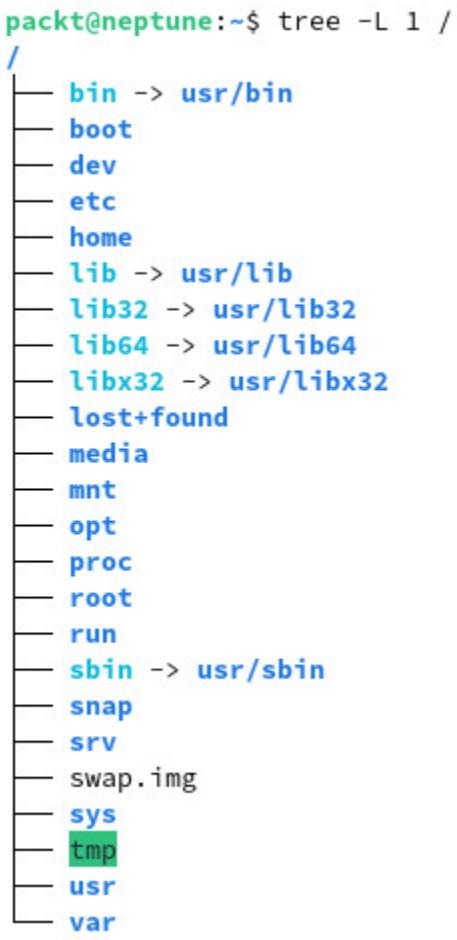

Figure 2.6 – The tree command with option and argument on Ubuntu

IMPORTANT NOTE

Remember that some of the directories you are about to open will contain a large number of files and/or other directories, which will clutter your terminal window.

The following are the directories that exist on almost all versions of Linux. Here's a quick overview of the Linux root filesystem:

- **`/`**: Root directory. The root for all other directories.
- **`/bin`**: Essential command binaries. The place where binary programs are stored.
- **`/boot`**: Static files of the boot loader. The place where the kernel, bootloader, and **`initramfs`** are stored.
- **`/dev`**: Device files. Nodes to the device equipment, a kernel device list.
- **`/etc`**: Host-specific system configuration. Essential config files for the system, boot time loading scripts, **`crontab`**, **`fstab`** device storage tables, **`passwd`** user accounts file.
- **`/home`**: user Home directory. The place where the user's files are stored.
- **`/lib`**: Essential shared libraries and kernel modules. Shared libraries are similar to **Dynamic Link Library** (**DLL**) files in Windows.

- **`/media`**: Mount point for removable media. For external devices and USB external media.
- **`/mnt`**: Mount point for mounting a filesystem temporarily. Used for legacy systems.
- **`/opt`**: Add-on application software packages. The place where *optional* software is installed.
- **`/proc`**: Virtual filesystem managed by the kernel. a special directory structure that contains files essential for the system.
- **`/sbin`**: Essential system binaries. Vital programs for the system's operation.
- **`/srv`**: Data for services provided by this system.
- **`/tmp`**: Temporary files.
- **`/usr`**: Secondary hierarchy. The largest directory in Linux that contains support files for regular system users:
    - **`/usr/bin`** – system-executable files
    - **`/usr/lib`** – shared libraries from **`/usr/bin`**
    - **`/usr/local`** – source compiled programs not included in the distribution
    - **`/usr/sbin`** – specific system administration programs
    - **`/usr/share`** – data shared by the programs in **`/usr/bin`** such as config files, icons, wallpapers or sound files
    - **`/usr/share/doc`** – documentation for the system-wide files
- **`/var`**: Variable data. Only data that is modifiable by the user is stored here, such as databases, printing spool files, user mail, and others; **`/var/log`** – contains log files that register system activity

Next, we're going to learn how to work with these files and directories.

Working with files and directories

Remember that everything in Linux is a file. A directory is a file too. As such, it is essential to know how to work with them. Working with files in Linux implies the use of several commands for basic file and directory operations, file viewing, file creation, file location, file properties, and linking. Some of the commands, which will not be covered here, have uses closely related to files. These will be covered in the following section.

Understanding file paths

Each file in the FHS has a *path*. The path is the file's location represented in an easily readable representation. In Linux, all the files are stored in the root directory by using the FHS as a standard to organize them. Relations between files and directories inside this system are expressed through the forward-slash character (`/`). Throughout computing history, this was used as a symbol that described addresses. Paths are, in fact, addresses for files.

There are two types of paths in Linux, relative ones and absolute ones. An **absolute path** always starts with the root directory and follows the branches of the system up to the desired file. A **relative path** always refers to the **present working directory** (**pwd**) and represents the relative path to it. Thus, a relative path is always a path that is relative to your present working directory.

For example, let us refer to an existing file from our home directory, a file called **`poem`**. Being inside the home directory, and our **`pwd`** command being the **`home`** directory for **`packt`**, the absolute path of the file called **`poem`** would be as follows:

```
/home/packt/poem
```

If we were to show the contents of that file using the **`cat`** command, for example, we would use the command with the absolute path:

```
cat /home/packt/poem
```

The relative path to the same file would be relative to **`pwd`**, so in our case, where we're already inside the home directory, using the **`cat`** command would be like this:

```
cat poem
```

The absolute path is useful to know about when you work with files. After some practice, you will come to learn the paths to the most-used files. For example, one file that you will need to learn the path for is the **`passwd`** file. It resides in the **`/etc`** directory. Thus, when you will refer to it, you will use its absolute path, **`/etc/passwd`**. Using a relative path to that file would imply that you are either inside its parent directory or somewhere close in the FHS.

Working with relative paths involves knowing two special characters used to work with the FHS. One special character is the dot (`.`), and it refers to the current directory. The other is two consecutive dots (`..`) and refers to the parent directory of the current directory. When working with relative paths, make sure that you always check what directory you are in. Use the **`pwd`** command to show your present working directory.

A good example of working with relative paths is when you are already inside the parent directory and need to refer to it. If you need to see the accounts list from your system, which is stored inside the **`passwd`** file, you can refer to it by using a relative path. For this exercise, we are inside our home directory:

```
packt@neptune:~$ pwd
/home/packt
packt@neptune:~$ cat passwd
cat: passwd: No such file or directory
packt@neptune:~$ cat ../../etc/passwd
```

Figure 2.7 – Using the relative path of a file

Here, we first check our present working directory with the `pwd` command, and the output will be our home directory's path, `/home/packt`. Secondly, we try to show the contents of the `passwd` file using the `cat` command right from the home directory, but the output will be an error message saying that there is no such file or directory inside our home directory. We used the relative path, which is always relative to our present working directory, hence the error. Thirdly, we use the double-consecutive dots special characters to refer to the file with its relative path. In this case, the command is `cat ../../etc/passwd`.

TIP

Always use the Tab key on your keyboard for autocompletion and to check whether the path you typed is correct or not. In the preceding example, we typed `../../etc` and pressed Tab, which autocompleted with a forward slash. Then, we typed the first two letters of the file we were looking for and pressed Tab again. This showed us a list of files inside the `/etc` directory that started with `pa`. Seeing that `passwd` was in there, we knew that the path was right, so we typed two more `s` characters and pressed Tab again. This completed the command for us and we pressed Enter/Return to execute the command.

The path in the final command is relative to our home directory and it translates as follows: *concatenate the file with the `passwd` name that is located in the `/etc` directory in the parent directory (first two dots) of the parent directory (second two dots) of our current directory (home)*. Therefore, the `/etc/passwd` absolute path is translated into a relative path to our home directory like this: `../../etc/passwd`.

Next, we are going to learn about basic file operations in Linux.

Basic file operations

Daily, as a system administrator, you will manipulate files. This includes creating, copying, moving, listing, deleting, linking, and so on. The basic commands for these operations have already been discussed throughout this chapter, but now it is time to get into more detail about their use, options, and attributes. Some more advanced commands will be detailed in the following sections.

Creating files

There are situations when you will need to **create** new files. You have one option to create a new empty file with the `touch` command. When you use it, it will create a new file with you as the file owner and with a size of zero, because it is an empty file.

In the following example, we create a new file called `new-report` inside the `~/packt/` directory:

```
packt@neptune:~$ pwd
/home/packt
packt@neptune:~$ touch new-report
packt@neptune:~$ ls -l new-report
-rw-rw-r-- 1 packt packt 0 Mar  2 14:24 new-report
packt@neptune:~$ touch new-report
packt@neptune:~$ ls -l new-report
-rw-rw-r-- 1 packt packt 0 Mar  2 14:25 new-report
packt@neptune:~$
```

Figure 2.8 – Using the touch command to create and alter files

The `touch` command is also used to change the modification time of a file without changing the file itself. Notice the difference between the initial time when we first created the `new-report` file and the new time after using the `touch` command on it. You can also change the access time by using the `-a` option of the `touch` command. By default, the long listing of the `ls` command shows only the modification/creation time. If you want to see the access time, there is the `atime` parameter you can use with the `- time` option. See the example used in the following figure:

```
packt@neptune:~$ ls -l new-report
-rw-rw-r-- 1 packt packt 67 Mar  2 14:30 new-report
packt@neptune:~$ touch -a new-report
packt@neptune:~$ ls -l --time=atime new-report
-rw-rw-r-- 1 packt packt 67 Mar  2 14:31 new-report
packt@neptune:~$
```

Figure 2.9 – Using touch to alter the access time

The modification, creation, and access time stamps are very useful, especially when using commands such as `find`. They give you a more *granular* search pattern. We will get back to this command with more examples in future sections.

Files can also be created by using redirection and the `echo` command. `echo` is a command that prints the string given as a parameter to the standard output (the screen). The output of the `echo` command can be written directly to a file by using the output redirection:

```
packt@neptune:~$ echo this is a presentation
this is a presentation
packt@neptune:~$ echo this is also a presentation > art-file
packt@neptune:~$ cat art-file
this is also a presentation
packt@neptune:~$ echo this is a new presentation >> art-file
packt@neptune:~$ cat art-file
this is also a presentation
this is a new presentation
packt@neptune:~$
```

Figure 2.10 – Using echo with output redirection

In the preceding example, we redirected the text from the **`echo`** command to the presentation file. It did not exist at the beginning, so it was automatically created by the command. The first **`echo`** command added a line to the file by using the **`>`** operator. The second **`echo`** command appended a new line of text to the end of the file, by using the **`>>`** operator.

Listing files

We have already used some examples with the **`ls`** command before, so you are somewhat familiar with it. We covered the **`-l`** option as an example of the command's structure. Thus, we will not cover it any further here. We will explore new options for this essential and useful command:

- **`ls -lh`**: The **`-l`** option lists the files in an extended format, while the **`-h`** option shows the size of the file in a human-readable format, with the size in kilobytes or megabytes rather than bytes.
- **`ls -la`**: The **`-a`** option shows all the files, including hidden ones. Combined with the **`-l`** option, the output will be a list of all the files and their details.
- **`ls -ltr`**: The **`-t`** option sorts files by their modification time, showing the newest first. The **`-r`** option reverses the order of the sort.
- **`ls – lS`**: The **`-S`** option sorts the files by their size, with the largest file first.
- **`ls -R`**: The **`-R`** option shows the contents of the current or specified directory in recursive mode.

A method used frequently for listing files and directories is called **long listing**, and it uses the **`ls -la`** command. Let's look at it in detail here, even though we will discuss this thoroughly in *Chapter 4*, *Managing Users and Groups*.

One example of a long listing used on our home directory can be seen in *Figure 2.11*, when we compared the output of **`ls -la`** with the output of the **`tree`** command. The following code snippet is a short example:

```
total 48
drwxr-x--- 5 packt packt 4096 Mar  2 14:44 .
drwxr-xr-x 3 root  root  4096 Feb 27 08:58 ..
-rw-rw-r-- 1 packt packt   55 Mar  2 14:46 art-file
-rw------- 1 packt packt 1039 Mar  1 18:56 .bash_history
```

In the output, the first row after the command shows the number of blocks inside the directory listed. After that, each line represents one file or subdirectory, with the following detailed information:

- The first character is the type of the file: **`d`** for the directory, **`:`** for the file, **`l`** for a link, **`c`** for the character device, and **`b`** for the block device
- The following nine characters represent the permissions (detailed in *Chapter 4*, *Managing Users and Groups*)
- The hard link for that file (see the *Working with links* subsection in this chapter)
- The owner's PID and GID (details in *Chapter 4*, *Managing Users and Groups*)
- The size of the file (the number depends on whether it is in human-readable format or not)
- The last modification time of the file
- The name of the file or directory

The first two lines are a reference to itself (the dot) and to the parent directory (the two dots from the second line).

The next section will teach you how to copy and move files.

Copying and moving files

To copy files in Linux, the **`cp`** command is used. The **`mv`** command moves files around the filesystem. This command is also used to rename files.

To copy a file, you can use the **`cp`** command in the simplest way:

```
cp source_file_path destination_file_path
```

Here, **`source_file_path`** is the name of the file to be copied, and **`destination_file_path`** is the name of the destination file. You can also copy multiple files inside a directory that already exists. If the destination directory does not exist, the shell will signal to you that the target is not a directory.

Now let's look at some variations of these commands:

- **`cp -a`**: The **`-a`** option copies an entire directory hierarchy in recursive mode by preserving all the attributes and links. In the following example, we copied the entire **`dir1`** directory that we previously created inside our home directory, to a newly created directory called **`backup_dir1`** by using the **`-a`** option:

```
packt@neptune:~$ ls
art-file  dir1  new-report  users
packt@neptune:~$ mkdir backup_dir1
packt@neptune:~$ ls
art-file  backup_dir1  dir1  new-report  users
packt@neptune:~$ cp -a dir1/ backup_dir1/
packt@neptune:~$ ls backup_dir1/
dir1
packt@neptune:~$
```

Figure 2.11 – Using the copy command with the -a option

- **`cp -r`**: This option is similar to **`-a`**, but it does not preserve attributes, only symbolic links.
- **`cp -p`**: The **`-p`** option retains the file's permissions and timestamps. Otherwise, just by using **`cp`** in its simplest form, copies of the files will be owned by your user with a timestamp of the time you did the copy operation.
- **`cp -R`**: The **`-R`** option allows you to copy a directory recursively. In the following example, we will use the **`ls`** command to show you the contents of the **`~/packt/`** directory, and then the **`cp -R`** command to copy the contents of the **`/files`** directory to the **`/new-files`** one. The **`/new-files`** directory did not exist. The **`cp -R`** command created it:

```
packt@neptune:~$ ls
art-file  backup_dir1  dir1  new-report  users
packt@neptune:~$ mkdir files
packt@neptune:~$ cd files/
packt@neptune:~/files$ touch files{1..6}
packt@neptune:~/files$ cd ..
packt@neptune:~$ ls
art-file  backup_dir1  dir1  files  new-report  users
packt@neptune:~$ cp -R files/ new-files
packt@neptune:~$ ls
art-file  backup_dir1  dir1  files  new-files  new-report  users
packt@neptune:~$ cd new-files/
packt@neptune:~/new-files$ ls
files1  files2  files3  files4  files5  files6
packt@neptune:~/new-files$
```

Figure 2.12 – Using the cp -R command

Moving files around is done with the **`mv`** command. It is either used to move files and directories from one destination to another or to rename a file. The following is an example in which we rename **`files1`** into **`old-files1`** using the **`mv`** command: **`mv files1 old-files1`**.

There are many other options that you could learn about just by visiting the manual pages. Feel free to explore them and use them in your daily tasks.

Working with links

Links are a compelling option in Linux. They can be used as a means of protection for the original files, or just as a tool to keep multiple copies of a file without having separate hard copies. Consider it as a tool to create alternative names for the same file.

The **`ln`** command can be used to do this and create two types of links:

- Symbolic links
- Hard links

Those two links are different types of files that point to the original file. A **symbolic link** is a physical file that points to the original file; they are linked and have the same content. Also, it can span different filesystems and physical media, meaning that it can link to original files that are on other drives or partitions with different types of filesystems. The command used is as follows:

```
ln -s [original_filename] [link_filename]
```

Here is an example in which we listed the contents of the `~/packt` directory and then created a symbolic link to the `new-report` file using the `ln -s` command and then listed the contents again:

```
packt@neptune:~$ ls
art-file  backup_dir1  dir1  files  new-files  new-report  users
packt@neptune:~$ ln -s new-report new-report-link
packt@neptune:~$ ls -l
total 28
-rw-rw-r-- 1 packt packt   55 Mar  2 14:46 art-file
drwxrwxr-x 3 packt packt 4096 Mar  2 15:20 backup_dir1
drwxrwxr-x 2 packt packt 4096 Mar  2 15:19 dir1
drwxrwxr-x 2 packt packt 4096 Mar  2 16:04 files
drwxrwxr-x 2 packt packt 4096 Mar  2 16:04 new-files
-rw-rw-r-- 1 packt packt   67 Mar  2 14:30 new-report
lrwxrwxrwx 1 packt packt   10 Mar  2 16:10 new-report-link -> new-report
-rw-rw-r-- 1 packt packt   67 Feb 27 14:33 users
packt@neptune:~$
```

Figure 2.13 – Using symbolic links

You can see that the link created is named `new-report-link` and is visually represented with an `->` arrow that shows the original file that it points to. You can also distinguish the difference in size between the two files, the link and the original one. The permissions are different too. This is a way to know that they are two *different* physical files. To double-check that they are different physical files, you can use the `ls -i` command to show the **inode** before every file. In the following example, you can see that `new-report` and `new-report-link` have different inodes:

```
packt@neptune:~$ ls -li
total 28
137603 -rw-rw-r-- 1 packt packt   55 Mar  2 14:46 art-file
137621 drwxrwxr-x 3 packt packt 4096 Mar  2 15:20 backup_dir1
137604 drwxrwxr-x 2 packt packt 4096 Mar  2 15:19 dir1
137631 drwxrwxr-x 2 packt packt 4096 Mar  2 16:04 files
137638 drwxrwxr-x 2 packt packt 4096 Mar  2 16:04 new-files
137599 -rw-rw-r-- 1 packt packt   67 Mar  2 14:30 new-report
137849 lrwxrwxrwx 1 packt packt   10 Mar  2 16:10 new-report-link -> new-report
131094 -rw-rw-r-- 1 packt packt   67 Feb 27 14:33 users
packt@neptune:~$
```

Figure 2.14 – Comparing the inodes for the symbolic link and original file

If you want to know where the link points to and you do not want to use `ls -l`, there is the **`readlink`** command. It is available in both Ubuntu and CentOS. The output of the command will simply be the name of the file that the symbolic link points to. It only works in the case of symbolic links:

```
packt@neptune:~$ readlink new-report-link
new-report
```

In the preceding example, you can see that the output shows that the **`new-report-link`** file is a symbolic link to the file named **`new-report`**.

In contrast, a **hard link** is a different virtual file that points to the original file. They are both physically the same. The command is simply **`ln`** without any options:

```
ln [original-file] [linked-file]
```

In the following example, we created a hard link for the **`new-report`** file, and we named it **`new-report-ln`**:

```
packt@neptune:~/links$ echo the first report > new-report
packt@neptune:~/links$ cat new-report
the first report
packt@neptune:~/links$ ln new-report new-report-ln
packt@neptune:~/links$ ls -l
total 8
-rw-rw-r-- 2 packt packt 17 Mar  2 16:23 new-report
-rw-rw-r-- 2 packt packt 17 Mar  2 16:23 new-report-ln
packt@neptune:~/links$ echo hard link report >> new-report-ln
packt@neptune:~/links$ cat new-report-ln
the first report
hard link report
packt@neptune:~/links$ cat new-report
the first report
hard link report
packt@neptune:~/links$ ls -li
total 8
138839 -rw-rw-r-- 2 packt packt 34 Mar  2 16:24 new-report
138839 -rw-rw-r-- 2 packt packt 34 Mar  2 16:24 new-report-ln
packt@neptune:~/links$
```

Figure 2.15 – Working with hard links

In the output, you will see that they have the same size and the same inode, and after altering the original file using **`echo`** and output redirection, the changes were available to both files. The two files have a different representation than symbolic links. They appear as two different files in your listing,

with no visual aids to show which file is pointed to. Essentially, a hard link is linked to the inode of the original file. You can see it as a new name for a file, similar to but not identical to renaming it.

Deleting files

In Linux, you have the remove (`rm`) command for deleting files. In its simplest form, the `rm` command is used without an option. For more control over how you delete items, you could use the `-i`, `-f`, and `-r` options:

- `rm -i`: This option enables interactive mode by asking you for acceptance before deleting:

```
packt@neptune:~$ ls
art-file     dir1   links      new-report       users
backup_dir1  files  new-files  new-report-link
packt@neptune:~$ rm -i art-file
rm: remove regular file 'art-file'? y
packt@neptune:~$ ls
backup_dir1  dir1  files  links  new-files  new-report  new-report-link  users
packt@neptune:~$
```

Figure 2.16 – Removing a file interactively

In the preceding example, we deleted `art-file` by using the `-i` option. When asked to interact, you have two options. You can approve the action by typing `y` (yes), or `n` (no) to cancel the action.

- `rm -f` : The `-f` option deletes the file by force, without any interaction from the user:

```
packt@neptune:~$ ls
backup_dir1  dir1  files  links  new-files  new-report  new-report-link  users
packt@neptune:~$ rm -f new-report-link
packt@neptune:~$ ls
backup_dir1  dir1  files  links  new-files  new-report  users
packt@neptune:~$
```

Figure 2.17 – Force remove a file

We deleted the `new-report-link` file created earlier by using the `rm -f` command. It did not ask for our approval and deleted the file directly.

- `rm -r`: This option deletes the files in recursive mode, and it is used to delete multiple files and directories. For example, we will try to delete the `new-files` directory. When using the `rm` command in its simplest way, the output will show an error saying that it cannot delete a directory. But when used with the `-r` option, the directory is deleted right away:

```
packt@neptune:~$ ls
backup_dir1  dir1  files  links  new-files  new-report  users
packt@neptune:~$ rm new-files/
rm: cannot remove 'new-files/': Is a directory
packt@neptune:~$ rm -r new-files/
packt@neptune:~$ ls
backup_dir1  dir1  files  links  new-report  users
packt@neptune:~$
```

Figure 2.18 – Remove a directory recursively

IMPORTANT NOTE

We advise extra caution when using the ***rm*** *command. The most destructive mode is to use* ***rm -rf****. This will delete files, directories, and anything without warning. Pay attention as there is no going back from this. Once used, the damage will be done.*

Most of the time, removing files is a one-way street, with no turning back. This makes the process of deleting files a very important one, and a backup before a deletion could save you a lot of unnecessary stress.

Creating directories

In Linux, you can create a new directory with the **mkdir** command. In the following example, we will create a new directory called **development**:

```
mkdir development
```

If you want to create more directories and sub-directories at once, you will need to use the **-p** option (**p** from the parent), as shown in the following figure:

```
packt@neptune:~$ mkdir -p reports/month/day
packt@neptune:~$ tree reports/
reports/
└── month
    └── day

2 directories, 0 files
packt@neptune:~$
```

Figure 2.19 – Creating parent directories

Directories are files too in Linux, only that they have special attributes. They are essential to organizing your filesystem. For more options with this useful tool, feel free to visit the manual pages.

Deleting directories

The Linux command for removing directories is called **rmdir**. It is designed to default by deleting only empty directories. Let's see what happens if we try to delete a directory that is not empty:

```
packt@neptune:~$ rmdir reports/
rmdir: failed to remove 'reports/': Directory not empty
packt@neptune:~$
```

Figure 2.20 – Using the rmdir command

This is a precautionary measure from the shell, as deleting a directory that is not empty could have disastrous consequences, as we've seen when using the `rm` command. The `rmdir` command does not have a `-i` option such as `rm`. The only way to delete the directory using the `rmdir` command is to delete files inside it first manually. However, the `rm -r` command shown earlier is still useful and more versatile when deleting directories.

Now that you know how to work with directories in Linux, we will proceed to show you commands for file viewing.

Commands for file viewing

As everything in Linux is a file, being able to view and work with file contents is an essential asset for any system administrator. In this section, we will learn commands for file viewing, as almost all files contain text that, at some point, should be readable.

The cat command

This command was used in some of our previous examples in this chapter. It is short for *concatenate* and is used to print the contents of the file to the screen. We have used `cat` several times during this chapter, but here is yet another example. We have two existing files, one called `new-report` and the other called `users`. Let us show you how to use `cat` in the following image:

```
packt@neptune:~$ cat new-report
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor i
ncididunt ut labore et dolore magna aliqua.
packt@neptune:~$ cat new-report users
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor i
ncididunt ut labore et dolore magna aliqua.
packt    pts/0        2023-02-27 13:49 (192.168.124.1)
/home/packt
packt@neptune:~$
```

Figure 2.21 – Example of using the cat command

In this example, we first used the command to show the contents of only one file, the one called `new-report`. The second command was used to show the contents of two files at once, both `new-report` and `users`. The `cat` command is showing the contents of both on the screen. Both files are located in the same directory, which is also the user's working directory. If you would like to concatenate the

contents of files that are not inside your present working directory, you would need to use their absolute path.

The **`cat`** command has several options available, which we will not cover here, as most of the time, its purest form will be the most used. For more details, see the manual pages.

The less command

There are times when a file has so much text that it will cover many screens, and it will be difficult to view on your terminal using just **`cat`**. This is where the **`less`** command is handy. It shows one screen at a time. How much a screen means, it all depends on the size of your terminal window. Let's take, for example, the **`/etc/passwd`** file. It could have multiple lines that you would not be able to fit in just one screen. You could use the following command:

```
$ less /etc/passwd
```

When you press *Enter*, the contents of the file will be shown on your screen. To navigate through it, you could use the following keys:

- Space bar: Move forward one screen
- *Enter*: Move forward one line
- *b*: Move backward one screen
- */*: Enter search mode; this searches forward in your file
- *?*: Search mode; this searches backward in your file
- *v*: Edit your file with the default editor
- *g*: Jump to the beginning of the file
- *Shift* + *g*: Jump to the end of the file
- *q*: Exit the output.

The **`less`** command has a multitude of options that could be used. We advise you to consult the manual pages to find out more information about this command.

The head command

This command is handy when you only want to print to the screen the beginning (the head) of a text file. By default, it will print only the first 10 lines of the file. You can use the same **`/etc/passwd`** file for the head exercise and execute the following command. Watch what happens. It prints the first 10 lines and then exits the command, taking you back to the shell prompt:

```
head /etc/passwd
```

One useful option of this command is to print more or less than 10 lines of the file. For this, you can use the **`-n`** argument or simply just **`-`** with the number of lines you want to print. For the **`/etc/passwd`**

file, we will first use the head command without any options, and then we will use it with the number of lines argument, as shown in the following figure:

```
packt@neptune:~$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
packt@neptune:~$ head -n 3 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
packt@neptune:~$ head -3 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
packt@neptune:~$
```

Figure 2.22 – Using the head command

Many other options that this command provides can prove useful for your work as a system administrator, but we will not cover them here. Feel free to explore them yourself.

The tail command

The **`tail`** command is similar to the **`head`** command, only it prints the last 10 lines of a file, by default. You can use the same **`-n`** argument as for the head command, to see a specific number of lines from the end of a file. However, the **`tail`** command is commonly used for actively watching log files that are constantly changing. It can print the last lines of the file as other applications are writing to it. Take, for example, the following line:

```
tail -f /var/log/syslog
```

Using the **`-f`** option will make the command watch the **`/var/log/syslog`** file as it is being written. It will show you the contents of the file on the screen effectively. The **`-f`** option will cause the tail command to stop during a log rotation, and in this case, the **`-F`** option should be used instead. When using the **`-F`** option, the command will continue to show the output even during a log rotation. To exit that screen, you will need to press *Ctrl + C* to go back to the shell prompt. The following is an example of the output of the previous command:

```
packt@neptune:~$ tail -f /var/log/syslog
Mar  2 18:54:26 neptune systemd[1]: Starting Time & Date Service...
Mar  2 18:54:27 neptune dbus-daemon[643]: [system] Successfully activated servic
e 'org.freedesktop.timedate1'
Mar  2 18:54:27 neptune systemd[1]: Started Time & Date Service.
Mar  2 18:54:27 neptune snapd[653]: storehelpers.go:769: cannot refresh: snap ha
s no updates available: "core20", "lxd", "snapd"
Mar  2 18:54:27 neptune snapd[653]: autorefresh.go:551: auto-refresh: all snaps
are up-to-date
Mar  2 18:54:57 neptune systemd[1]: systemd-timedated.service: Deactivated succe
ssfully.
Mar  2 19:04:26 neptune systemd[1]: Starting Cleanup of Temporary Directories...
Mar  2 19:04:26 neptune systemd[1]: systemd-tmpfiles-clean.service: Deactivated
successfully.
Mar  2 19:04:26 neptune systemd[1]: Finished Cleanup of Temporary Directories.
Mar  2 19:17:01 neptune CRON[1081]: (root) CMD (   cd / && run-parts --report /e
tc/cron.hourly)
Mar  2 19:27:39 neptune systemd[1]: Started Session 4 of User packt.
Mar  2 19:27:47 neptune systemd[1]: Starting Update APT News...
Mar  2 19:27:47 neptune systemd[1]: Starting Update the local ESM caches...
Mar  2 19:27:48 neptune systemd[1]: esm-cache.service: Deactivated successfully.
Mar  2 19:27:48 neptune systemd[1]: Finished Update the local ESM caches.
Mar  2 19:27:48 neptune systemd[1]: apt-news.service: Deactivated successfully.
Mar  2 19:27:48 neptune systemd[1]: Finished Update APT News.
```

Figure 2.23 – The use of tail command for real-time log file observation

Next, let us learn how to view file properties in Linux.

Commands for file properties

There could be times when just viewing the contents of a file is not enough, and you need extra information about that file. There are other handy commands that you could use, and we describe them in the following sections.

The stat command

The `stat` command gives you more information than the `ls` command does. The example in the following figure shows a comparison between the `ls` and `stat` outputs for the same file:

```
packt@neptune:~$ ls
backup_dir1  development  dir1  files  links  new-report  reports  users
packt@neptune:~$ ls -l new-report
-rw-rw-r-- 1 packt packt 124 Mar  2 19:05 new-report
packt@neptune:~$ stat new-report
  File: new-report
  Size: 124             Blocks: 8          IO Block: 4096   regular file
Device: fd00h/64768d    Inode: 137599      Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/   packt)   Gid: ( 1000/   packt)
Access: 2023-03-02 19:06:00.455577063 +0000
Modify: 2023-03-02 19:05:55.243562331 +0000
Change: 2023-03-02 19:05:55.243562331 +0000
 Birth: 2023-03-02 14:24:14.834170882 +0000
packt@neptune:~$
```

Figure 2.24 – Using the stat command

The `stat` command gives you more information about the name, size, number of blocks, type of file, inode, number of links, permissions, UID and GID, and `atime`, `mtime`, and `ctime`. To find out more information about it, please refer to the Linux manual pages.

The file command

This command simply reports on the type of file. Here is an example of a text file and a command file:

```
packt@neptune:~$ file new-report
new-report: ASCII text
packt@neptune:~$ file /usr/bin/wh
whatis             which              whiptail           whoami
whereis            which.debianutils  who
packt@neptune:~$ file /usr/bin/who
/usr/bin/who: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamical
ly linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=98fb99c65957c7
4a5216e69b3dd5032df976a2dc, for GNU/Linux 3.2.0, stripped
packt@neptune:~$ file /usr/bin/which
/usr/bin/which: symbolic link to /etc/alternatives/which
packt@neptune:~$ file /usr/bin/whoami
/usr/bin/whoami: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynami
cally linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=a18184f5d26
fce82fe5ccf842d3cf7b9c729030f, for GNU/Linux 3.2.0, stripped
packt@neptune:~$
```

Figure 2.25 – Using the file command

Linux does not rely on file extensions and types as some other operating systems do. In this respect, the `file` command determines the file type more by its contents than anything else.

Commands for configuring file ownership and permissions

In Linux, **file security** is set by ownership and permissions. **File ownership** is determined by the file's owner and the owner's group. Judging by the owner, a file may have one of three types of ownership assigned to it: *user*, *group*, and *other*. The user is, most of the time, the owner of a file. Whoever created the file is its owner. The owner can be changed using the `chown` command. When setting up group ownership, you determine the permissions for everyone in that group. This is set up using the `chgrp` command. When it comes to other users, the reference is to everyone else on that system, someone who did not create the file, so it is not the owner, and who does not belong to the owner's group. Other is also known as, or referred to as the world.

Besides setting user ownership, the system must know how to determine user behavior, and it does that through the use of **permissions**. We will do a quick review of a file's properties by using the `ls -l` command:

```
packt@neptune:~$ ls -l
total 32
drwxrwxr-x 3 packt packt 4096 Mar  2 15:20 backup_dir1
drwxrwxr-x 2 packt packt 4096 Mar  2 18:51 development
drwxrwxr-x 2 packt packt 4096 Mar  2 15:19 dir1
drwxrwxr-x 2 packt packt 4096 Mar  2 16:04 files
drwxrwxr-x 2 packt packt 4096 Mar  2 16:23 links
-rw-rw-r-- 1 packt packt  124 Mar  2 19:05 new-report
drwxrwxr-x 3 packt packt 4096 Mar  2 18:52 reports
-rw-rw-r-- 1 packt packt   67 Feb 27 14:33 users
```

Figure 2.26 – Long listing output

In the preceding examples, you see two different types of permissions for the files inside our home directory. Each line has 12 characters reserved for special attributes and permissions. Out of those 12, only 10 are used in the preceding examples. Nine of them represent the permissions, and the first one is the file type. There are three easy-to-remember abbreviations for permissions:

- `r` is for `read` permission
- `w` is for `write` permission
- `x` is for `execute` permission
- `-` is for no permission

The nine characters are divided into three regions, each consisting of three characters. The first three characters are reserved for user permissions, the following three characters are reserved for group permissions, and the last three characters represent other, or global permissions.

File types also have their codes, as follows:

- `d`: The letter `d` shows that it is a directory

- `-`: The hyphen shows that it is a file
- `l`: The letter `l` shows that it is a symbolic link
- `p`: The letter `p` shows that it is a named pipe; a special file that facilitates the communication between programs
- `s`: The letter `s` shows that it is a socket, similar to the pipe but with bi-directional and network communications
- `b`: The letter `b` shows that it is a block device; a file that corresponds to a hardware device
- `c`: The letter `c` shows that it is a character device; similar to a block device

The permission string is a 10-bit string. The first bit is reserved for the file type. The next nine bits determine the permissions by dividing them into 3-bit packets. Each packet is expressed by an **octal number** (because an octal number has three bytes). Thus, permissions are represented using a power of two:

- `read` is 2 ^ 2 (two to the power of two), which equals 4
- `write` is 2 ^ 1 (two to the power of one), which equals 2
- `execute` is 2 ^ 0 (two to the power of zero), which equals 1

In this respect, file permissions should be represented according to the following diagram:

```
|  r  |  w  |  x  |  r  |  w  |  x  |  r  |  w  |  x  |
|                 |                 |                 |
   owner / user          group          other / world

 read  write  exec  read  write exec  read  write exec

|  4    2     1   |   4    2     1   |   4    2     1  |
```

Figure 2.27 – File permissions explained

In the preceding diagram, you have the permissions shown as a string of nine characters, just like you would see them in the `ls -la` output. The row is divided into three different sections, one for owner/user, one for group, and one for other/world. These are shown in the first two rows. The other two rows show you the types of permissions (`read`, `write`, and `execute`) and the octal numbers in the following paragraph.

This is useful as it relates the octal representations to the character representations of permissions. Thus, if you were to translate a permission shown as `rwx r-x` into octal, based on the preceding diagram, you could easily say it is `755`. This is because, for the first group, the owner, you have all of them active (`rwx`), which translates into *4+2+1=7*. For the second group, you have only two permissions active, `r` and `x`, which translates into *4+1=5*. Finally, for the last group, you have also two permissions active, similar to the second group (`r` and `x`), which translates to *4+1=5*. Now you know that the permission in the octal is `755`.

As an exercise, you should try to translate into octal the following permissions:

- `rwx rwx`
- `rwx r-x`
- `rwx r-x - - -`
- `rwx - - - - - -`
- `rw- rw- rw-`
- `rw- rw- r - -`
- `rw- rw- - - -`
- `rw- r- - r- -`
- `rw- r- - - - -`
- `rw- - - - - - -`
- `r - - - - - - - -`

IMPORTANT NOTE

*There are some other vital commands such as **`umask`**, **`chown`**, **`chmod`**, and **`chgrp`**, which are used to change or set the default creation mode, owner, mode (access permissions), and group, respectively. They will be briefly introduced here as they involve setting the file's properties, but for a more detailed description, please refer to Chapter 4, Managing Users and Groups.*

Commands for file compression, uncompression, and archiving

In Linux, the standard tool for archiving is called **`tar`**, from tape archive. It was initially used in Unix to write files to external tape devices for archiving. Nowadays, in Linux, it is also used to write to a file in a compressed format. Other popular formats, apart from **`tar`** archives, are **`gzip`** and **`bzip`** for compressed archives, together with the popular **`zip`** from Windows. Now let's look at the **`tar`** command in detail.

The tar command for compressing and un-compressing

This command is used with options and does not offer compression by default. To use compression, we would need to use specific options. Here are some of the most useful arguments available for **`tar`**:

- **`tar -c`**: Creates an archive
- **`tar -r`**: Appends files to an already existing archive
- **`tar -u`**: Appends only changed files to an existing archive
- **`tar -A`**: Appends an archive to the end of another archive
- **`tar -t`**: Lists the contents of the archive
- **`tar -x`**: Extracts the archive contents
- **`tar -z`**: Uses **`gzip`** compression for the archive

- **`tar -j`**: Uses **`bzip2`** compression for the archive
- **`tar -v`**: Uses verbose mode by printing extra information on the screen
- **`tar -p`**: Restores original permission and ownership for the extracted files
- **`tar -f`**: Specifies the name of the output file

There is a chance that in your daily tasks, you will have to use these arguments in combination with each other.

For example, to create an archive of the **`files`** directory, we used the **`-cvf`** arguments combined, as shown here:

```
packt@neptune:~$ ls
backup_dir1  development  dir1  files  links  new-report  reports  users
packt@neptune:~$ tar -cvf files-archive.tar files/
files/
files/files4
files/files1
files/files5
files/files6
files/files3
files/files2
packt@neptune:~$ ls
backup_dir1  dir1   files-archive.tar  new-report  users
development  files  links              reports
```

Figure 2.28 – Using the tar command

The archive created is not compressed. To use compression, we would need to add the **`-z`** or **`-j`** arguments. Next, we will use the **`-z`** option for the **`gzip`** compression algorithm. See the following example and compare the size of the two archive files. As a general rule, it is advised to use an extension for such files:

```
packt@neptune:~$ tar -czvf files-archive-gzipped.tar.gz files
files/
files/files4
files/files1
files/files5
files/files6
files/files3
files/files2
packt@neptune:~$ ls -l files-archive-gzipped.tar.gz files-archive.tar
-rw-rw-r-- 1 packt packt   189 Mar  2 20:08 files-archive-gzipped.tar.gz
-rw-rw-r-- 1 packt packt 10240 Mar  2 20:04 files-archive.tar
```

Figure 2.29 – Compressing a tar archive using gzip

To uncompress a tar archive, you can use the **-x** option (shown at the beginning of this subsection). For example, let us uncompress the **files-archive.tar** file that we created earlier in this subsection, and also add a target directory for the uncompressed files to be added to by using the **-C** option. The target directory needs to be created beforehand. To do this, we will use the following commands:

```
mkdir uncompressed-directory
tar -xvf files-archive.tar -C uncompressed-directory
```

This will extract the files from the archive and add them to the uncompressed-directory directory. To uncompress a **gzip**-compressed archive, for example, **files-archive-gzipped.tar.gz**, we will add the **-z** option to the ones already used in the previous command, as shown in the following snippet:

```
tar -xvzf files-archive-gzipped.tar.gz -C uncompressed-directory
```

There you go, now you know how to archive and unarchive files in Linux. There are other useful archiving tools in Linux, but **tar** is still the most commonly used one. Feel free to explore the others or other useful options for **tar**.

Commands for locating files

Locating files in Linux is an essential task for any system administrator. As Linux systems contain vast numbers of files, finding files might be an intimidating task. Nevertheless, you have handy tools at your disposal, and knowing how to use them will be one of your greatest assets. Among these commands, we will discuss **locate**, **which**, **whereis**, and **find** in the following sections.

The locate command

The **locate** command is not installed by default on Ubuntu. To install it, use the following command to create an index of all the file locations on your system:

```
sudo apt install mlocate
```

Thus, when you execute the command, it searches for your file inside the database. It uses the **updatedb** command as its partner.

Before starting to use the locate command, you should execute **updatedb** to update the location database. After you do that, you can start locating files. In the following example, we will locate any file that has **new-report** in its name:

```
packt@neptune:~$ locate new-report
/home/packt/new-report
/home/packt/links/new-report
/home/packt/links/new-report-ln
packt@neptune:~$
```

Figure 2.30 – Using the locate command

If we were to search for a file with a more generic name, such as **presentation**, the output would be too long and irrelevant. Here is an example where we used output redirection to a file and the **wc** (word count) command to show only the number of lines, words, and bytes of the file to the standard output:

```
packt@neptune:~$ locate presentation > /home/packt/locate-search && wc /home/pac
kt/locate-search
  8   8 663 /home/packt/locate-search
packt@neptune:~$ cat locate-search
/usr/share/mime/application/vnd.ms-powerpoint.presentation.macroenabled.12.xml
/usr/share/mime/application/vnd.oasis.opendocument.presentation-flat-xml.xml
/usr/share/mime/application/vnd.oasis.opendocument.presentation-template.xml
/usr/share/mime/application/vnd.oasis.opendocument.presentation.xml
/usr/share/mime/application/vnd.openxmlformats-officedocument.presentationml.pre
sentation.xml
/usr/share/mime/application/vnd.openxmlformats-officedocument.presentationml.sli
de.xml
/usr/share/mime/application/vnd.openxmlformats-officedocument.presentationml.sli
deshow.xml
/usr/share/mime/application/vnd.openxmlformats-officedocument.presentationml.tem
plate.xml
```

Figure 2.31 – Using the locate command with output redirection and the wc command

In the preceding output, the resulting file has eight lines. This means that there were eight files located that have the string **presentation** in their name. The exact number is used for the words inside the file, as there are no spaces between the paths, so every line is detected as a single word. Also, the resulting file has 663 bytes. Feel free to experiment with other strings. For more options for the **locate** command, please refer to the Linux manual pages.

The which command

This command locates an executable file (program or command) in the shell's search path. For example, to locate the **ls** command, type the following:

```
packt@neptune:~$ which ls
/usr/bin/ls
```

You see that the output is the path of the **ls** command: **/usr/bin/ls**.

Now try it with the **cd** command:

```
which cd
```

You will see that there is no output. This is because the **cd** command is built inside the shell and has no other location for the command to show.

The whereis command

This command finds *only* executable files, documentation files, and source code files. Therefore, it might not find what you want, so use it with caution:

```
packt@neptune:~$ whereis ls
ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz
packt@neptune:~$ whereis cd
cd:
packt@neptune:~$
```

Figure 2.32 – Using the whereis command

Once again, the output for the **cd** command shows nothing relevant, as it is a built-in shell command. As for the **ls** command, the output shows the location of the command itself and the location of the manual pages.

The find command

This command is one of the most powerful commands in Linux. It can search for files in directories and subdirectories based on certain criteria. It has more than 50 options. Its main drawback is the syntax, as it is somehow different from other Linux commands. The best way to learn how the **find** command works is by example. This is why we will show you a large number of examples using this command, hoping that you will become proficient in using it. To see its powerful options, please refer to the manual pages.

The following is a series of exercises using the **find** command, that we thought would be useful for you to know. We will provide the commands to use, but we will not provide you with all the resulting outputs, as some can be fairly long.

- Find, inside the root directory, all the files that have the **e100** string in the name and print them to the standard output:

  ```
  sudo find / -name e100 -print
  ```

- Find, inside the root directory, all the files that have the **file** string in their name and are of type **file**, and print the results to the standard output:

  ```
  sudo find / -name file -type f -print
  ```

- Find all the files that have the **print** string in their name, by looking only inside the **/opt**, **/usr**, and **/var** directories:

  ```
  sudo find /opt /usr /var -name print -type f -print
  ```

- Find all the files in the root directory that have the **`.conf`** extension:

```
sudo find / type f -name "*.conf"
```

- Find all the files in the root directory that have the **`file`** string in their name and no extension:

```
sudo find / -type f -name "file*.*"
```

- Find, in the root directory, all the files with the following extensions: **`.c`**, **`.sh`**, and **`.py`**, and add the list to a file named **`findfile`**:

```
sudo find / -type f \( -name "*.c" -o -name "*.sh" -o -name "*.py" \) > findfile
```

- Find, in the root directory, all the files with the **`.c`** extension, sort them, and add them to a file:

```
sudo find / -type f -name "*.c" -print | sort > findfile2
```

- Find all the files in the root directory, with the permission set to **`0664`**:

```
sudo find / -type f -perm 0664
```

- Find all the files in the root directory that are read-only (have read-only permission) for their owner:

```
sudo find / -type f -perm /u=r
```

- Find all the files in the root directory that are executable:

```
sudo find / -type f -perm /a=x
```

- Find all the files inside the root directory that were modified two days ago:

```
sudo find / -type f -mtime 2
```

- Find all the files in the root directory that have been accessed in the last two days:

```
sudo find / -type f -atime 2
```

- Find all the files that have been modified in the last two to five days:

```
sudo find / -type f -mtime +2 -mtime -5
```

- Find all the files that have been modified in the last 10 minutes:

```
sudo find / -type f -mmin -10
```

- Find all the files that have been created in the last 10 minutes:

```
sudo find / -type f -cmin -10
```

- Find all the files that have been accessed in the last 10 minutes:

```
sudo find / -type f -amin -10
```

- Find all the files that are 5 MB in size:

```
sudo find / -type f -size 5M
```

- Find all the files that have a size between 5 and 10 MB:

```
sudo find / -type f -size +5M -size -10M
```

- Find all the empty files and empty directories:

```
sudo find / -type f -empty
sudo find / -type d -empty
```

- Find all the largest files in the **`/etc`** directory and print to the standard output the first five. Please take into account that this command could be very resource heavy. Do not try to do this for your entire root directory, as you might run out of system memory:

```
sudo find /etc -type f -exec ls -l {} \; | sort -n -r | head -5
```

- Find the smallest first five files in **`/etc`** directory:

```
sudo find /etc -type f -exec ls -s {} \; | sort -n | head -5
```

Feel free to experiment with as many types of find options as you want. The command is very permissive and powerful. Use it with caution.

Commands for text manipulation

Text manipulation is probably the best asset of Linux. It gives you a plethora of tools to work with text at the command line. Some of the more important and widely used ones are **`grep`**, **`tee`**, and the more powerful ones such as **`sed`** and **`awk`**. However, we will come back to those commands in *Chapter 8*, when we will show you how to create and use scripts. In this section, we will only give you a hint on how to use them on the command line.

The grep command

This is one of the most powerful commands in Linux. It is also an extremely useful one. It has the power to search for strings inside text files. It has many powerful options too:

- **`grep -v`**: Show the lines that are not according to the search criteria
- **`grep -l`**: Show only the filenames that match the criteria
- **`grep -L`**: Show only the lines that do *not* comply with the criteria
- **`grep -c`**: A counter that shows the number of lines matching the criteria
- **`grep -n`**: Show the line number where the string was found
- **`grep -i`**: Searches are case insensitive
- **`grep -r`**: Search recursively inside a directory structure
- **`grep -R`**: Search recursively inside a directory structure *AND* follow all symbolic links
- **`grep -E`**: Use extended regular expressions
- **`grep -F`**: Use a strict list of strings instead of regular expressions

Here are some examples of how to use the **grep** command:

- Find out the last time the **sudo** command was used:

```
sudo grep sudo /var/log/auth.log
```

- Search for the **packt** string inside text files from the **/etc** directory:

```
grep -R packt /etc
```

- Show the exact line where the match was found:

```
grep -Rn packt /etc
```

- If you don't want to see the filename of each file where the match was found, use the **-h** option. Then, **grep** will only show you the lines where the match was found:

```
grep -Rh packt /etc
```

- To show only the name of the file where the match was found, use **-l**:

```
grep -Rl packt /etc
```

Most likely, **grep** will be used in combination with shell pipes. Here are some examples:

- If you want to see only the directories from your current working directory, you could pipe the **ls** command output to **grep**. In the following example, we listed only the lines that start with the letter **d**, which represent directories:

```
ls -la | grep '^d'
```

- If you want to display the model of your CPU, you could use the following command:

```
cat /proc/cpuinfo | grep -i 'Model'
```

You will find **grep** to be one of your closest friends as a Linux system administrator, so don't be afraid to dig deeper into its options and hidden gems.

The tee command

This command is very similar to the **cat** command. Basically, it does the same thing, by copying the standard input to standard output with no alteration, but it also copies that into one or more files.

In the following example, we use the **wc** command to count the number of lines inside the **/etc/passwd/** file. We pipe the output to the **tee** command using the **-a** option (append if the file already exists), which writes it to a new file called **no-users** and prints it to the standard output at the same time. We then use the **cat** command to double-check the contents of the new file:

```
packt@neptune:~$ wc -l /etc/passwd | tee no-users
34 /etc/passwd
packt@neptune:~$ cat no-users
34 /etc/passwd
packt@neptune:~$
```

Figure 2.33 – Using the tee command

The `tee` command is more of an underdog of file-manipulating commands. While it is quite powerful, its use can easily be overlooked. Nevertheless, we encourage you to use its powers as often as you can.

In the following section, we will show you how to use text editors from the command line in Linux.

Using text editors to create and edit files

Linux has several command-line text editors that you can use. There are **nano**, **Emacs**, and **Vim**, among others. Those are the most used ones. There are also **Pico**, **JOE**, and **ed** as text editors that are less frequently used than the aforementioned ones. We will cover Vim, as there is a very good chance that you will find it on any Linux system that you work with. Nevertheless, the current trend is to replace Vim with nano as the default text editor. Ubuntu, for example, does not have Vim installed by default, but CentOS does. Fedora is currently looking to make nano the default text editor. Therefore, you might want to learn nano, but for legacy purposes, Vim is a very useful tool to know.

Using Vim to edit text files

Vim is the improved version of **vi**, the default text editor from Unix. It is a very powerful editing tool. This power comes with many options that can be used to ease your work, and this can be overwhelming. In this sub-section, we will introduce you to the basic commands of the text editor, just enough to help you be comfortable using it.

Vim is a mode-based editor, as its operation is organized around different modes. In a nutshell, those modes are as follows:

- `command` mode is the default mode, waiting for a command
- `insert` mode is the text insert mode
- `replace` mode is the text replace mode
- `search` mode is the special mode for searching a document

Let’s see how we can switch between these modes.

Switching between modes

When you first open Vim, you will be introduced to an empty editor that only shows information about the version used and a few help commands. You are in `command` mode. This means that Vim is waiting for a command to operate.

To activate `insert` mode, press *I* on your keyboard. You will be able to start inserting text at the current position of your cursor. You can also press *A* (for append) to start editing to the right of your cursor's position. Both *I* and *A* will activate `insert` mode. To exit the current mode, press the *Esc* key. It will get you back to `command` mode.

If you open a file that already has text in it, while in `command` mode, you can navigate the file using your arrow keys. As Vim inherited the vi workflow, you can also use *H* (to move left), *J* (to move down), *K* (to move up), and *L* (to move right). Those are legacy keys from a time when terminal keyboards did not have separate arrow keys.

While still in `command` mode (the default mode), you can activate `replace` mode by pressing *R* on your keyboard. You can replace the character that is right at the position of your cursor.

While in `command` mode, `search` mode is activated by pressing the */* key. Once in your mode, you can start typing a search string and then press *Enter*.

There is also `last line` mode, or `ex command` mode. This mode is activated by pressing *:*. This is an extended mode where commands such as `w` for saving the file, `q` for quitting, or `wq` for saving and quitting at the same time.

Basic Vim commands

Working with Vim implies that you are comfortable with using keyboard shortcuts for using basic commands. We will guide you to the Vim documentation page (https://vimdoc.sourceforge.net/) for all the commands available, and we will give you a quick glimpse of the most useful ones in the following image:

key	action		key	action
yy	copy a block of text		^	move the cursor to the beginning of the line
p	paste the copied block		$	move the cursor to the end of the line
u	undo the last operation		gg	move the cursor to the beginning of the document
x	delete next character to the right of the cursor		G	move the cursor to the end of the document
X	delete preceding character (relative to the cursor)		:n	move the cursor to line number “n”
dd	delete entire line on which the cursor is positioned		i	insert before the cursor’s position
h	move the cursor left		I	insert at the beginning of the line
l	move the cursor right		a	append at the right of the cursor
k	move the cursor up		A	append at the end of the line
j	move the cursor down		o	insert at the beginning of next line
w	move the cursor right, to the beginning ot the next word		/	activate “search” mode and look for strings
b	move the cursor left, to the beginning of the previous word		?	search backward from the cursor’s position
n	show the next position of the searched string			
N	show the previous position of the searched string			
:wq	save changes and exit Vim			
:q!	enforce exit without saving			
:w!	enforce save without exiting			
ZZ	save and quit the file			
:w file	save as a new file called “file”			

Figure 2.34 – Basic Vim commands

Vim can be quite intimidating for newcomers to Linux. There is no shame if you prefer other editors, as there are plenty to choose from. Now, we will show you a glimpse of nano.

The nano text editor

Vim is a powerful text editor and knowing how to use it is an important thing for any system administrator. Nevertheless, other text editors are equally powerful and even easier to use.

This is the case with **nano**, which is installed by default in Ubuntu and Rocky Linux and can be used right out of the box in both Linux distributions. The default editor is not set up in the `.bashrc` file by using the `$EDITOR` variable. However, in Ubuntu, you can check the default editor on your system by using the following command:

```
packt@neptune:~$ sudo update-alternatives --config editor
There are 4 choices for the alternative editor (providing /usr/bin/editor).

  Selection    Path                Priority   Status
------------------------------------------------------------
* 0            /bin/nano            40        auto mode
  1            /bin/ed             -100       manual mode
  2            /bin/nano            40        manual mode
  3            /usr/bin/vim.basic   30        manual mode
  4            /usr/bin/vim.tiny    15        manual mode

Press <enter> to keep the current choice[*], or type selection number:
```

Figure 2.35 – Checking the default text editor on Ubuntu

You can invoke the nano editor by using the `nano` command on Ubuntu/Debian and Fedora/Rocky Linux or openSUSE. When you type the command, the nano editor will open, with a very straightforward interface that can be easier to use than Vim or Emacs for example. Feel free to use your preferred text editor.

Summary

In this chapter, you learned how to work with the most commonly used commands in Linux. You now know how to manage (create, delete, copy, and move) files, how the filesystem is organized, how to work with directories, and how to view file contents. You now understand the shell and basic permissions. The skills you have learned will help you manage files in any Linux distribution and edit text files. You have learned how to work with Vim, one of the most widely used command-line text editors in Linux. Those skills will help you to learn how to use other text editors such as nano and Emacs. You will use these skills in almost every chapter of this book, as well as in your everyday job as a system administrator.

In the next chapter, you will learn how to manage packages, including how to install, remove, and query packages in both Debian and Red Hat-based distributions. This skill is important for any administrator and must be part of any basic training.

Questions

In our second chapter, we covered the Linux filesystem and the basic commands that will serve as the foundation for the entire book. Here are some questions for you to test your knowledge and for further practice:

1. What is the command that creates a compressed archive with all the files inside the `/etc` directory that use the `.conf` extension?

Hint: Use the `tar` command just as shown in this chapter.

2. What is the command that lists the first five files inside `/etc` and sorts them by dimension in descending order?

Hint: Use `find` combined with `sort` and `head`.

3. What command creates a hierarchical directory structure?

Hint: Use `mkdir` just as shown in this chapter.

4. What is the command that searches for files with three different extensions inside the root?

Hint: Use the `find` command.

5. Find out which commands inside Linux have the **Set owner User ID** (**SUID**) set up.

Hint: Use the `find` command with the `-perm` parameter.

6. Which command is used to create a file with 1,000 lines of randomly generated words (one word per line)?

Hint: Use the `shuf` command (not shown in this chapter).

7. Perform the same exercise as before, but this time generate a file with 1,000 randomly generated numbers.

Hint: Use a `for` loop.

8. How do you find out when **`sudo`** was last used and which commands were executed by it?

Hint: Use the `grep` command.

Further reading

For more information about what was covered in this chapter, please refer to the following Packt titles:

- *Fundamentals of Linux*, by *Oliver Pelz*
- *Mastering Ubuntu Server – Fourth Edition*, by *Jay LaCroix*

3

Linux Software Management

Software management is an important aspect of Linux system administration because, at some level, you will have to work with software packages as a system administrator. Knowing how to work with software packages is an asset that you will master after finishing this chapter.

In this chapter, you will learn how to use specific software management commands, as well as learn how software packages work, depending on your distribution of choice. You will learn about the latest **Snap** and **Flatpak** package types and how to use them on modern Linux distributions.

In this chapter, we're going to cover the following main topics:

- Linux software package types
- Managing software packages
- Installing new desktop environments in Linux

Technical requirements

No special technical requirements are needed for this chapter, just a working installation of Linux on your system. **Ubuntu**, **Fedora** (or **AlmaLinux**), or **openSUSE** are equally suitable for this chapter's exercises as we will cover all types of package managers.

Linux software package types

As you've already learned by now, a Linux distribution comes packed with a kernel and applications on top of it. Although plenty of applications are already installed by default, there will certainly be occasions when you will need to install some new ones or remove ones that you don't need.

In Linux, applications come bundled into **repositories**. A repository is a centrally managed location that consists of software packages maintained by developers. These could contain individual applications or operating system-related files. Each Linux distribution comes with several official repositories, but on top of those, you can add some new ones. The way to add them is specific to each distribution, and we will get into more details later in this chapter.

Linux has several types of packages available. Ubuntu uses `deb` packages, as it is based on Debian, while Fedora (or Rocky Linux and AlmaLinux) uses `rpm` packages, as it is based on RHEL. There is also openSUSE, which uses `rpm` packages too, but it was based on Slackware at its inception. Besides

those, two new package types have been recently introduced – the snap packages developed by Canonical, the company behind Ubuntu, and the flatpak packages, developed by a large community of developers and organizations, including GNOME, Red Hat, and Endless.

The DEB and RPM package types

DEB and RPM are the oldest types of packages and are used by Ubuntu and Fedora, respectively. They are still widely used, even though the two new types mentioned earlier (snaps and flatpaks) are starting to gain ground on Linux on desktops.

Both package types are compliant with the **Linux Standard Base** (**LSB**) specifications. The last iteration of LSB is version 5.0, released in 2015. You can find more information about it at https://refspecs.linuxfoundation.org/lsb.shtml#PACKAGEFMT.

The DEB package's anatomy

DEB was introduced with the Debian distribution back in 1993 and has been in use ever since on every Debian and Ubuntu derivative. A `deb` package is a binary package. This means that it contains the files of the program itself, as well as its dependencies and meta-information files, all contained inside an archive.

To check the contents of a binary `deb` package, you can use the `ar` command. It is not installed by default in Ubuntu 22.04.2 LTS, so you will have to install it yourself using the following command:

```
$ sudo apt install binutils
```

There you go – you have installed a package in Ubuntu! Now, once `ar` has been installed, you can check the contents of any `deb` package. For this exercise, we've downloaded the `deb` package of a password manager called **1password** and checked its contents. To query the package, perform the following steps:

1. Use the `wget` command; the file will be downloaded inside your current working directory:

```
wget https://downloads.1password.com/linux/debian/amd64/stable/1password-latest.deb
```

2. After that, use the `ar t 1password-latest.deb` command to view the contents of the binary package. The `t` option will display a table of contents for the archive:

```
packt@neptune:~$ ar t 1password-latest.deb
debian-binary
control.tar.gz
data.tar.xz
_gpgorigin
packt@neptune:~$
```

Figure 3.1 – Using the ar command to view the contents of a deb file

As you can see, the output listed four files, from which two are archives. You can also investigate the package with the `ar` command.

3. Use the **`ar x 1password-latest.deb`** command to extract the contents of the package to your present working directory:

```
$ ar x 1password-latest.deb
```

4. Use the **`ls`** command to list the contents of your directory. You will see that the four files have been extracted and are ready to inspect. The **`debian-binary`** file is a text file that contains the version of the package file format, which in our case is 2.0. You can concatenate the file to verify your package with the help of the following command:

```
$ cat debian-binary
```

5. The **`control.tar.gz`** archive contains meta-information packages and scripts to be run during the installation or before and after, depending on the case. The **`data.tar.xz`** archive contains the executable files and libraries of the program that are going to be extracted during the installation. You can check the contents with the following command:

```
$ tar tJf data.tar.xz | head
```

6. The last file is a **`gpg`** signature file.

IMPORTANT NOTE

*A **`gpg`** file is a file that uses the GNU Privacy Guard encryption. It uses an encryption standard known as OpenGPG (defined by the RFC4880 standard). It is usually used to sign package files as it offers a safe way for developers to distribute software. For more information on this matter, you can read the official documentation at https://www.openpgp.org/.*

The following screenshot shows the outputs of these commands:

```
packt@neptune:~$ ls
1password-latest.deb  development                  _gpgorigin     package-list
backup_dir1           dir1                         links          poem
control.tar.gz        files                        locate-search  poem-spaces
data.tar.xz           files-archive-gzipped.tar.gz new-report     reports
debian-binary         files-archive.tar            no-users       users
packt@neptune:~$ cat debian-binary
2.0
packt@neptune:~$ tar tJf data.tar.xz | head
./
./usr/
./usr/share/
./usr/share/doc/
./usr/share/doc/1password/
./usr/share/doc/1password/changelog.gz
./usr/share/icons/
./usr/share/icons/hicolor/
./usr/share/icons/hicolor/128x128/
./usr/share/icons/hicolor/128x128/apps/
```

Figure 3.2 – The contents of a deb package

Meta-information for each package is a collection of files that are essential for the programs to run. They contain information about certain package prerequisites and all their dependencies, conflicts, and suggestions. Feel free to explore everything that a package is made of using just the packaging-related commands.

Now that we know what a Debian-based package consists of, let's look at the components of a Red Hat package.

The RPM packages anatomy

The **Red Hat Package Manager** (**RPM**) packages were developed by Red Hat and are used in Fedora, CentOS, RHEL, AlmaLinux, Rocky Linux, SUSE, and openSUSE. RPM binary packages are similar to DEB binary packages in that they are also packaged as an archive.

Let's test the `rpm` package of `1password`, just as we did with the `deb` package in the previous section:

1. Download the `rpm` package using the following command:

```
# wget https://downloads.1password.com/linux/rpm/stable/x86_64/1password-latest.rpm
```

If you want to use the same `ar` command, you will see that in the case of `rpms`, the archiving tool will not recognize the file format. Nevertheless, there are other more powerful tools to use.

2. We will use the `rpm` command, the designated low-level package manager for rpms. We will use the `-q` (query), `-p` (package name), and `-l` (list) options:

```
# rpm -qpl 1password-latest.rpm
```

The output, contrary to the `deb` package, will be a list of all the files related to the application, along with their installation locations for your system.

3. To see the meta-information for the package, run the `rpm` command with the `-q`, `-p`, and `-i` (install) options. The following is a short excerpt from the command's output:

```
[packt@fedora ~]$ rpm -qpi 1password-latest.rpm
warning: 1password-latest.rpm: Header V4 RSA/SHA512 Signature, key ID 2012ea22:
NOKEY
Name        : 1password
Version     : 8.10.0
Release     : 1
Architecture: x86_64
Install Date: (not installed)
Group       : default
Size        : 374273045
License     : LicenseRef-1Password-Proprietary
Signature   : RSA/SHA512, Tue 14 Feb 2023 12:59:08 AM EET, Key ID ac2d62742012ea
22
```

Figure 3.3 – Meta-information of the rpm package

The output will contain information about the application's name, version, release, architecture, installation date, group, size, license, signature, source RPM, build date and host, URL, relocation, and summary.

4. To see which other dependencies the package will require at installation, you can run the same **`rpm`** command with the **`-q`**, **`-p`**, and **`--requires`** options:

```
$ rpm -qp --requires 1password-latest.rpm
```

The output is shown in the following screenshot:

```
[packt@fedora ~]$ rpm -qp --requires 1password-latest.rpm
warning: 1password-latest.rpm: Header V4 RSA/SHA512 Signature, key ID 2012ea22:
NOKEY
/bin/sh
/bin/sh
libasound.so.2()(64bit)
libatk-1.0.so.0()(64bit)
libatk-bridge-2.0.so.0()(64bit)
libdrm.so.2()(64bit)
libgbm.so.1()(64bit)
libgdk_pixbuf-2.0.so.0()(64bit)
libgtk-3.so.0()(64bit)
libnss3.so()(64bit)
libuuid.so.1()(64bit)
libxshmfence.so.1()(64bit)
rpmlib(CompressedFileNames) <= 3.0.4-1
rpmlib(PayloadFilesHavePrefix) <= 4.0-1
rpmlib(PayloadIsXz) <= 5.2-1
udev
```

Figure 3.4 – Package requirements

You now know what Debian and Red Hat packages are and what they contain. DEB and RPM packages are not the only types available on Linux. They are perhaps the most widely used and known, but there are also other types, depending on the distribution you choose. Also, as we stated earlier, there are new packages available for cross-platform Linux use. Those newer packages are called flatpaks and snaps, and we will detail them in the following section.

The snap and flatpak package types

Snap and **Flatpak** are relatively new package types, and they are considered to be the future of apps on Linux. They both build and run applications in isolated containers for more security and

portability. Both have been created to overcome the need for desktop applications' ease of installation and portability.

Even though major Linux distributions have large application repositories, distributing software for so many types of Linux distributions, each with its own kind of package types, can become a serious issue for **independent software vendors** (**ISVs**) or community maintainers. This is where both snaps and flatpaks come to the rescue, aiming to reduce the weight of distributing software.

Let's consider that we are ISVs, aiming to develop our product on Linux. Once a new version of our software is available, we need to create at least two types of packages to be directly downloaded from our website – a `.deb` package for Debian/Ubuntu/Mint and other derivatives, and a `.rpm` package for Fedora/RHEL/SUSE and other derivatives.

But if we want to overcome this and make our app available cross-distribution for most of the existing Linux distributions, we can distribute it as a flatpak or snap. The flatpak package would be available through **Flathub**, the centralized flatpak repository, and the snap package would be available through the Snap Store, the centralized snap repository. Either one is equally suitable for our aim to distribute the app for all major Linux distributions with minimal resource consumption and centralized effort.

IMPORTANT NOTE

Both package types are trying to overcome the overall fragmentation of the Linux ecosystem when it comes to packages. However, these two packages have different philosophies, even though they want to solve the same problem. Snaps emerged as a new type of package that would be available on the IoT and server versions of Canonical's Ubuntu, while flatpaks emerged from the need to have a coherent package type for desktop applications in Linux. Thus, flatpaks are not available on server or IoT versions of Linux, only for desktop editions. As both packages evolve, more and more distributions are starting to provide them by default, with flatpak being the winner in terms of the number of distributions that are offering it by default. On the other hand, snaps are mostly available by default on official Ubuntu versions, starting with version 23.04. Flatpaks are available by default in Fedora, openSUSE, Pop!_OS, Linux Mint, KDE neon, and other distributions.

The takeaway from this situation is that the effort to distribute software for Linux is higher than in the case of the same app packaged for Windows or macOS. Hopefully, in the future, there will be only one universal package for distributing software for Linux, and this will all be for the better for both users and developers alike.

The snap package's anatomy

The snap file is a **SquashFS** file. This means that it has its own filesystem encapsulated in an immutable container. It has a very restrictive environment, with specific rules for isolation and confinement. Every snap file has a meta-information directory that stores files that control its behavior.

Snaps, as opposed to flatpaks, are used not only for desktop applications but also for a wider range of server and embedded apps. This is because Snap has its origins in the Ubuntu **Snappy** for IoT and phones, the distribution that emerged as the beacon of convergence effort from Canonical, Ubuntu's developer.

The flatpak package's anatomy

Flatpak is based on a technology called **OSTree**. The technology was started by developers from GNOME and Red Hat, and it is now heavily used in Fedora Silverblue in the form of **rpm-ostree**. It is a new upgrade system for Linux that is meant to work alongside existing package management systems. It was inspired by Git since it operates similarly. Consider it as a version control system at the OS level. It uses a content-addressed object store, allows you to share branches, and offers transactional upgrades, as well as rollback and snapshot options, for the OS.

Currently, the project has changed its name to **libostree** for a smooth focus on projects that already use the technology. Among many projects that use it, we will bring just two into the discussion: flatpak and rpm-ostree. The rpm-ostree project is considered to be a next-generation hybrid package system for distributions such as Fedora and CentOS/RHEL. They are based on the Atomic project developed by Fedora and Red Hat teams, which brings immutable infrastructure for servers and desktops alike. The openSUSE developers had a similar technology developed called Snapper, which was an OS snapshot tool for its `btrfs` filesystems.

Flatpak uses libostree, which is similar to rpm-ostree, but it is solely used for desktop application containers, with no bootloader management. Flatpak uses sandboxing based on another project named **Bubblewrap**, which allows unprivileged users to access user namespaces and use container features.

Both snaps and flatpaks have full support for graphical installations but also have commands for easier installations and setup from the shell. In the following sections, we will focus solely on command operations for all package types.

Managing software packages

Each distribution has its own **package manager**. There are two types of package managers for each distribution – one for low-level and one for high-level package management. For an RPM-based distribution such as CentOS or Fedora, the low-level tool is the `rpm` command, while the high-level tools are the `yum` and `dnf` commands. For openSUSE, another major RPM-based distribution, the low-level tool is the same `rpm` command, but in terms of high-level tools, the `zypper` command is used. For DEB-based distributions, the low-level command is `dpkg` and the high-level command is `apt` (or the now deprecated `apt-get`).

What is the difference between low-level and high-level package managers in Linux? The low-level package managers are responsible for the backend of any package manipulation and are capable of unpacking packages, running scripts, and installing apps. The high-end managers are responsible for dependency resolution, installing and downloading packages (and groups of packages), and metadata searching.

Managing DEB packages

Usually, for any distribution, package management is handled by the administrator or by a user with root privileges (`sudo`). Package management implies any type of package manipulation, such as installation, search, download, and removal. For all these types of operations, there are specific Linux commands, and we will show you how to use them in the following sections.

The main repositories of Ubuntu and Debian

Ubuntu's official repositories consist of about 60,000 packages, which take the form of binary `.deb` packages or snap packages. The configuration of the system repositories is stored in one file, the `/etc/apt/sources.list` file. Ubuntu has four main repositories, also called package sources, and you will see them detailed inside the `sources.list` file. These repositories are as follows:

- `Main`: Contains free and open source software supported by Canonical
- `Universe`: Contains free and open source software supported by the community
- `Restricted`: Contains proprietary software
- `Multiverse`: Contains software restricted by copyright

All the repositories are enabled by default in the `sources.list` file. If you would like to disable some of them, feel free to edit the file accordingly.

In Debian, the repository information is stored in the same `/etc/apt/sources.list` file. The only difference is that it uses different names for the main package sources, as follows:

- `main`: Contains software that is compliant with Debian's free software guidelines
- `contrib`: Software that can, or not, be compliant with free software guidelines but is not part of the main distribution
- `non-free`: Software that is not open source and is not compliant with free software guidelines

Debian's and Ubuntu's source files are very similar as they have the same information structure inside. What is different is the parts that are specific to each distribution's package source.

Both are based on Debian's **advanced package tool** (**APT**), so we will detail it in the following section.

APT-related commands

Until four years ago, packages in any Debian-based distribution were implemented using the **`apt-get`** command. Since then, a new and improved command called **`apt`** (derived from the abbreviation **APT**) is used as the high-level package manager. The new command is more streamlined and better structured than **`apt-get`**, thus offering a more integrated experience.

Before doing any kind of work with the **`apt`** command, you should update the list of all available packages in your repositories. You can do this with the following command:

```
$ sudo apt update
```

The output of the preceding command will show you if any updates are available. The number of packages that require updates will be shown, together with a command that you could run if you want more details about them.

Before going any further, we encourage you to use the **`apt --help`** command as this will show you the most commonly used APT-related commands. The output is shown in the following screenshot:

```
Most used commands:
  list - list packages based on package names
  search - search in package descriptions
  show - show package details
  install - install packages
  reinstall - reinstall packages
  remove - remove packages
  autoremove - Remove automatically all unused packages
  update - update list of available packages
  upgrade - upgrade the system by installing/upgrading packages
  full-upgrade - upgrade the system by removing/installing/upgrading packages
  edit-sources - edit the source information file
  satisfy - satisfy dependency strings
```

Figure 3.5 – The most commonly used apt commands

Let's dive into some of these in more detail.

Installing and removing packages

Basic system administration tasks include installing and removing packages. In this section, we will show you how to install and remove packages using the **`apt`** command.

To install a new package, you can use the **`apt install`** command. We used this command at the beginning of this chapter when we talked about the DEB package's anatomy. Remember that we had to install the **`ar`** command as an alternative to inspect **`.deb`** packages. Back then, we used the following command:

```
$ sudo apt install binutils
```

This command installed several packages on the system and among them the one that we need to fulfill our action. The **`apt`** command automatically installs any requisite dependencies too.

To remove a package, you can use the **`apt remove`** or **`apt purge`** command. The first one removes the installed packages and all their dependencies installed by the **`apt install`** command. The latter will uninstall the packages, just like **`apt remove`**, but also deletes any configuration files created by the applications.

In the following example, we are removing the **`binutils`** applications we installed previously using the **`apt remove`** command:

```
$ sudo apt remove binutils
```

The output will show you a list of packages that are no longer needed and remove them from the system, asking for your confirmation to continue. This is a very good safety measure as it allows you to review the files that are going to be deleted. If you feel confident about the operation, you can add a **`-y`** parameter at the end of the command, which tells the shell that the answer to any question provided by the command will automatically be *Yes*.

However, using **`apt remove`** will not remove all the configuration files related to the removed application. To see which files are still on your system, you can use the **`find`** command. For example, to see the files related to the **`binutils`** package that were not removed, we can use the following command:

```
sudo find / -type d -name *binutils 2>/dev/null
```

The output will show the directories (hence the **`-type d`** option that was used with the command), where binutils-related files remain after the removal of the package.

Another tool that's used to remove packages and all the configuration files associated with them is **`apt purge`**. If you want to use the **`apt purge`** command instead of **`apt remove`**, you can use it as follows:

```
$ sudo apt purge binutils
```

The output is similar to the **`apt remove`** command, showing you which packages will be removed and how much space will be freed on the disk, and asking for your confirmation to continue the operation.

IMPORTANT NOTE

*If you plan on using **`apt purge`** to remove the same package (in our case, **`binutils`**), you will have to install it again as it was removed using the **`apt remove`** command.*

The **apt remove** command has a **purge** option too, which has the same outcome as the **apt purge** command. The syntax is as follows:

```
sudo apt remove --purge [packagename]
```

As stated earlier, when using the **apt remove** command, some configuration files are left behind in case the operation was an accident and the user wants to revert to the previous configuration. The files that are not deleted by the **remove** command are small user configuration files that can easily be restored. If the operation was not an accident and you still want to get rid of all the files, you can still use the **apt purge** command to do that by using the same name as those of the already removed packages.

Upgrading the system

Now and then, you will need to perform a system upgrade to ensure that you have all the latest security updates and patches installed. In Ubuntu and Debian, you will always use two different commands to accomplish this. One is **apt update**, which will update the repository list and makes sure it has all the information concerning the updates that are available for the system. The other command is **apt upgrade**, which upgrades the packages. You can use them both in the same command using metacharacters:

```
$ sudo apt update; sudo apt upgrade
```

The **update** command will sometimes show you which packages are no longer required, with a message similar to the following:

```
The following packages were automatically installed and are no longer required:
  libfprint-2-tod1 libllvm9
Use 'sudo apt autoremove' to remove them.
```

You can use the **sudo apt autoremove** command to remove the unneeded packages after you perform the upgrade. The **autoremove** command's output will show you which packages will be removed and how much space will be freed on the disk and will ask for your approval to continue the operation.

Let's say that during our work with Ubuntu, a new distribution is released, and we would like to use that as it has newer packages of the software we use. Using the command line, we can make a full distribution upgrade. The command for this action is as follows:

```
$ sudo apt dist-upgrade
```

Similarly, we can also use the following command:

```
$ sudo apt full-upgrade
```

Upgrading to a newer distribution version should be a flawless process, but this is not always a guarantee. It all depends on your custom configurations. No matter the situation, we advise you to do a full system backup before upgrading to a new version.

Managing package information

Working with packages sometimes implies the use of information-gathering tools. Simply installing and removing packages is not enough. You will need to search for certain packages to show details about them, create lists based on specific criteria, and so on.

To search for a specific package, you can use the `apt search` command. It will list all the packages that have the searched string in their name, as well as others that use the string in various ways. For example, let's search for the `nmap` package:

```
$ sudo apt search nmap
```

The output will show a considerably long list of packages that use the `nmap` string in various ways. You will still have to scroll up and down the list to find the package you want. For better results, you can pipe the output to the `grep` command, but you will notice a warning, similar to the one shown in the following screenshot:

```
packt@neptune:~$ sudo apt search nmap | grep nmap
[sudo] password for packt:

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

  IPv4 address parser for the nmap format
libcoq-mathcomp-finmap/jammy 1.5.1-1 amd64
libnmap-parser-perl/jammy 1.37-1 all
  module to parse nmap scan results with perl
librust-cpp-synmap-dev/jammy 0.3.0-1 amd64
  Library for doing location lookup based on free openwlanmap.org data
  Library for doing location lookup based on free openwlanmap.org data
nmap/jammy-updates 7.91+dfsg1+really7.80+dfsg1-2ubuntu0.1 amd64
nmap-common/jammy-updates 7.91+dfsg1+really7.80+dfsg1-2ubuntu0.1 all
  Architecture independent files for nmap
nmapsi4/jammy 0.5~alpha2-3 amd64
  graphical interface to nmap, the network scanner
python-libnmap-doc/jammy 0.7.2-1 all
python3-libnmap/jammy 0.7.2-1 all
python3-nmap/jammy 0.6.1-1.1 all
```

Figure 3.6 – Output of the apt search command

Following the warning, the output shows a short list of packages that contain the `nmap` string, and among them is the actual package we are looking for, as highlighted in *Figure 3.5*.

To overcome that warning, you can use a legacy command called **apt-cache search**. By running it, you will get a list of packages as output, but it won't be as detailed as the output of the **apt search** command:

```
packt@neptune:~$ sudo apt-cache search nmap | grep nmap
golang-github-malfunkt-iprange-dev - IPv4 address parser for the nmap format
libcoq-mathcomp-finmap - finite sets and maps extension for Mathematical Compone
nts
libnmap-parser-perl - module to parse nmap scan results with perl
librust-cpp-synmap-dev - Sourcemap and full crate parsing support for `cpp_syn`
- Rust source code
libwlocate-dev - Library for doing location lookup based on free openwlanmap.org
 data
libwlocate0 - Library for doing location lookup based on free openwlanmap.org da
ta
nmap - The Network Mapper
nmap-common - Architecture independent files for nmap
nmapsi4 - graphical interface to nmap, the network scanner
```

Figure 3.7 – The output of the apt-cache command

Now that we know that the **nmap** package exists in Ubuntu repositories, we can investigate it further by showing more details using the **apt show** command:

```
$ apt show nmap
```

The output will show a detailed description, including the package's name, version, priority, origin and section, maintainer, size, dependencies, suggested extra packages, download size, APT sources, and description.

apt also has a useful **list** command, which can list packages based on certain criteria. For example, if we use the **apt list** command alone, it will list all the packages available. But if we use different options, the output will be personalized.

To show the installed packages, we can use the **-- installed** option:

```
$ sudo apt list --installed
```

To list all the packages, use the following command:

```
$ sudo apt list
```

For comparative reasons, we will redirect each output to a different file, and then compare the two files. This is an easier task to do to see the differences between the two outputs since the lists are reasonably large. We will now run the specific commands, as follows:

```
$ sudo apt list --installed > list-installed
$ sudo apt list > list
```

You can compare the two resulting files by using the `ls -la` command and observe the difference in size. You will see that the `list` file will be significantly larger than the `list-installed` file.

There are other ways in which to compare the two outputs, and we would like to let you discover them by yourself, as an exercise for this sub-section. Feel free to use any other APT-related commands you would like, and practice with them enough to get familiar with their use. APT is a powerful tool, and every system administrator needs to know how to use it to sustain a usable and well-maintained Linux system. Usability is closely related to the apps that are used and their system-wide optimization.

Managing RPM packages

RPM packages are the equivalent packages for Linux distributions such as Fedora, AlmaLinux, Rocky Linux, RHEL, and openSUSE/SLES. They have dedicated high-level tools, including `dnf`, `yum`, and `zypper`. The low-level tool is the `rpm` command.

In RHEL, the default package manager is **Yellow Dog Updater, Modified** (**YUM**) and it is based on **Dandified YUM** (**DNF**), the default package manager in Fedora. If you use both Fedora and RHEL, for ease of use, you can use only one of those as they are the same command. For consistency, we will use YUM for all the examples in this chapter.

YUM is the default high-level manager. It can install, remove, update, and package queries, as well as resolve dependencies. YUM can manage packages installed from repositories or local `.rpm` packages.

The main repositories in Fedora/RHEL-based distributions

Repositories are all managed from the `/etc/yum.repos.d/` directory, with configuration available inside the `/etc/yum.conf` file. If you do a listing for the `repos` directory, the output will be similar to the following screenshot:

```
[packt@venus ~]$ ls -l /etc/yum.repos.d/
total 44
-rw-r--r--. 1 root root 1019 Nov 15 09:42 almalinux-appstream.repo
-rw-r--r--. 1 root root  983 Nov 15 09:42 almalinux-baseos.repo
-rw-r--r--. 1 root root  947 Nov 15 09:42 almalinux-crb.repo
-rw-r--r--. 1 root root  983 Nov 15 09:42 almalinux-extras.repo
-rw-r--r--. 1 root root 1103 Nov 15 09:42 almalinux-highavailability.repo
-rw-r--r--. 1 root root  947 Nov 15 09:42 almalinux-nfv.repo
-rw-r--r--. 1 root root  959 Nov 15 09:42 almalinux-plus.repo
-rw-r--r--. 1 root root 1103 Nov 15 09:42 almalinux-resilientstorage.repo
-rw-r--r--. 1 root root  935 Nov 15 09:42 almalinux-rt.repo
-rw-r--r--. 1 root root  995 Nov 15 09:42 almalinux-saphana.repo
-rw-r--r--. 1 root root  947 Nov 15 09:42 almalinux-sap.repo
[packt@venus ~]$
```

Figure 3.8 – RHEL derivative repositories

All these files listed contain vital information about the repository, such as its name, mirror list, the **gpg** key's location, and enabled status. All the ones listed are official repositories.

YUM-related commands

YUM has many commands and options, but the most commonly used ones are related to package installation, removal, search, information query, system update, and repository listing.

Installing and removing packages

To install a package from a repository in AlmaLinux/Rocky Linux (or Fedora), simply run the **yum install** command. In the following example, we will install the GIMP application from the command line:

```
$ sudo yum install gimp
```

If you already have a package downloaded and would like to install it, you can use the **yum localinstall** command. Here, we have downloaded the 1password **.rpm** package:

```
wget https://downloads.1password.com/linux/rpm/stable/x86_64/1password-latest.rpm
```

Then, we installed it with the following command:

```
sudo yum localinstall 1password-latest.rpm
```

The **localinstall** command automatically resolves the dependencies needed and shows the source (repository) for each of them.

This is a very powerful command that makes using the **rpm** command itself almost redundant in some cases. The main difference between the **yum install** and **yum localinstall** commands is that the

latter is capable of solving dependencies for locally downloaded packages. While the former looks for packages inside the active repositories, the latter looks for packages to install in the current working directory.

To remove a package from the system, use the **`yum remove`** command. We will remove the newly installed 1password package:

```
sudo yum remove 1password.x86_64
```

You will be asked if you want to remove all the packages that the application installed. Choose accordingly and proceed.

IMPORTANT NOTE

The default action for pressing the Enter or Return key while inside a command dialogue in Fedora or RHEL derivatives is N (for no, or negative), while in Ubuntu, the default action is set to Y (for yes). This is a precautionary safety measure, which requires your extra attention and intervention.

The output, very similar to the output of the installation command, will show you which packages and dependencies will be removed if you proceed with the command.

As you can see, all the dependencies installed with the package using the **`yum localinstall`** command will be removed using the **`yum remove`** command. If you're asked to proceed, type **`y`** and continue with the operation.

Updating the system

To upgrade a Fedora/RHEL-based system, we can use the **`yum upgrade`** command. There is also a **`yum update`** command, which has the same effect by updating the installed packages:

```
$ sudo yum upgrade
```

You can use the **`-y`** option to automatically respond to the command's questions.

There is also an **`upgrade-minimal`** command, which installs only the newest security updates for packages.

Managing package information

Managing files with **`yum`** is very similar to managing files with **`apt`**. There are plenty of commands to use, and we will detail some of them – the ones we consider to be the most commonly used. To find out more about those commands and their use, run **`yum --help`**.

To see an overview of the **`yum`** command history and which package was managed, use the following command:

```
$ sudo yum history
```

This will give you an output that shows every **yum** command that was run, how many packages were altered, and the time and date when the actions were executed, as in the following example:

```
[packt@venus ~]$ sudo yum history
ID     | Command line             | Date and time    | Action(s)      | Altered
-------------------------------------------------------------------------------
     4 | remove 1password.x86_64  | 2023-03-05 10:32 | Removed        |  118 EE
     3 | localinstall 1password-la | 2023-03-05 10:31 | Install        |  118 EE
     2 | update -y                | 2023-03-03 21:53 | I, U           |  106  <
     1 |                          | 2023-03-03 21:38 | Install        |  604 >E
[packt@venus ~]$
```

Figure 3.9 – Using the yum history command

To show details about a certain package, we have the **yum info** command. We will query the **nmap** package, similar to what we did in Ubuntu. In CentOS, the command will be as follows:

```
Available Packages
Name         : nmap
Epoch        : 3
Version      : 7.91
Release      : 10.el9
Architecture : x86_64
Size         : 5.4 M
Source       : nmap-7.91-10.el9.src.rpm
Repository   : appstream
Summary      : Network exploration tool and security scanner
URL          : http://nmap.org/
License      : Nmap
Description  : Nmap is a utility for network exploration or security auditing.
             : It supports ping scanning (determine which hosts are up), many
             : port scanning techniques (determine what services the hosts are
             : offering), and TCP/IP fingerprinting (remote host operating
             : system identification). Nmap also offers flexible target and port
```

Figure 3.10 – Using the yum info command

The output will show you the name, version, release, source, repository, and description, very similar to what we saw with the **.deb** packages.

To list all the installed packages or all the packages for that matter, we can use the **yum list** command:

```
# yum list
```

To see only the installed packages, run the following command:

```
# yum list installed
```

If we redirect the output of each command to specific files and then compare the two files, we will see the differences between them, similar to what we did in Ubuntu. The output shows the name of the packages, followed by the version and release number, and the repository from which it was installed. Here is a short excerpt:

```
xml-common.noarch        0.6.3-58.el9             @AppStream
xz.x86_64                5.2.5-8.el9_0            @anaconda
xz-libs.x86_64           5.2.5-8.el9_0            @anaconda
yajl.x86_64              2.1.0-21.el9_0           @AppStream
yum.noarch               4.12.0-4.el9.alma        @anaconda
zip.x86_64               3.0-33.el9               @anaconda
zlib.x86_64              1.2.11-35.el9_1          @baseos
```

Figure 3.11 – Excerpt of the yum list installed command

As we have covered the most commonly used commands for both DEB and RPM files, we did not cover a specific package manager for openSUSE and SUSE SLE called **Zypper**. We will quickly show you some commands to get you acquainted with Zypper and let you give openSUSE a try next.

Working with Zypper

In the case of openSUSE, **Zypper** is the package manager, similar to APT and DNF from Debian/Ubuntu and Fedora/RHEL. The following sections cover some useful commands.

Installing and removing packages

Similar to using APT and DNF, the Zypper package manager in openSUSE is used to install and remove packages using almost the same syntax. For example, we will install `nmap` using the `zypper` command. But first, let's search for the name in the respective repositories to see if it exists. We will use the following command:

```
sudo zypper search nmap
```

The output of this command is a list of packages containing the `nmap` string in their name, together with the type and a summary:

```
packt@localhost:~> sudo zypper search nmap
[sudo] password for root:
Loading repository data...
Reading installed packages...

S | Name                                | Summary                          | Type
--+-------------------------------------+----------------------------------+--------
  | nmap                                | Network exploration tool and->   | package
  | nmap-parse-output                   | A tool for analyzing Nmap sc->   | package
  | nmap-parse-output-bash-completion   | Bash Completion for nmap-par->   | package
  | nmapsi4                             | A Graphical Front-End for Nmap   | package
  | python3-chainmap                    | Backport/clone of ChainMap f->   | package
  | python3-dragonmapper                | Identification and conversio->   | package
packt@localhost:~>
```

Figure 3.12 – Using the zypper search command in openSUSE

You will notice `s` in the first column of the list. It consists of the status of the package, and the output will be different if the package was already installed.

From the search output, we can see that the name of the package for the Nmap application is **`nmap`** (it could have been a different name, hence why we used the **`search`** command in the first place), so we will proceed and install it on our system. We will use the **`zypper install`** command to do so.

IMPORTANT NOTE

*In openSUSE, you can use short versions of the command. For example, instead of using **`zypper install`**, you can use **`zypper in`**, followed by the name of the package you want to install. The same goes for **`zypper update`**, which can be used as **`zypper up`**, and also for **`dist-upgrade`**, where you can use **`dup`**. Alternatively, you can use the **`zypper remove`** command as **`zypper rm`**. Check the manual pages for more information.*

So, here is the command to install the **`nmap`** package:

```
sudo zypper install nmap
```

Alternatively, you can use the following command:

```
sudo zypper in nmap
```

You can see the output in the following screenshot:

```
packt@localhost:~> sudo zypper in nmap
Loading repository data...
Reading installed packages...
Resolving package dependencies...

The following 3 NEW packages are going to be installed:
  libpcap1 libssh2-1 nmap

3 new packages to install.
Overall download size: 5.6 MiB. Already cached: 0 B. After the operation,
additional 24.4 MiB will be used.
Continue? [y/n/v/...? shows all options] (y):
```

Figure 3.13 – Using the zypper in command

The output shows which packages will be installed. What is important to notice here is that Zypper is automatically dealing with dependencies, just like other package managers. Besides **`nmap`**, there are two more library packages ready to be installed. Type **`y`** to continue the installation and the packages will be installed.

Now, let's use the **`zypper search nmap`** command one more time to see how the list has changed when it comes to showing the package information about **`nmap`**:

```
S  | Name                                | Summary                          | Type
---+-------------------------------------+----------------------------------+--------
i+ | nmap                                | Network exploration tool an-> | package
   | nmap-parse-output                   | A tool for analyzing Nmap s-> | package
   | nmap-parse-output-bash-completion | Bash Completion for nmap-pa-> | package
   | nmapsi4                             | A Graphical Front-End for N-> | package
   | python3-chainmap                    | Backport/clone of ChainMap -> | package
   | python3-dragonmapper                | Identification and conversi-> | package
```

Figure 3.14 – Checking the status of nmap with zypper search

In the output, you will see that the first column of the list has **`i+`** in front of the **`nmap`** package we just installed. This means that the package and its dependencies are already installed. So, if you are searching for some package and it is already installed, you will know this by checking the first column of the list, which is the status column.

Now, let's remove the same package we already installed. We will use the following command:

```
sudo zypper remove nmap
```

Alternatively, we can use the following command:

```
sudo zypper rm nmap
```

The output is shown in the following screenshot:

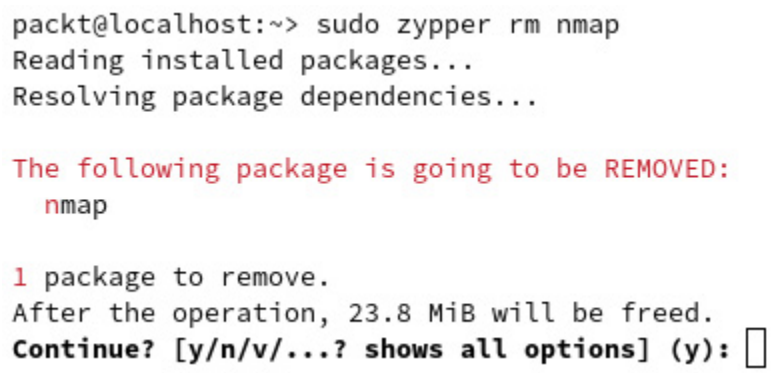

Figure 3.15 – Using the zypper remove command

The output of this command shows which packages are going to be removed. As you can see, only the **`nmap`** package will be removed; the other dependencies installed that were alongside won't be removed. To remove them together with the package, use the **`--clean-deps`** argument when using the command. Details are shown in the following screenshot:

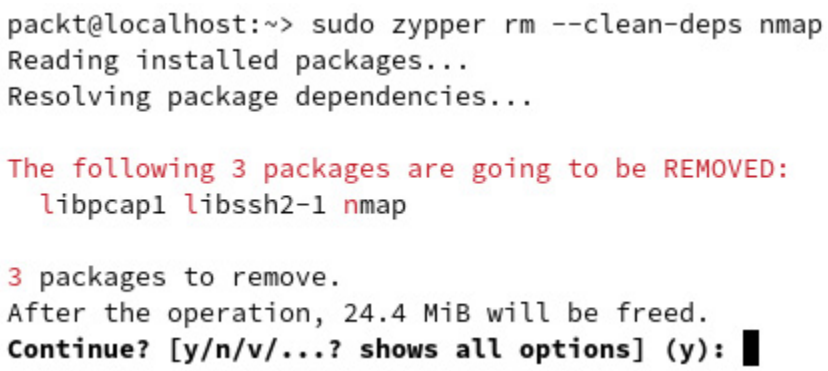

Figure 3.16 – Removing dependencies

Now that you've learned how to use **`zypper`** to install and remove packages in openSUSE, let's learn how to use it to update or upgrade the entire system.

Upgrading and updating the system

Before updating a system, you might want to see which updates are available. For this, you can use the following command:

```
zypper list-updates
```

The output of this command will show all the updates available on your system. To install the updates, use the following command:

```
sudo zypper update
```

An alternative is the following command:

```
sudo zypper up
```

If you use these commands with no parameters, as we just showed, all the available updates will be installed. You can also update individual packages by including the package name as a parameter for the **`update`** command.

Some more useful commands in openSUSE are used for adding and managing locks to a package if we don't want it to be updated or removed. Let's learn how to do this using the same **`nmap`** package. If you removed it as we did in the previous section, please install it again. We will add a lock, check for that lock, and then remove the lock for the package.

To add a lock to a package, use the **`add-lock`** or **`zypper al`** command. To see the locked packages on your system, you can use the **`zypper ll`** command (list locks); to remove a lock from a package, you can use the **`zypper rl`** command (remove locks):

```
packt@localhost:~> sudo zypper al nmap
Specified lock has been successfully added.
packt@localhost:~> sudo zypper ll

# | Name | Type    | Repository | Comment
--+------+---------+------------+--------
1 | nmap | package | (any)      |

packt@localhost:~> sudo zypper rl nmap
1 lock has been successfully removed.
```

Figure 3.17 – Adding and removing locks to and from packages with Zypper

Now, let's lock the **`nmap`** package again and try to remove it. You will see that the package will not be removed. First, you will be asked what to do to remove it. Details are shown in the following figure:

```
packt@localhost:~> sudo zypper al nmap
packt@localhost:~> sudo zypper remove nmap
Reading installed packages...
Resolving package dependencies...

Problem: conflicting requests
 Solution 1: remove lock to allow removal of nmap-7.92-150400.1.8.x86_64
 Solution 2: do not ask to delete all solvables providing nmap.x86_64 = 7.92-150
400.1.8

Choose from above solutions by number or cancel [1/2/c/d/?] (c):
```

Figure 3.18 – Trying to remove a locked package

Updating is straightforward, and you also learned how to use the lock option in Zypper to protect different packages. Now that you know how to update your openSUSE system, we'll learn how we

can find information about certain packages in the following section.

Managing package information

As shown when using the APT and DNF package managers in Ubuntu and Fedora, we can use Zypper in openSUSE to obtain information about packages. Let's use the same **`nmap`** package as in the previous section and obtain more information about it. To do this, we will use the **`zypper info`** command:

```
sudo zypper info nmap
```

As shown in *Figure 3.19*, the information provided is similar to that in Ubuntu and RHEL-based distributions. As we uninstalled the **`nmap`** package, the information shown in the output will state that the package is not installed. There is also a longer description for the package, which we did not include in the following screenshot:

```
------------------------------
Repository     : Main Repository
Name           : nmap
Version        : 7.92-150400.1.8
Arch           : x86_64
Vendor         : SUSE LLC <https://www.suse.com/>
Installed Size : 23.8 MiB
Installed      : No
Status         : not installed
Source package : nmap-7.92-150400.1.8.src
Upstream URL   : https://nmap.org/
Summary        : Network exploration tool and security scanner
Description    :
```

Figure 3.19 – Using the zypper info command

Now, let's learn how to manage flatpaks and snaps on a Linux machine.

Using the snap and flatpak packages

Snaps and flatpaks are relatively new package types that are used in various Linux distributions. In this section, we will show you how to manage those types of packages. For snaps, we will use Ubuntu as our test distribution, while for flatpaks, we will use Fedora, even though, with a little bit of work, both package types can work on either distribution.

Managing snap packages on Ubuntu

Snap is installed by default in Ubuntu 22.04.2 LTS. Therefore, you don't have to do anything to install it. Simply start searching for the package you want and install it on your system. We will use the Slack application to show you how to work with snaps.

Searching for snaps

Slack is available in the Snap Store, so you can install it. To make sure, you can search for it using the **`snap find`** command, as in the following example:

```
$ snap find "slack"
```

In the command's output, you will see many more packages that contain the **`slack`** string or are related to the Slack application, but only the first on the list is the one we are looking for.

IMPORTANT NOTE

*In any Linux distribution, two apps originating from different packages and installed with different package managers can coexist. For example, Slack can be installed using the **`deb`** file provided by the website, as well as the one installed from the Snap Store.*

If the output says that the package is available, we can proceed and install it on our system.

Installing a snap package

To install the **`snap`** package for Slack, we can use the **`snap install`** command:

```
packt@neptune:~$ sudo snap install slack
[sudo] password for packt:
slack 4.29.149 from Slack✓ installed
packt@neptune:~$
```

Figure 3.20 – Installing the Slack snap package

Next, let's see how we can find out more about the **`snap`** package we just installed.

Snap package information

If you want to find out more about the package, you can use the **`snap info`** command:

```
$ snap info slack
```

The output will show you relevant information about the package, including its name, summary, publisher, description, and ID. The last piece of information that's displayed will be about the available **channels**, which are as follows in the case of our Slack package:

```
commands:
  - slack
snap-id:      JUJH91Ved74jd4ZgJCpzMBtYbPOzTlsD
tracking:     latest/stable
refresh-date: today at 08:46 UTC
channels:
  latest/stable:     4.29.149 2022-11-28 (68) 121MB -
  latest/candidate:  ↑
  latest/beta:       ↑
  latest/edge:       ↑
  insider/stable:    -
  insider/candidate: -
  insider/beta:      -
  insider/edge:      4.25.1   2022-04-01 (61) 108MB -
installed:           4.29.149            (68) 121MB -
```

Figure 3.21 – Snap channels shown for the Slack app

Each channel contains information about a specific version and it is important to know which one to choose. By default, the stable channel will be chosen by the `install` command, but if you would like a different version, you could use the `--channel` option during installation. In the preceding example, we used the default option.

Displaying installed snap packages

If you want to see a list of the installed snaps on your system, you can use the `snap list` command. Even though we only installed Slack on the system, in the output, you will see that many more apps have been installed. Some, such as `core` and `snapd`, are installed by default from the distribution's installation and are required by the system:

```
packt@neptune:~$ snap list
Name              Version         Rev    Tracking       Publisher   Notes
bare              1.0             5      latest/stable  canonical✓  base
core18            20230207        2697   latest/stable  canonical✓  base
core20            20230126        1822   latest/stable  canonical✓  base
gnome-3-34-1804   0+git.3556cb3   77     latest/stable  canonical✓  -
gtk-common-themes 0.1-81-g442e511 1535   latest/stable  canonical✓  -
lxd               5.0.2-838e1b2   24322  5.0/stable/…   canonical✓  -
slack             4.29.149        68     latest/stable  slack✓      -
snapd             2.58.2          18357  latest/stable  canonical✓  snapd
```

Figure 3.22 – Output of the snap list command

Now, we'll learn how to update a snap package.

Updating a snap package

Snaps are automatically updated. Therefore, you won't have to do anything yourself. The least you can do is check whether an update is available and speed up its installation using the **`snap refresh`** command, as follows:

```
$ sudo snap refresh slack
```

Following an update, if you want to go back to a previously used version of the app, you can use the **`snap revert`** command, as in the following example:

```
$ sudo snap revert slack
```

In the next section, we'll learn how to enable and disable snap packages.

Enabling or disabling snap packages

If we decide to not use an application temporarily, we can disable that app using the **`snap disable`** command. If we decide to reuse the app, we can enable it again using the **`snap enable`** command:

```
packt@neptune:~$ snap disable slack
error: access denied (try with sudo)
packt@neptune:~$ sudo snap disable slack
slack disabled
packt@neptune:~$ sudo snap enable slack
slack enabled
packt@neptune:~$
```

Figure 3.23 – Enabling and disabling a snap app

Remember to use **`sudo`** to enable and disable a snap application. If disabling is not what you are looking for, you can completely remove the snap.

Removing a snap package

When removing a snap application, the associated configuration files, users, and data are also removed. You can use the **`snap remove`** command to do this, as in the following example:

```
$ sudo snap remove slack
```

After removal, an application's internal user, configuration, and system data are saved and retained for 31 days. These files are called snapshots, they are archived and saved under **`/var/lib/snapd/snapshots`**, and they contain the following types of files: a **`.json`** file containing a description of the snapshot, a **`.tgz`** file containing system data, and specific **`.tgz`** files that contain

each system's user details. A short listing of the aforementioned directory will show the automatically created snapshot for Slack:

```
packt@neptune:~$ sudo snap remove slack
slack removed
packt@neptune:~$ sudo ls /var/lib/snapd/snapshots/
1_slack_4.29.149_68.zip
packt@neptune:~$
```

Figure 3.24 – Showing the existing snapshots after removal

If you don't want the snapshots to be created, you can use the **`--purge`** option for the **`snap remove`** command. For applications that use a large amount of data, those snapshots could have a significant size and impact the available disk space. To see the snapshots saved on your system, use the **`snap saved`** command:

```
packt@neptune:~$ sudo snap saved
Set  Snap   Age    Version   Rev  Size    Notes
1    slack  3m40s  4.29.149  68     673B  auto
packt@neptune:~$
```

Figure 3.25 – Showing the saved snapshots

The output shows that in the list, in our case, just one app has been removed, with the first column indicating the ID of the snapshot (**`set`**). If you would like to delete a snapshot, you can do so by using the **`snap forget`** command. In our case, to delete the Slack application's snapshot, we can use the following command:

```
packt@neptune:~$ sudo snap saved
Set  Snap   Age    Version   Rev  Size    Notes
1    slack  3m40s  4.29.149  68     673B  auto
packt@neptune:~$ sudo snap forget 1
Snapshot #1 forgotten.
packt@neptune:~$ sudo snap saved
No snapshots found.
packt@neptune:~$
```

Figure 3.26 – Using the snap forget command to delete a snapshot

To verify that the snapshot was removed, we used the **`snap saved`** command again, as shown in the preceding figure.

Snaps are versatile packages and easy to use. This package type is the choice of Ubuntu developers, but they are not commonly used on other distributions. If you would like to install snaps on

distributions other than Ubuntu, follow the instructions at https://snapcraft.io/docs/installing-snapd and test its full capabilities.

Now, we will test the other new kid on the block: flatpaks. Our test distribution will be Fedora, but keep in mind that flatpaks are also supported by Ubuntu-based distributions such as Linux Mint and elementary OS, and Debian-based distributions such as PureOS and Endless OS. A list of all the supported Linux distributions can be found at flatpak.org.

Managing flatpak packages on Fedora Linux

As flatpaks are available only as desktop applications, we will use Fedora Linux Workstation as our use case. In this scenario, you could use RHEL/AlmaLinux/Rocky Linux on a server, but Fedora for your workstation.

Similar to snaps, flatpaks are isolated applications that run inside sandboxes. Each flatpak contains the needed runtimes and libraries for the application. Flatpaks offer full support for graphical user interface management tools, together with a full set of commands that can be used from the **command-line interface** (**CLI**). The main command is `flatpak`, which has several other built-in commands to use for package management. To see all of them, use the following command:

```
$ flatpak --help
```

In the following sections, we will detail some of the widely used commands for flatpak package management. But before that, let's say a few words about how flatpak apps are named and how they will appear on the command line so that there will be no confusion in this regard.

Each app has an identifier in a form similar to `com.company.App`. Each part of this is meant to easily identify an app and its developer. The final part identifies the application's name since the preceding one identifies the entity that developed the app. This is an easy way for developers to publish and deliver multiple apps.

Adding flatpak repositories

Repositories must be set up if you wish to install applications. Flatpaks call repositories **remotes**, so this will be the term by which we will refer to them.

On our Fedora 37 machine, flatpak is already installed, but we will need to add the `flathub` repository. We will add it with the `flatpak remote-add` command, as shown in the following example:

```
$ sudo flatpak remote-add --if-not-exists flathub
https://dl.flathub.org/repo/flathub.flatpakrepo
```

Here, we used the `--if-not-exists` argument, which stops the command if the repository already exists, without showing any error. Once the repository has been added, we can start installing packages from it, but not before a mandatory system restart.

In Fedora 37 and previous versions, not all the apps from the Flathub repository are available by default, but starting with version 38, developers are aiming to provide all the apps from Flathub out of the box by default. Let's learn how to install an application from Flathub on our Fedora Workstation.

Installing a flatpak application

To install a package, we need to know its name. We can go to https://flathub.org/home and search for apps there. We will search for a piece of software called **Open Broadcaster Software** (**OBS**) studio on the website and follow the instructions provided. We can either click on the **Install** button in the top right-hand corner or use the commands from the lower half of the web page. We will use the following command:

```
$ sudo flatpak install flathub com.obsproject.Studio
```

On recent versions of flatpak (since version 1.2), installation can be performed with a much simpler command. In this case, you only need the name of the app, as follows:

```
$ sudo flatpak install slack
```

The result is the same as using the first `install` command shown previously.

Managing flatpak applications

After installing an application, you can run it using the command line with the following command:

```
$ flatpak run com.obsproject.Studio
```

If you want to update all the applications and runtimes, you can use this command:

```
$ sudo flatpak update
```

To remove a flatpak package, simply run the `flatpak uninstall` command:

```
$ sudo flatpak uninstall com.obsproject.Studio
```

To list all the flatpak applications and runtimes installed, you can use the `flatpak list` command:

```
[packt@fedora ~]$ flatpak list
Name             Application ID                  Version Branch      Installation
OBS Studio       com.obsproject.Studio           29.0.2  stable      system
Mesa             …desktop.Platform.GL.default 22.3.5  22.08       system
Mesa (Extra)     …desktop.Platform.GL.default 22.3.5  22.08-extra system
Adwaita theme    org.kde.KStyle.Adwaita                 6.4         system
KDE Application… org.kde.Platform                       6.4         system
[packt@fedora ~]$
```

Figure 3.27 – The flatpak list command's output

To see only the installed applications, you can use the `--app` argument:

```
$ flatpak list --app
```

The commands shown here are the most commonly used for flatpak package management. Needless to say, there are many other commands that we will not cover here, but you are free to look them up and test them on your system. For a quick overview of the basic flatpak commands, you can refer to the following link: https://docs.flatpak.org/en/latest/flatpak-command-reference.html.

Flatpaks are versatile and can provide access to newer app versions. Let's say you want to use a solid base operating system, but the downside of that is that you will get old versions of base applications by default. Using flatpaks can overcome this and give you access to newer app versions. Feel free to browse the apps available on Flathub and test the ones you find interesting and useful.

You now know how to install new applications on your operating system, using either the command line or the graphical user interface. Apart from that, you can also install new desktop environments. We will show you how in the following section.

Installing new desktop environments in Linux

We will continue to use Fedora as an example, but the commands shown here can also be used for any RHEL-based distribution, such as AlmaLinux or Rocky Linux.

By default, Fedora Workstation uses GNOME as the desktop environment, but what if you would like to use another one, such as KDE? Before showing you how, we would like to give you some information about the graphical desktop environments available for Linux.

Linux is all about choice, and this can't be more true when it comes to **desktop environments** (**DEs**). There are dozens of DEs available, such as GNOME, KDE, Xfce, LXDE, LXQT, Pantheon, and others. The most widely used DEs on Linux are GNOME, KDE, and Xfce, and the first two have the largest communities. If you want to use the very best and latest of GNOME, for example, you can try distributions such as Fedora, openSUSE Tumbleweed with GNOME, or Arch Linux (or Manjaro). If

you want to use the best of KDE, you can try KDE neon, openSUSE Tumbleweed with KDE, or Arch Linux with KDE (or Manjaro). For Xfce, you can try MX Linux (based on Debian), which defaults to Xfce, or openSUSE with Xfce. As a rule, the most widely used Linux distributions offer variants, also called *flavors* (in the case of Ubuntu) or *spins* (in the case of Fedora) with different desktop environments available. The RHEL and SUSE commercial versions come with GNOME only by default. For more information about the DEs described here, refer to the following websites:

- For KDE, visit www.kde.org
- For GNOME, visit www.gnome.org
- For Xfce, visit www.xfce.org

Now, let's learn how to install a different DE on our default Fedora Workstation.

Installing KDE Plasma on Fedora Linux

In Fedora and derivative distributions (and also in openSUSE), there are application groups available that ease the process of installing larger apps and their dependencies. And this becomes extremely useful when you're planning to install many apps as part of a larger *group*, just like a DE is.

To install a group, you can use the **`dnf install`** command and appeal the group using **`@`** and the name of the group. Alternatively, you can use the **`dnf groupinstall`** command while using the name of the group within quotes.

To check the groups that are available from the Fedora repositories, you can use the following command:

```
$ dnf group list --all | grep "KDE"
```

The output will be a list of groups from Fedora repos, and somewhere in there, the **KDE Plasma Workspaces** will be available. To install it, you can use the following command:

```
$ sudo dnf groupinstall "KDE Plasma Workspaces"
```

Alternatively, you can use the following command:

```
$ sudo dnf install @kde-desktop-environment
```

This command will install the new KDE Plasma environment, as shown in the following figure:

```
[packt@fedora ~]$ sudo dnf group list --all | grep "KDE"
[sudo] password for packt:
   KDE Plasma Workspaces
[packt@fedora ~]$ sudo dnf groupinstall "KDE Plasma Workspaces"
```

Figure 3.28 – Installing the KDE Plasma DE

The installation might take a while, depending on your internet connection. To start using KDE Plasma as your DE, you will need to log out of the active session. On the login screen, select your user, and then, in the bottom-right corner, click on the wheel icon and select **Plasma** when the option becomes available. You will have two options, one for **Wayland** and the other for the **X11** display manager:

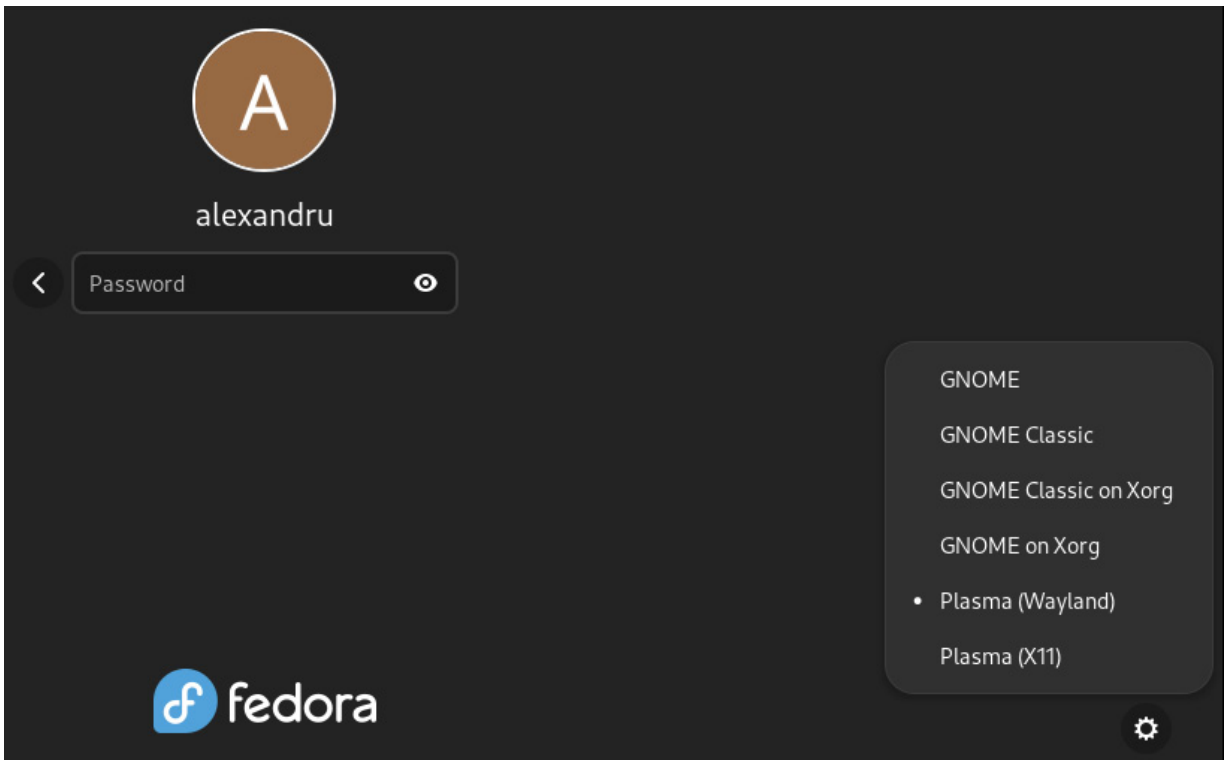

Figure 3.29 – Selecting Plasma (Wayland) on the login screen

Wayland is the newer option and might not have full support in KDE compared to the support it has in GNOME. Choose according to your preference.

Now, you can log into KDE Plasma on Fedora Workstation. The following screenshot shows the **Info Center** application inside KDE Plasma, with details about the installed version and hardware:

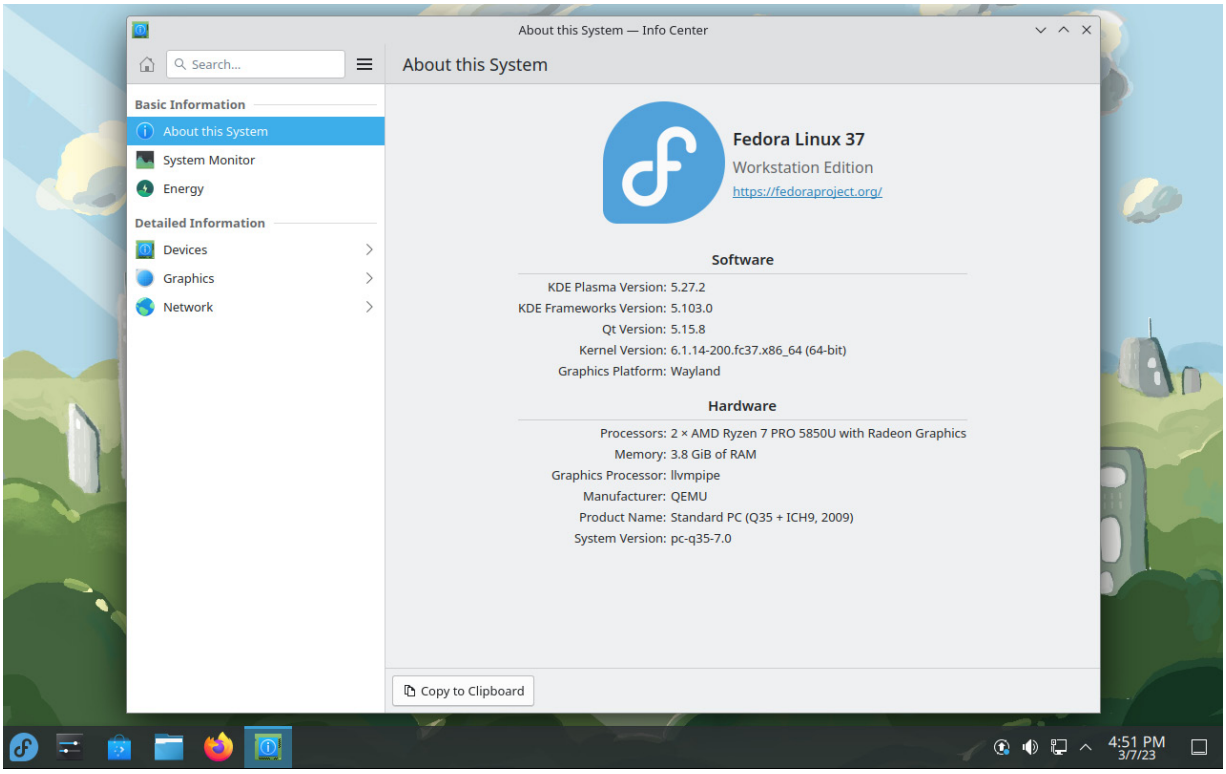

Figure 3.30 – Info Center in KDE Plasma on Fedora 37

With that, you have learned about working with packages in Linux, and even installing a new DE. This is sufficient for you to start fiddling around with your new OS. You can install new applications, configure them, and make your distribution the way you want it to be.

Summary

In this chapter, you learned how to work with packages in Ubuntu, Fedora/AlmaLinux, and openSUSE, and the skills you've learned will help you to manage packages in any Linux distribution. You learned how to work with both `.deb` and `.rpm` packages, and also the newer ones, such as flatpaks and snaps. You will use the skills you've learned here in every chapter of this book, as well as in your day-to-day job as a systems administrator – or even in your free time, enjoying your Linux operating system.

In the next chapter, we will show you how to manage user accounts and permissions, where you will be introduced to general concepts and specific tools.

Questions

Now that you have a clear idea of how to manage software packages, here are some exercises that will contribute further to your learning:

1. Make a list of all the packages installed on your system.

Hint: Consider using the `apt list --installed` command.

2. Add support for flatpaks on your Ubuntu system.

Hint: Follow the documentation at flatpak.org.

3. Test other distributions and use their package managers. We recommend that you try openSUSE and, if you feel confident, Arch Linux.

Further reading

For more information about what was covered in this chapter, please refer to the following resources:

- *Mastering Linux Administration – First Edition*, Alexandru Calcatinge, Julian Balog
- Snapcraft.io official documentation: https://snapcraft.io/docs
- Flatpak documentation: https://docs.flatpak.org/en/latest/
- openSUSE official documentation: https://doc.opensuse.org/

4

Managing Users and Groups

Linux is a multiuser, multitasking operating system, which means multiple users can access the operating system at the same time while sharing platform resources, with the kernel performing tasks for each user concurrently and independently. Linux provides the required isolation and security mechanisms to avoid multiple users accessing or deleting each other's files.

When multiple users are accessing the system, permissions come into play. We'll learn how **permissions** work in Linux, with their essential read, write, and execution tenets. We'll introduce you to the concept of a *superuser* (`root`) account, with complete access to the operating system resources.

Along the way, we'll take a hands-on approach to the topics learned, further deepening the assimilation of key concepts through practical examples. This chapter covers the following topics:

- Managing users
- Managing groups
- Managing permissions

We hope that by the end of the chapter, you will be comfortable with the command-line utilities for creating, modifying, and deleting users and groups, while proficiently handling file and directory permissions.

Let's take a quick look at the technical requirements for this chapter.

Technical requirements

You need a working Linux distribution installed on either a **virtual machine** (**VM**) or a desktop platform. In case you don't have one already, *Chapter 1*, *Installing Linux*, will drive you through the related process. In this chapter, we'll be using Ubuntu or Fedora, but most of the commands and examples used would pertain to any other Linux platform.

Managing users

In this context, a **user** is anyone using a computer or a system resource. In its simplest form, a Linux *user* or *user account* is identified by a name and a **unique identifier**, known as a **UID**.

From a purely technical point of view, in Linux, we have the following types of users:

- **Normal (or regular) users**: General-purpose, everyday user accounts, mostly suited for personal use and for common application and file management tasks, with limited access to system-wide resources. A regular user account usually has a *login* shell and a *home* directory.
- **System users**: These are similar to regular user accounts, except they may lack a login shell or a home directory. System accounts are usually assigned to background application services, mostly for security reasons and to limit the attack surface associated with the related resources—for example, a web server daemon handling public requests should run as a system account, ideally without login or **`root`** privileges. Consequently, possible vulnerabilities exposed through the web server would remain strictly isolated to the limited action realm of the associated system account.
- **Superusers**: These are privileged user accounts, with full access to system resources, including the permission to create, modify, and delete user accounts. The **`root`** user is an example of a superuser.

In Linux, only the **`root`** user or users with **`sudo`** privileges (**sudoers**) can create, modify, or delete user accounts.

Understanding sudo

The **`root`** user is the default superuser account in Linux, and it has the ability to do anything on a system. Ideally, acting as **`root`** on a system should generally be avoided due to safety and security reasons. With **`sudo`**, Linux provides a mechanism for *promoting* a regular user account to superuser privileges, using an additional layer of security. This way, a **`sudo`** user is generally used instead of **`root`**.

`sudo` is a command-line utility that allows a permitted user to execute commands with the security privileges of a superuser or another user (depending on the local system's security policy). **`sudo`** originally stood for *superuser do* due to its initial implementation of acting exclusively as the superuser, but has since been expanded to support not only the superuser but also other (restricted) user impersonations. Thus, it is also referred to as *substitute user do*. Yet, more often than not, it is perceived as *superuser do* due to its frequent use in Linux administrative tasks.

Most of the command-line tools for managing users in Linux require **`sudo`** privileges unless the related tasks are carried out by the **`root`** user. If we want to avoid using the root context, we can't genuinely proceed with the rest of this chapter—and create a user in particular—before we have a user account with superuser privileges. So, let's take this chicken-and-egg scenario out of the way first.

Most Linux distributions create an additional user account with superuser privileges, besides **`root`**, during installation. The reason, as noted before, is to provide an extra layer of security and safety for elevated operations. The simplest way to check whether a user account has **`sudo`** privileges is to run the following command in a terminal, while logged in with the related user account:

```
sudo -v
```

According to the **sudo** manual (**man sudo**), the **-v** option causes **sudo** to update the user's cached credentials and authenticate the user if the cached credentials expired.

If the user (for example, **julian**) doesn't have superuser privileges on the local machine (for example, **neptune**), the preceding command yields the following (or a similar) error:

```
Sorry, user julian may not run sudo on neptune.
```

In recent Linux distributions, the execution of a **sudo** command usually grants elevated permissions for a limited time. Ubuntu, for example, has a 15-minute **sudo** elevation span, after which time a **sudo** user would need to authenticate again. Subsequent invocations of **sudo** may not prompt for a password if done within the **sudo** cache credential timeout.

If we don't have a default superuser account, we can always use the root context to create new users (see the next chapter) and elevate them to **sudoer** privileges. We'll learn more about this in the *Creating a superuser* section, later in this chapter.

Now, let's have a look at how to create, modify, and delete users.

Creating, modifying, and deleting users

In this section, we explore a few command-line tools and some common tasks for managing users. The example commands and procedures are shown for Ubuntu and Fedora, but the same principles apply to any other Linux distribution. Some user management **command-line interface** (**CLI**) tools may differ or may not be available on specific Linux platforms (for example, **useradd** is not available on Alpine Linux, and **adduser** should be used instead). Please check the documentation of the Linux distribution of your choice for the equivalent commands.

Creating users

To create users, we can use either the **useradd** or the **adduser** command, although on some Linux distributions (for example, Debian or Ubuntu), the recommended way is to use the **adduser** command in favor of the low-level **useradd** utility. We'll cover both in this section.

adduser is a Perl script using **useradd**—basically, a shim of the **useradd** command—with a user-friendly guided configuration. Both command-line tools are installed by default in Ubuntu and Fedora. Let's take a brief look at each of these commands.

Creating users with useradd

The syntax for the **useradd** command is shown here:

```
useradd [OPTIONS] USER
```

In its simplest invocation, the following command creates a user account (`julian`):

```
sudo useradd julian
```

The user information is stored in a `/etc/passwd` file. Here's the related user data for `julian`:

```
sudo cat /etc/passwd | grep julian
```

In our case, this is the output:

```
packt@neptune:~$ sudo useradd julian
packt@neptune:~$ sudo cat /etc/passwd | grep julian
julian:x:1001:1001::/home/julian:/bin/sh
packt@neptune:~$
```

Figure 4.1 – The user record created with useradd

Let's analyze the related user record. Each entry is delimited by a colon (`:`) and is listed here:

- `julian`: Username
- `x`: Encrypted password (password hash is stored in `/etc/shadow`)
- `1001`: The UID
- `1001`: The user **group ID** (**GID**)
- `::` The **General Electric Comprehensive Operating Supervisor** (**GECOS**) field—for example, display name (in our case, empty), explained later in this section
- `/home/julian`: User home folder
- `/bin/sh`: Default login shell for the user

IMPORTANT NOTE

The GECOS field is a string of comma-delimited attributes, reflecting general information about the user account (for example, real name, company, and phone number). In Linux, the GECOS field is the fifth field in a user record. See more information at https://en.wikipedia.org/wiki/Gecos_field.

We can also use the `getent` command to retrieve the preceding user information, as follows:

```
getent passwd julian
```

To view the UID (`uid`), GID (`gid`), and group membership associated with a user, we can use the `id` command, as follows:

```
id julian
```

This command gives us the following output:

```
packt@neptune:~$ sudo useradd julian
packt@neptune:~$ sudo cat /etc/passwd | grep julian
julian:x:1001:1001::/home/julian:/bin/sh
packt@neptune:~$ getent passwd julian
julian:x:1001:1001::/home/julian:/bin/sh
packt@neptune:~$ id julian
uid=1001(julian) gid=1001(julian) groups=1001(julian)
packt@neptune:~$
```

Figure 4.2 – The UID information

With the simple invocation of **useradd**, the command creates the user (**julian**) with some immediate default values (as enumerated), while other user-related data is empty—for example, we have no full name or password specified for the user yet. Also, while the home directory has a default value (for example, **/home/julian**), the actual filesystem folder will not be created unless the **useradd** command is invoked with the **-m** or the **--create-home** option, as follows:

```
sudo useradd -m julian
```

Without a home directory, regular users would not have the ability to save their files in a private location on the system. On the other hand, some system accounts may not need a home directory since they don't have a login shell. For example, a database server (for example, PostgreSQL) may run with a non-root system account (for example, **postgres**) that only needs access to database resources in specific locations (for example, **/var/lib/pgsql**), controlled via other permission mechanisms (for example, **Security-Enhanced Linux** (**SELinux**)).

For our regular user, if we also wanted to specify a full name (display name), the command would change to this:

```
sudo useradd -m -c "Julian" julian
```

The **-c, --comment** option parameter of **useradd** expects a *comment*, also known as the GECOS field (the fifth field in our user record), with multiple comma-separated values. In our case, we specify the full name (for example, **Julian**). For more information, check out the **useradd** manual (**man useradd**) or **useradd --help**.

The user still won't have a password yet, and consequently, there would be no way for them to log in (for example, via a **graphical user interface** (**GUI**) or **Secure Shell** (**SSH**)). To create a password for **julian**, we invoke the **passwd** command, like this:

```
sudo passwd julian
```

You can see the following output:

```
packt@neptune:~$ sudo passwd julian
New password:
Retype new password:
passwd: password updated successfully
packt@neptune:~$
```

Figure 4.3 – Creating or changing the user password

The **passwd** command will prompt for the new user's password. With the password set, there will be a new entry added to the **/etc/shadow** file. This file stores the secure password hashes (not the passwords!) for each user. Only superusers can access the content of this file. Here's the command to retrieve the related information for the user **julian**:

```
sudo getent shadow julian
```

You can also use the following command:

```
sudo cat /etc/shadow | grep julian
```

The output of both commands is shown in the following screenshot:

```
packt@neptune:~$ sudo getent shadow julian
julian:$y$j9T$mI5gG19NBCTDj1B2PVAr8/$/sK8PINvEm1KLYk1QnE1w/05s4tJd.WtOUVOZpmaJLB
:19424:0:99999:7:::
packt@neptune:~$ sudo cat /etc/shadow | grep julian
julian:$y$j9T$mI5gG19NBCTDj1B2PVAr8/$/sK8PINvEm1KLYk1QnE1w/05s4tJd.WtOUVOZpmaJLB
:19424:0:99999:7:::
packt@neptune:~$
```

Figure 4.4 – Information about the user from the shadow file

Once the password has been set, in normal circumstances, the user can log in to the system (via SSH or GUI). If the Linux distribution has a GUI, the new user will show up on the login screen.

As noted, with the **useradd** command, we have low-level granular control over how we create user accounts, but sometimes we may prefer a more user-friendly approach. Enter the **adduser** command.

Creating users with adduser

The **adduser** command is a Perl wrapper for **useradd**. The syntax for this command is shown here:

```
adduser [OPTIONS] USER
```

sudo may prompt for the superuser password. **adduser** will prompt for the new user's password and other user-related information (as shown in *Figure 4.5*).

Let's create a new user account (**alex**) with **adduser**, as follows:

```
sudo adduser alex
```

The preceding command yields the following output:

```
packt@neptune:~$ sudo adduser alex
Adding user `alex' ...
Adding new group `alex' (1002) ...
Adding new user `alex' (1002) with group `alex' ...
Creating home directory `/home/alex' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for alex
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n]
```

Figure 4.5 – The adduser command

In Fedora, the preceding invocation of the **`adduser`** command will simply run without prompting the user for a password or any other information.

We can see the related user entry in **`/etc/passwd`** with **`getent`**, as follows:

```
getent passwd alex
```

The following is the output:

```
packt@neptune:~$ getent passwd alex
alex:x:1002:1002:,,,:/home/alex:/bin/bash
packt@neptune:~$
```

Figure 4.6 – Viewing user information with getent

In the preceding examples, we created a regular user account. Administrators or superusers can also elevate the privileges of a regular user to a superuser. Let's see how in the following section.

Creating a superuser

When a regular user is given the power to run **`sudo`**, they become a superuser. Let's assume we have a regular user created via any of the examples shown in the *Creating users* section.

Promoting the user to a superuser (or *sudoer*) requires a **`sudo`** group membership. In Linux, the **`sudo`** group is a reserved system group for users with elevated or **`root`** privileges. To make the user **`julian`**

a sudoer, we simply need to add the user to the **sudo** group, like this (in Ubuntu):

```
sudo usermod -aG sudo julian
```

The **-aG** options of **usermod** instruct the command to append (**-a, --append**) the user to the specified group (**-G, --group**)—in our case, **sudo**.

To verify our user is now a sudoer, first make sure the related user information reflects the **sudo** membership by running the following command:

```
id julian
```

This gives us the following output:

```
packt@neptune:~$ sudo usermod -aG sudo julian
packt@neptune:~$ id julian
uid=1001(julian) gid=1001(julian) groups=1001(julian),27(sudo)
packt@neptune:~$
```

Figure 4.7 – Looking for the sudo membership of a user

The output shows that the **sudo** group membership (GID) in the **groups** tag is **27(sudo)**.

To verify the **sudo** access for the user **julian**, run the following command:

```
su - julian
```

The preceding command prompts for the password of the user **julian**. A successful login would usually validate the superuser context. Alternatively, the user (**julian**) can run the **sudo -v** command in their terminal session to validate the **sudo** privileges. For more information on superuser privileges, see the *Understanding sudo* section earlier in the chapter.

With multiple users created, a system administrator may want to view or list all the users in the system. In the next section, we provide a few ways to accomplish this task.

Viewing users

There are a few ways for a superuser to view all users configured in the system. As previously noted, the user information is stored in the **/etc/passwd** and **/etc/shadow** files. Besides simply viewing these files, we can parse them and extract only the usernames with the following command:

```
cat /etc/passwd | cut -d: -f1 | less
```

Alternatively, we can parse the **/etc/shadow** file, like this:

```
sudo cat /etc/shadow | cut -d: -f1 | less
```

In the preceding commands, we read the content from the related files (with `cat`). Next, we piped the result to a delimiter-based parsing (with `cut`, on the `:` delimiter) and picked the first field (`-f1`). Finally, we chose a paginated display of the results, using the `less` command (to exit the command's output, press *Q*).

Note the use of `sudo` for the `shadow` file since access is limited to superusers only, due to the sensitive nature of the password hash data. Alternatively, we can use the `getent` command to retrieve the user information.

The following command lists all the users configured in the system:

```
getent passwd
```

The preceding command reads the `/etc/passwd` file. Alternatively, we can retrieve the same information from `/etc/shadow`, as follows:

```
sudo getent shadow
```

For both commands, we can further pipe the `getent` output to `| cut -d: -f1` to list only the usernames, like this:

```
sudo getent shadow | cut -d: -f1 | less | column
```

The output will be similar to this:

```
packt@neptune:~$ cat /etc/passwd | cut -d: -f1 | less
packt@neptune:~$ sudo cat /etc/shadow | cut -d: -f1 | less
packt@neptune:~$ sudo getent shadow | cut -d: -f1 | less | column
root                    www-data                sshd
daemon                  backup                  syslog
bin                     list                    uuidd
sys                     irc                     tcpdump
sync                    gnats                   tss
games                   nobody                  landscape
man                     _apt                    usbmux
lp                      systemd-network         packt
mail                    systemd-resolve         lxd
news                    messagebus              fwupd-refresh
uucp                    systemd-timesync        julian
proxy                   pollinate               alex
packt@neptune:~$
```

Figure 4.8 – Viewing usernames

With new users created, administrators or superusers may want to change certain user-related information, such as password, password expiration, full name, or login shell. Next, we take a look at some of the most common ways to accomplish this task.

Modifying users

A superuser can run the **usermod** command to modify user settings, with the following syntax:

```
usermod [OPTIONS] USER
```

The examples in this section apply to a user we previously created (**julian**) with the simplest invocation of the **useradd** command. As noted in the previous section, the related user record in **/etc/passwd** has no full name for the user, and the user has no password either.

Let's change the following settings for our user (**julian**):

- **Full name**: To **Julian** (initially empty)
- **Home folder**: Move to **/local/julian** (from default **/home/julian**)
- **Login shell**: **/bin/bash** (from default **/bin/sh**)

The command-line utility for changing all the preceding information is shown here:

```
sudo usermod -c "Julian" -d /local/julian -m -s /bin/bash julian
```

Here are the command options, briefly explained:

- **-c, --comment "Julian"**: The full username
- **-d, --home local/julian**: The user's new home directory
- **-m, --move**: Move the content of the current home directory to the new location
- **-s, --shell /bin/sh**: The user login shell

The related change, retrieved with the **getent** command, is shown here:

```
getent passwd julian
```

We get the following output:

```
packt@neptune:~$ sudo usermod -c "Julian" -d /local/julian -m -s /bin/bash julia
n
packt@neptune:~$ getent passwd julian
julian:x:1001:1001:Julian:/local/julian:/bin/bash
packt@neptune:~$
```

Figure 4.9 – The user changes reflected with getent

Here are a few more examples of changing user settings with the **usermod** command-line utility.

Changing the username

The **-l, --login** option parameter of **usermod** specifies a new login username. The following command changes the username from **julian** to **balog** (that is, first name to last name), as illustrated

here:

```
sudo usermod -l "balog" julian
```

In a production environment, we may have to add to the preceding command, as we may also want to change the display name and the home directory of the user (for consistency reasons). In a previous example in the *Creating users with useradd* section, we showcased the **`-d, --home`** and **`-m, --move`** option parameters, which would accommodate such changes.

Locking or unlocking a user

A superuser or administrator may choose to temporarily or permanently lock a specific user with the **`-L, --lock`** option parameter of **`usermod`**, as follows:

```
sudo usermod -L julian
```

As a result of the preceding command, the login attempt for the user **`julian`** would be denied. Should the user try to SSH into the Linux machine, they would get a **Permission denied, please try again** error message. Also, the related username will be removed from the login screen if the Linux platform has a GUI.

To unlock the user, we invoke the **`-U, --unlock`** option parameter, as follows:

```
sudo usermod -U julian
```

The preceding command restores system access for the user.

For more information on the **`usermod`** utility, please check out the related documentation (**`man usermod`**) or the command-line help (**`usermod --help`**).

Although the recommended way of modifying user settings is via the **`usermod`** command-line utility, some users may find it easier to manually edit the **`/etc/passwd`** file. The following section shows you how.

Modifying users via /etc/passwd

A superuser can also manually edit the **`/etc/passwd`** file to modify user data by updating the relevant line. Although the editing can be done with a text editor of your choice (for example, **`nano`**), we recommend the use of the **`vipw`** command-line utility for a safer approach. **`vipw`** enables the required locks to prevent possible data corruption—for example, in case a superuser performs a change at the same time regular users change their password.

The following command initiates the editing of the **`/etc/passwd`** file by also prompting for the preferred text editor (for example, **`nano`** or **`vim`**):

```
sudo vipw
```

For example, we can change the settings for user **julian** by editing the following line:

```
julian:x:1001:1001:Julian,,,:/home/julian:/bin/bash
```

The meaning of the colon (`:`)-separated fields was previously described in the *Creating users with useradd* section. Each of these fields can be manually altered in the **/etc/passwd** file, resulting in changes equivalent to the corresponding **usermod** invocation.

For more information on the **vipw** command-line utility, you can refer to the related system manual (**man vipw**).

Another relatively common administrative task for a user account is to change a password or set up a password expiration. Although **usermod** can change a user password via the **-p** or **--password** option, it requires an encrypted hash string (and not a cleartext password). Generating an encrypted password hash would be an extra step. An easier way is to use the **passwd** utility to change the password.

A superuser (administrator) can change the password of a user (for example, the user **julian**) with the following command:

```
sudo passwd julian
```

The output will ask for the new password for the respective user. To change the expiration time of a password (the password age), the **chage** command is used. For example, to set a 30-day password age for the user **julian**, we will use the following command:

```
sudo chage -M 30 julian
```

This will force the user **julian** to change their password every month. The password time availability is defined system-wide by the password policy. It is found inside the **/etc/login.defs** file, inside the **Password aging controls** section. If you change these entries, the password policy will be set for every user. Another common practice is to force the user to change the password at the first login. You can enforce this by using the following command (we will use **julian** as our example again):

```
sudo chage -d 0 julian
```

This command will force the user **julian** to enter their own password the first time they log in to the system.

Sometimes, administrators are required to remove specific users from the system. The next section shows a couple of ways of accomplishing this task.

Deleting users

The most common way to remove users from the system is to use the **`userdel`** command-line tool. The general syntax of the **`userdel`** command is shown here:

```
userdel [OPTIONS] USER
```

For example, to remove the user **`julian`**, a superuser would run the following command:

```
sudo userdel -f -r julian
```

Here are the command options used:

- **`-f, --force`**: Removes all files in the user's home directory, even if not owned by the user
- **`-r, --remove`**: Removes the user's home directory and mail spool

The **`userdel`** command removes the related user data from the system, including the user's home directory (when invoked with the **`-f`** or **`--force`** option) and the related entries in the **`/etc/passwd`** and **`/etc/shadow`** files.

There is also an alternative way, which could be handy in some odd cleanup scenarios. The next section shows how.

Deleting users via /etc/passwd and /etc/shadow

A superuser can edit the **`/etc/passwd`** and **`/etc/shadow`** files and manually remove the corresponding lines for the user (for example, **`julian`**). Please note that both files have to be edited for consistency and complete removal of the related user account.

Edit the **`/etc/passwd`** file using the **`vipw`** command-line utility, as follows:

```
sudo vipw
```

Remove the following line (for the user **`julian`**):

```
julian:x:1001:1001:Julian,,,:/home/julian:/bin/bash
```

Next, edit the **`/etc/shadow`** file using the **`-s`** or **`--shadow`** option with **`vipw`**, as follows:

```
sudo vipw -s
```

Remove the following line (for the user **`julian`**):

```
julian:$6$xDdd7Eay/RKYjeTm$Sf.../:18519:0:99999:7:::
```

After editing the preceding files, a superuser may also need to remove the deleted user's home directory, as follows:

```
sudo rm -rf /home/julian
```

For more information on the **`userdel`** utility, please check out the related documentation (**`man userdel`**) or the command-line help (**`userdel --help`**).

The user management concepts and commands learned so far apply exclusively to individual users in the system. When multiple users in the system have a common access level or permission attribute, they are collectively referred to as a group. Groups can be regarded as standalone organizational units we can create, modify, or delete. We can also define and alter user memberships associated with groups. The next section focuses on group management internals.

Managing groups

Linux uses groups to organize users. Simply put, a group is a collection of users sharing a common attribute. Examples of such groups could be *employees*, *developers*, *managers*, and so on. In Linux, a group is uniquely identified by a GID. Users within the same group share the same GID.

From a user's perspective, there are two types of groups, outlined here:

- **Primary group**: The user's initial (default) login group
- **Supplementary groups**: A list of groups the user is also a member of; also known as **secondary groups**

Every Linux user is a member of a primary group. A user can belong to multiple supplementary groups or no supplementary groups at all. In other words, there is one mandatory primary group associated with each Linux user, and a user can have multiple or no supplementary group memberships.

From a practical point of view, we can look at groups as a permissive context of collaboration for a select number of users. Imagine a *developers* group having access to developer-specific resources. Each user in this group has access to these resources. Users outside the *developers* group may not have access unless they authenticate with a group password if the group has one.

In the following section, we provide detailed examples of how to manage groups and set up group memberships for users. Most related commands require *superuser* or **`sudo`** privileges.

Creating, modifying, and deleting groups

While our primary focus remains on group administrative tasks, some related operations still involve user-related commands. Command-line utilities such as **`groupadd`**, **`groupmod`**, and **`groupdel`** are targeted strictly at creating, modifying, and deleting groups, respectively. On the other hand, the **`useradd`** and **`usermod`** commands carry group-specific options when associating users with groups.

We'll also introduce you to **gpasswd**, a command-line tool specializing in group administration, combining user- and group-related operations.

With this aspect in mind, let's take a look at how to create, modify, and delete groups and how to manipulate group memberships for users.

Creating groups

To create a new group, a superuser invokes the **groupadd** command-line utility. Here's the basic syntax of the related command:

```
groupadd [OPTIONS] GROUP
```

Let's create a new group (**developers**), with default settings, as follows:

```
sudo groupadd developers
```

The group information is stored in the **/etc/group** file. Here's the related data for the **developers** group:

```
cat /etc/group | grep developers
```

The command yields the following output:

```
packt@neptune:~$ sudo groupadd developers
[sudo] password for packt:
packt@neptune:~$ cat /etc/group | grep developers
developers:x:1003:
packt@neptune:~$
```

Figure 4.10 – The group with default attributes

Let's analyze the related group record. Each entry is delimited by a colon (`:`) and is listed here:

- **developers**: Group name
- **x**: Encrypted password (password hash is stored in **/etc/gshadow**)
- **1003**: GID

We can also use the **getent** command to retrieve the preceding group information, as follows:

```
getent group developers
```

A superuser may choose to create a group with a specific GID, using the **-g, --gid** option parameter with **groupadd**. For example, the following command creates the **developers** group (if it doesn't exist) with a GID of **1200**:

```
sudo groupadd -g 1200 developers
```

For more information on the **groupadd** command-line utility, please refer to the related documentation (**man groupadd**).

Group-related data is stored in the **/etc/group** and **/etc/gshadow** files. The **/etc/group** file contains generic group membership information, while the **/etc/gshadow** file stores the encrypted password hashes for each group. Let's take a brief look at group passwords.

Understanding group passwords

By default, a group doesn't have a password when created with the simplest invocation of the **groupadd** command (for example, **groupadd developers**). Although **groupadd** supports an encrypted password (via the **-p, --password** option parameter), this would require an extra step to generate a secure password hash. There's a better and simpler way to create a group password: by using the **gpasswd** command-line utility.

IMPORTANT NOTE

***gpasswd** is a command-line tool that helps with everyday group administration tasks.*

The following command creates a password for the **developers** group:

```
sudo gpasswd developers
```

We get prompted to enter and re-enter a password, as illustrated here:

```
packt@neptune:~$ sudo gpasswd developers
Changing the password for group developers
New Password:
Re-enter new password:
```

Figure 4.11 – Creating a password for the developers group

The purpose of a group password is to protect access to group resources. A group password is inherently insecure when shared among group members, yet a Linux administrator may choose to keep the group password private while group members collaborate unhindered within the group's security context.

Here's a quick explanation of how it works. When a member of a specific group (for example, **developers**) logs in to that group (using the **newgrp** command), the user is not prompted for the group password. When users who don't belong to the group attempt to log in, they will be prompted for the group password.

In general, a group can have administrators, members, and a password. Members of a group who are the group's administrators may use **gpasswd** without being prompted for a password, as long as

they're logged in to the group. Also, group administrators don't need superuser privileges to perform group administrative tasks for a group they are the administrator of.

We'll take a closer look at **gpasswd** in the next sections, where we further focus on group management tasks, as well as adding users to a group and removing users from a group. But for now, let's keep our attention strictly at the group level and see how we can modify a user group.

Modifying groups

The most common way to modify the definition of a group is via the **groupmod** command-line utility. Here's the basic syntax for the command:

```
groupmod [OPTIONS] GROUP
```

The most common operations when changing a group's definition are related to the GID, group name, and group password. Let's take a look at each of these changes. We assume our previously created group is named **developers**, with a GID of **1003**.

To change the GID to **1200**, a superuser invokes the **groupmod** command with the **-g, --gid** option parameter, as follows:

```
sudo groupmod -g 1200 developers
```

To change the group name from **developers** to **devops**, we invoke the **-n, --new-name** option, like this:

```
sudo groupmod -n devops developers
```

We can verify the preceding changes for the **devops** group with the following command:

```
getent group devops
```

The command yields the following output:

```
packt@neptune:~$ getent group developers
developers:x:1003:
packt@neptune:~$ sudo groupmod -g 1200 developers
packt@neptune:~$ getent group developers
developers:x:1200:
packt@neptune:~$ sudo groupmod -n devops developers
packt@neptune:~$ getent group devops
devops:x:1200:
```

Figure 4.12 – Verifying the group changes

To change the group password for **devops**, the simplest way is to use **gpasswd**, as follows:

```
sudo gpasswd devops
```

We are prompted to enter and re-enter a password.

To remove the group password for **devops**, we invoke the **gpasswd** command with the **-r, --remove-password** option, as follows:

```
sudo gpasswd -r devops
```

As the command has no visible outcome or message, we will be prompted back to the shell:

```
packt@neptune:~$ sudo gpasswd devops
Changing the password for group devops
New Password:
Re-enter new password:
packt@neptune:~$ sudo gpasswd -r devops
packt@neptune:~$
```

Figure 4.13 – Setting a new group password and removing a group password

For more information on **groupmod** and **gpasswd**, refer to the system manuals of these utilities (**man groupmod** and **man gpasswd**), or simply invoke the **-h, --help** option for each.

Next, we look at how to delete groups.

Deleting groups

To delete groups, we use the **groupdel** command-line utility. The related syntax is shown here:

```
groupdel [OPTIONS] GROUP
```

By default, Linux enforces referential integrity between a primary group and the users associated with that primary group. We cannot delete a group that has been assigned as a primary group for some users before deleting the users of that primary group. In other words, by default, Linux doesn't want to leave the users with dangling primary GIDs.

For example, when we first added the user **julian**, they were assigned automatically to the **julian** primary group. We then added the user to the **sudoers** group.

Let's attempt to add the user **julian** to the **devops** group. A superuser may run the **usermod** command with the **-g, --gid** option parameter to *change* the primary group of a user. The command should be invoked for each user. Here's an example of removing the user **julian** from the **julian** primary group. First, let's get the current data for the user, as follows:

```
id julian
```

This is the output:

```
packt@neptune:~$ id julian
uid=1003(julian) gid=1003(julian) groups=1003(julian),27(sudo)
```

Figure 4.14 – Retrieving the current primary group for the user

Now, let us add the user **julian** to the **devops** group. The **-g, --gid** option parameter of the **usermod** command accepts both a *GID* and a group *name*. The specified group name must already be present in the system; otherwise, the command will fail. If we want to change the primary group (for example, to **devops**), we simply specify the group name in the **-g, --gid** option parameter, as follows:

```
sudo usermod -g devops julian
```

The output is shown in the following screenshot:

```
packt@neptune:~$ id julian
uid=1003(julian) gid=1003(julian) groups=1003(julian),27(sudo)
packt@neptune:~$ sudo usermod -g devops julian
packt@neptune:~$ id julian
uid=1003(julian) gid=1200(devops) groups=1200(devops),27(sudo)
packt@neptune:~$
```

Figure 4.15 – Changing the primary group of the user

The result is that the user **julian** is now part of the **devops** group.

Now, let us attempt to delete the **devops** group, which is the primary group for the user **julian**. Attempting to delete the **devops** group results in an error, as can be seen in *Figure 4.16* (the first command used). Therefore, we cannot delete a group that is not empty.

A superuser may choose to *force* the deletion of a primary group, invoking **groupdel** with the **-f, --force** option, but this would be ill advised. This is because the command would result in users with orphaned primary GIDs and a possible security hole in the system. The maintenance and removal of such users would also become problematic.

In order to be able to delete the **devops** group, we need to assign another group to the user **julian**. What we can do is assign it to the initial primary group called **julian**, and then attempt to delete the **devops** group, now that it is empty. First, let us assign the user **julian** to the **julian** group with the following command:

```
sudo usermod -g julian julian
```

At this point, it's safe to delete the group (**devops**), as follows:

```
sudo groupdel devops
```

The outcome from the preceding commands is this:

```
packt@neptune:~$ groupdel devops
groupdel: cannot remove the primary group of user 'julian'
packt@neptune:~$ sudo usermod -g julian julian
[sudo] password for packt:
packt@neptune:~$ id julian
uid=1003(julian) gid=1003(julian) groups=1003(julian),27(sudo)
packt@neptune:~$ sudo groupdel devops
packt@neptune:~$ getent group devops
packt@neptune:~$
```

Figure 4.16 – Successful attempt to delete the group

For more information on the **groupdel** command-line utility, check out the related system manual (**man groupdel**), or simply invoke **groupdel --help**.

Modifying groups via /etc/group

An administrator can also manually edit the **/etc/group** file to modify group data by updating the related line. Although the editing can be done with a text editor of your choice (for example, **nano**), we recommend the use of the **vigr** command-line utility for a safer approach. **vigr** is similar to **vipr** (for modifying **/etc/passwd**) and sets safety locks to prevent possible data corruption during concurrent changes of group data.

The following command opens the **/etc/group** file for editing by also prompting for the preferred text editor (for example, **nano** or **vim**):

```
sudo vigr
```

For example, we can change the settings for the **developers** group by editing the following line:

```
developers:x:1200:julian,alex
```

When deleting groups using the **vigr** command, we're also prompted to remove the corresponding entry in the group shadow file **(/etc/gshadow**). The related command invokes the **-s** or **--shadow** option, as illustrated here:

```
sudo vigr -s
```

For more information on the **vigr** utility, please refer to the related system manual (**man vigr**).

As with most Linux tasks, all the preceding tasks could have been accomplished in different ways. The commands chosen are the most common ones, but there might be cases when a different approach may prove more appropriate.

In the next section, we'll take a glance at how to add users to primary and secondary groups and how to remove users from these groups.

Managing users in groups

So far, we've only created groups that have no users associated. There is not much use for empty user groups, so let's add some users to them.

Adding users to a group

Before we start adding users to a group, let's create a few groups. In the following example, we create the groups by also specifying their GID (via the **`-g, --gid`** option parameter of the **`groupadd`** command):

```
sudo groupadd -g 1100 admin
sudo groupadd -g 1200 developers
sudo groupadd -g 1300 devops
```

We can check the last groups created by using the following command:

```
cat /etc/group | tail -n 5
```

It will show us the last five lines of the **`/etc/group`** file. We can see the last five groups created.

Next, we create a couple of new users (**`alex2`** and **`julian2`** as we already have users **`alex`** and **`julian`**) and add them to some of the groups we just created. We'll have the **`admin`** group set as the *primary group* for both users, while the **`developers`** and **`devops`** groups are defined as *secondary* (or *supplementary*) *groups*. The code can be seen here:

```
sudo useradd -g admin -G developers,devops alex2
sudo useradd -g admin -G developers,devops julian2
```

The **`-g, --gid`** option parameter of the **`useradd`** command specifies the (unique) primary group (**`admin`**). The **`-G, --groups`** option parameter provides a comma-separated list (without intervening spaces) of the secondary group names (**`developers,devops`**).

We can verify the group memberships for both users with the following commands:

```
id alex2
id julian2
```

The output is shown in the following screenshot:

```
packt@neptune:~$ cat /etc/group | tail -n 5
alex:x:1002:
julian:x:1003:
admin:x:1100:
developers:x:1200:
devops:x:1300:
packt@neptune:~$ sudo useradd -g admin -G developers,devops alex2
packt@neptune:~$ sudo useradd -g admin -G developers,devops julian2
packt@neptune:~$ id alex2
uid=1004(alex2) gid=1100(admin) groups=1100(admin),1200(developers),1300(devops)
packt@neptune:~$ id julian2
uid=1005(julian2) gid=1100(admin) groups=1100(admin),1200(developers),1300(devop
s)
packt@neptune:~$
```

Figure 4.17 – Assigning groups to new users

As we can see, the **gid** attribute shows the primary group membership: **gid=1100(admin)**. The **groups** attribute shows the supplementary (secondary) groups: **groups=1100(admin),1200(developers),1300(devops)**.

With users scattered across multiple groups, an administrator is sometimes confronted with the task of moving users between groups. The following section shows how to do this.

Moving and removing users across groups

Building upon the previous example, let's assume the administrator wants to move (or add) the user **alex2** to a new secondary group called **managers**. Please note that, according to our previous examples, the user **alex2** has **admin** as the primary group and **developers**/**devops** as secondary groups (see the output of the **id alex2** command in *Figure 4.18*).

Let's create a **managers** group first, with GID **1400**. The code can be seen here:

```
sudo groupadd -g 1400 managers
```

Next, add our existing user, **alex2**, to the **managers** group. We use the **usermod** command with the **-G, --groups** option parameter to specify the secondary groups the user is associated with.

The simplest way to *append* a secondary group to a user is by invocation of the **-a, --append** option of the **usermod** command, as illustrated here:

```
sudo usermod -a -G managers alex2
```

The preceding command would preserve the existing secondary groups for the user **alex2** while adding the new **managers** group. Alternatively, we could run the following command:

```
sudo usermod -G developers,devops,managers alex2
```

In the preceding command, we specified multiple groups (with no intervening whitespace!).

IMPORTANT NOTE

*We preserved the existing secondary groups (**developers/devops**) and appended to the comma-separated list the **managers** additional secondary group. If we only had the **managers** group specified, the user **alex2** would have been removed from the **developers** and **devops** secondary groups.*

To verify whether the user **alex2** is now part of the **managers** group, run the following command:

```
id alex2
```

This is the output of the command:

```
uid=1004(alex2) gid=1100(admin)
groups=1100(admin),1200(developers),1300(devops),1400(managers)
```

As we can see, the **groups** attribute (highlighted) includes the related entry for the **managers** group: **1400(managers)**.

Similarly, if we wanted to *remove* the user **alex2** from the **developers** and **devops** secondary groups, to only be associated with the **managers** secondary group, we would run the following command:

```
sudo usermod -G managers alex2
```

This is the output:

```
packt@neptune:~$ sudo groupadd -g 1400 managers
packt@neptune:~$ sudo usermod -a -G managers alex2
packt@neptune:~$ id alex2
uid=1004(alex2) gid=1100(admin) groups=1100(admin),1200(developers),1300(devops)
,1400(managers)
packt@neptune:~$ sudo usermod -G managers alex2
packt@neptune:~$ id alex2
uid=1004(alex2) gid=1100(admin) groups=1100(admin),1400(managers)
packt@neptune:~$
```

Figure 4.18 – Verifying the secondary groups for the user

The **groups** tag now shows the primary group **admin** (by default) and the **managers** secondary group.

The command to remove the user **alex2** from all secondary groups is shown here:

```
sudo usermod -G '' alex2
```

The **usermod** command has an empty string (**''**) as the **-G, --groups** option parameter, to ensure no secondary groups are associated with the user. We can verify that the user **alex2** has no more secondary group memberships with the following command:

```
id alex
```

This is the output:

```
packt@neptune:~$ id alex2
uid=1004(alex2) gid=1100(admin) groups=1100(admin),1400(managers)
packt@neptune:~$ sudo usermod -G '' alex2
packt@neptune:~$ id alex2
uid=1004(alex2) gid=1100(admin) groups=1100(admin)
packt@neptune:~$
```

Figure 4.19 – Verifying the user has no secondary groups

As we can see, the **groups** tag only contains the **1100(admin)** primary GID, which by default is always shown for a user.

If an administrator chooses to remove the user **alex2** from a primary group or assign them to a different primary group, they must run the **usermod** command with the **-g, --gid** option parameter and specify the primary group name. A primary group is always mandatory for a user, and it must exist.

For example, to move the user **alex2** to the **managers** primary group, the administrator would run the following command:

```
sudo usermod -g managers alex2
```

The related user data can be obtained using the following command:

```
id alex2
```

The command yields the following output:

```
packt@neptune:~$ id alex2
uid=1004(alex2) gid=1100(admin) groups=1100(admin)
packt@neptune:~$ sudo usermod -g managers alex2
packt@neptune:~$ id alex2
uid=1004(alex2) gid=1400(managers) groups=1400(managers)
packt@neptune:~$
```

Figure 4.20 – Verifying the user has been assigned to the new primary group

The **gid** attribute of the user record in *Figure 4.21* reflects the new primary group: **gid=1400(managers)**.

If the administrator chooses to configure the user **alex2** without a specific primary group, they must first create an exclusive *group* (named **alex2**, for convenience), and have the GID matching the UID of the user **alex2** (**1004**), as follows:

```
sudo groupadd -g 1004 alex2
```

And now, we can remove the user **alex2** from the current primary group (**managers**) by specifying the exclusive primary group we just created (**alex2**), like this:

```
sudo usermod -g alex2 alex2
```

The related user record becomes this:

```
id alex2
```

This is the output:

```
packt@neptune:~$ id alex2
uid=1004(alex2) gid=1400(managers) groups=1400(managers)
packt@neptune:~$ sudo groupadd -g 1004 alex2
groupadd: group 'alex2' already exists
packt@neptune:~$ getent group | tail -n 2
managers:x:1400:
alex2:x:1004:
packt@neptune:~$ sudo usermod -g alex2 alex2
packt@neptune:~$ id alex2
uid=1004(alex2) gid=1004(alex2) groups=1004(alex2)
packt@neptune:~$
```

Figure 4.21 – Verifying the user has been removed from primary groups

The **gid** attribute of the user record reflects the exclusive primary group (matching the user): **gid=1004(alex2)**. Our user doesn't belong to any other primary groups anymore.

Adding, moving, and removing users across groups may become increasingly daunting tasks for a Linux administrator. Knowing at any time which users belong to which groups is valuable information, both for reporting purposes and user automation workflows. The following section provides a few commands for viewing user and group data.

Viewing users and groups

In this section, we will provide some potentially useful commands for retrieving group and group membership information. Before we get into any commands, we should keep in mind that group information is stored in the **/etc/group** and **/etc/gshadow** files. Among the two, the former has the information we're most interested in.

We can parse the **/etc/group** file to retrieve all groups, as follows:

```
cat /etc/group | cut -d: -f1 | column | less
```

The command yields the following output:

```
root                dip                 input
daemon              www-data            sgx
bin                 backup              kvm
sys                 operator            render
adm                 list                lxd
tty                 irc                 _ssh
disk                src                 crontab
lp                  gnats               syslog
mail                shadow              uuidd
news                utmp                tcpdump
uucp                video               tss
man                 sasl                landscape
proxy               plugdev             packt
kmem                staff               fwupd-refresh
dialout             games               plocate
fax                 users               alex
voice               nogroup             julian
cdrom               systemd-journal     admin
floppy              systemd-network     developers
tape                systemd-resolve     devops
sudo                messagebus          managers
audio               systemd-timesync    alex2
```

Figure 4.22 – Retrieving all group names

A similar command would use **getent**, which we can use like this:

```
getent group | cut -d: -f1 | column | less
```

The output of the preceding command is identical to the output shown in *Figure 4.22*. We can retrieve the information of an individual group (for example, **developers**) with the following command:

```
getent group developers
```

This is the output:

```
packt@neptune:~$ cat /etc/group | cut -d: -f1 | column | less
packt@neptune:~$ getent group | cut -d: -f1 | column | less
packt@neptune:~$ getent group developers
developers:x:1200:julian2
packt@neptune:~$
```

Figure 4.23 – Retrieving information for a single group

The output of the preceding command also reveals the members of the **developers** group (**julian2**).

To list all groups a specific user is a member of, we can use the **groups** command. For example, the following command lists all groups the user **alex** is a member of:

```
groups alex
```

This is the command output:

```
packt@neptune:~$ groups alex
alex : alex devops managers
packt@neptune:~$
```

Figure 4.24 – Retrieving group membership information of a user

The output of the previous command shows the groups for the user **`alex`**, starting with the primary group (**`alex`**).

A user can retrieve their own group membership using the **`groups`** command-line utility without specifying a group name. The following command is executed in a terminal session of the user **`packt`**, who is also an administrator (superuser):

```
groups
```

The command yields this output:

```
packt@neptune:~$ groups
packt adm cdrom sudo dip plugdev lxd
packt@neptune:~$
```

Figure 4.25 – The current user's groups

There are many other ways and commands to retrieve user- and group-related information. We hope that the preceding examples provide a basic idea about where and how to look for some of this information.

Next, let's look at how a user can switch or log in to specific groups.

Group login sessions

When a user logs in to the system, the group membership context is automatically set to the user's primary group. Once the user is logged in, any user-initiated task (such as creating a file or running a program) is associated with the user's primary group membership permissions. A user may also choose to access resources in other groups where they are also a member (that is, supplementary or secondary groups). To switch the group context or log in with a new group membership, a user invokes the **`newgrp`** command-line utility.

The basic syntax for the **`newgrp`** command is this:

```
newgrp GROUP
```

In the following example, we assume a user (**julian**) is a member of multiple groups—**admin** as the primary group, and **developers**/**devops** as secondary groups:

```
id julian
```

This is the output:

```
packt@neptune:~$ id julian
uid=1003(julian) gid=1100(admin) groups=1100(admin),1200(developers),1300(devops)
packt@neptune:~$
```

Figure 4.26 – A user with multiple group memberships

Let's impersonate the user **julian** for a while. We are currently logged in as the user **packt**. To change to the user **julian**, we will use the following command:

```
su julian
```

Remember that the user **julian** needs to have their password set in order to authenticate.

When logged in as **julian**, the default login session has the following user and group context:

```
whoami
```

In our case, this is the output:

```
packt@neptune:~$ whoami
packt
packt@neptune:~$ su julian
Password:
$ whoami
julian
$
```

Figure 4.27 – Getting the current user

The **whoami** command provides the current UID (see more details on the command with **man whoami** or **whoami --help**), as follows:

```
groups
```

This is the output:

```
$ whoami
julian
$ groups
admin developers devops
$
```

Figure 4.28 – Getting the current user's groups

The **groups** command displays all groups that the current user is a member of (see more details on the command with **man groups** or **groups --help**).

The user can also view their IDs (user and GIDs) by invoking the **id** command, as follows:

```
id
```

This is the output:

```
$ groups
admin developers devops
$ id
uid=1003(julian) gid=1100(admin) groups=1100(admin),1200(developers),1300(devops)
$
```

Figure 4.29 – Viewing the current user and GID information

There are various invocations of the **id** command that provide information on the current user and group session. The following command (with the **-g, --group** option) retrieves the ID of the current group session for the user:

```
id -g
1100
```

In our case, the preceding command shows **1100**—the GID corresponding to the user's primary group, which is **admin** (see the **gid** attribute in *Figure 4.30*). Upon login, the default group session is always the primary group corresponding to the user. If the user were to create a file, for example, the file permission attributes would reflect the primary group's ID. We'll look at the file permissions in more detail in the *Managing permissions* section.

Now, let's switch the group session for the current user to **developers**, as follows:

```
newgrp developers
```

The current group session yields this:

```
id -g
1200
```

The GID corresponds to the **developers** secondary GID, as displayed by the **groups** attribute in *Figure 4.30*: **1200(developers)**. If the user created any files now, the related file permission attributes would have the **developers** GID:

```
$ id
uid=1003(julian) gid=1100(admin) groups=1100(admin),1200(developers),1300(devops)
$ id -g
1100
$ newgrp developers
$ id -g
1200
$ id
uid=1003(julian) gid=1200(developers) groups=1200(developers),1100(admin),1300(de
vops)
$
```

Figure 4.30 – Switching the group session

If the user attempts to log in to a group they are not a member of (for example, **managers**), the **newgrp** command prompts for the **managers** group's password:

```
newgrp managers
```

If our user had the **managers** group password, or if they were a superuser, the group login attempt would succeed. Otherwise, the user would be denied access to the **managers** group's resources.

We conclude here our topic of managing users and groups. The examples of the related administrative tasks used throughout this section are certainly all-encompassing. In many of these cases, there are multiple ways to achieve the same result, using different commands or approaches.

By now, you should be relatively proficient in managing users and groups, and comfortable using the various command-line utilities for operating the related changes. Users and groups are managed in a relational fashion, where users belong to a group or groups are associated with users. We also learned that creating and managing users and groups requires superuser privileges. In Linux, user data is stored in the **/etc/passwd** and **/etc/shadow** files, while group information is found in **/etc/group** and **/etc/gshadow**. Besides using the dedicated command-line utilities, users and groups can also be altered by manually editing these files.

Next, we'll turn to the security and isolation context of the multiuser group environment. In Linux, the related functionality is accomplished by a system-level access layer that controls the read, write, and execute permissions of files and directories, by specific users and groups.

The following section explores the management and administrative tasks related to these permissions.

Managing permissions

A key tenet of Linux is the ability to allow multiple users to access the system while performing independent tasks simultaneously. The smooth operation of this multiuser, multitasking environment is controlled via **permissions**. The Linux kernel provides a robust framework for the underlying

security and isolation model. At the user level, dedicated tools and command-line utilities help Linux users and system administrators with related permission management tasks.

For some Linux users, especially beginners, Linux permissions may appear confusing at times. This section attempts to demystify some of the key concepts about file and directory permissions in Linux. You will learn about the basic permission *rights* of accessing files and directories—the *read*, *write*, and *execution* permissions. We explore some of the essential administrative tasks for viewing and changing permissions, using system-level command-line utilities.

Most of the topics discussed in this section should be regarded closely with users and groups. The related idioms can be as simple as *a user can read or update a file*, *a group has access to these files and directories*, or *a user can execute this program*.

Let's start with the basics, introducing file and directory permissions.

File and directory permissions

In Linux, permissions can be regarded as the *rights* or *privileges* to act upon a file or a directory. The basic rights, or *permission attributes*, are outlined here:

- **Read**: A *read* permission of a file allows users to view the content of the file. On a directory, the read permission allows users to list the content of the directory.
- **Write**: A *write* permission of a file allows users to modify the content of the file. For a directory, the write permission allows users to modify the content of the directory by adding, deleting, or renaming files.
- **Execute**: An *execute* or *executable* permission of a file allows users to run the related script, application, or service appointed by the file. For a directory, the execute permission allows users to enter the directory and make it the current working directory (using the **`cd`** command).

First, let's take a look at how to reveal the permissions for files and directories.

Viewing permissions

The most common way to view the permissions of a file or directory is by using the **`ls`** command-line utility. The basic syntax of this command is this:

```
ls [OPTIONS] FILE|DIRECTORY
```

Here is an example use of the **`ls`** command to view the permissions of the **`/etc/passwd`** file:

```
ls -l /etc/passwd
```

The command yields the following output:

```
-rw-r--r-- 1 root root 2010 Mar  9 08:57 /etc/passwd
```

The `-l` option of the `ls` command provides a detailed output by using the *long listing format*, according to the `ls` documentation (`man ls`).

Let's analyze the output, as follows:

```
-rw-r--r-- 1 root root 2010 Mar  9 08:57 /etc/passwd
```

We have nine segments, separated by single whitespace characters (delimiters). These are outlined here:

- `-rw-r--r--`: The file access permissions
- `1`: The number of hard links
- `root`: The *user* who is the owner of the file
- `root`: The *group* that is the owner of the file
- `2010`: The size of the file
- `Mar`: The month the file was created
- `9`: The day of the month the file was created
- `08:57`: The time of day the file was created
- `/etc/passwd`: The filename

Let's examine the file access permissions field (`-rw-r--r--`). File access permissions are defined as a 10-character field, grouped as follows:

- The first character (attribute) is reserved for the file type (see the *File types* section).
- The next 9 characters represent a 9-bit field, defining the effective permissions as 3 sequences of 3 attributes (bits) each: *user owner* permissions, *group owner* permissions, and *all other users'* permissions (see the *Permission attributes* section).

Let's take a look at the file type attributes.

File type attributes

The file type attributes are listed here:

- `d`: Directory
- `-`: Regular file
- `l`: Symbolic link
- `p`: Named pipe—a special file that facilitates communication between programs
- `s`: Socket—similar to a pipe but with bidirectional network communications
- `b`: Block device—a file that corresponds to a hardware device
- `c`: Character device—similar to a block device

Let’s have a closer look at the permission attributes.

Permission attributes

As previously noted, the access permissions are represented by a 9-bit field, a group of 3 sequences, each with 3 bits, defined as follows:

- **Bits 1-3**: *User* owner permissions
- **Bits 4-6**: *Group* owner permissions
- **Bits 7-9**: *All* other users’ (or *world*) permissions

Each permission attribute is a bit flag in the binary representation of the related 3-bit sequence. They can be represented either as a character or as an equivalent numerical value, also known as the *octal* value, depending on the range of the bit they represent.

Here are the permission attributes with their respective octal values:

- **`r`**: *Read* permission; 2 ^ 2 = **`4`** (bit 2 set)
- **`w`**: *Write* permission: 2 ^ 1 = **`2`** (bit 1 set)
- **`x`**: *Execute* permission: 2 ^ 0 = **`1`** (bit 0 set)
- **`-`**: *No* permission: **`0`** (no bits set)

The resulting corroborated number is also known as the *octal value* of the file permissions (see the *File permission examples* section). Here’s an illustration of the file permission attributes:

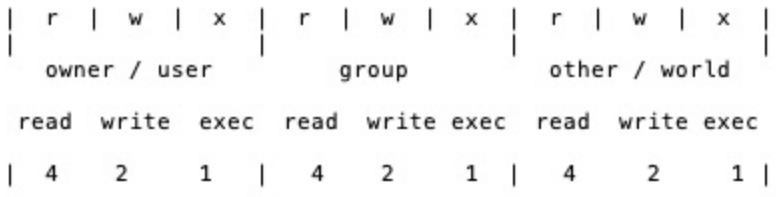

Figure 4.31 – The file permission attributes

Next, let’s consider some examples.

File permission examples

Now, let’s go back and evaluate the file access permissions for **`/etc/passwd`**: **`-rw-r--r--`**, as follows:

- **`-`**: The first character (byte) denotes the file type (a regular file, in our case)
- **`rw-`**: The next three-character sequence indicates the user owner permissions; (in our case, read (**`r`**); write (**`w`**); octal value = **`4`** (**`r`**) + **`2`** (**`w`**) = **`6`** (**`rw`**))
- **`r--`**: The next 3-byte sequence defines the group owner permissions (in our case, read (**`r`**); octal value = **`4`** (**`r`**))

- **`r--`**: The last three characters denote the permissions for all other users in the system (in our case, read (**`r`**); octal value = **`4`** (**`r`**))

According to the preceding information, the resulting octal value of the **`/etc/passwd`** file access permissions is **`644`**. Alternatively, we can query the octal value with the **`stat`** command, as follows:

```
stat --format '%a' /etc/passwd
```

The command yields the following output:

```
644
```

The **`stat`** command displays the file or filesystem status. The **`--format`** option parameter specifies the access rights in octal format (**`'%a'`**) for the output.

Here are a few examples of access permissions, with their corresponding octal values and descriptions. The three-character sequences are intentionally delimited with whitespace for clarity. The leading file type has been omitted:

- **`rwx`** (**`777`**): Read, write, and execute for all users including owner, group, and world
- **`rwx r-x`** (**`755`**): Read and execute for all users; the file owner has write permissions
- **`rwx r-x ---`** (**`750`**): Read and execute for owner and group; the owner has write permissions while others have no access
- **`rwx --- ---`** (**`700`**): Read, write, and execute for owner; everyone else has no permissions
- **`rw- rw- rw-`** (**`666`**): Read and write for all users; there are no execute permissions
- **`rw- rw- r--`** (**`664`**): Read and write for owner and group; read for others
- **`rw- rw- ---`** (**`660`**): Read and write for owner and group; others have no permissions
- **`rw- r-- r--`** (**`644`**): Read and write for owner; read for group and others
- **`rw- r-- ---`** (**`640`**): Read and write for owner; read for group; no permissions for others
- **`rw- --- ---`** (**`600`**): Read and write for owner; no permissions for group and others
- **`r-- --- ---`** (**`400`**): Read for owner; no permissions for others

Read, write, and execute are the most common types of file access permissions. So far, we have mostly focused on permission types and their representation. In the next section, we will explore a few command-line tools used for altering permissions.

Changing permissions

Modifying file and directory access permissions is a common Linux administrative task. In this section, we will learn about a few command-line utilities that are handy when it comes to changing permissions and ownership of files and directories. These tools are installed with any modern-day Linux distribution, and their use is similar across most Linux platforms.

Using chmod

The **chmod** command is short for *change mode*, and it's used to set access permissions on files and directories. The **chmod** command can be used by both the current user (owner) and a superuser.

Changing permissions can be done in two different modes: **relative** and **absolute**. Let's take a look at each of them.

Using chmod in relative mode

Changing permissions in **relative** mode is probably the easiest of the two. It is important to remember the following:

- *To whom* we change permissions: **u** = user (owner), **g** = group, **o** = others
- *How* we change permissions: **+** = add, **-** = remove, **=** = exactly as is
- *Which* permission we change: **r** = read, **w** = write, **x** = execute

Let's explore a few examples of using **chmod** in relative mode.

In our first example, we want to add write (**w**) permissions for all *other* (**o**) users (*world*), to **myfile**, as follows:

```
chmod o+w myfile
```

The related command-line output is shown here:

```
packt@neptune:~$ touch myfile
packt@neptune:~$ ls -l myfile
-rw-rw-r-- 1 packt packt 0 Mar  9 17:12 myfile
packt@neptune:~$ chmod o+w myfile
packt@neptune:~$ ls -l myfile
-rw-rw-rw- 1 packt packt 0 Mar  9 17:12 myfile
packt@neptune:~$
```

Figure 4.32 – Setting write permissions to all other users

In the next example, we remove the read (**r**) and write (**w**) permissions for the current user owner (**u**) of **myfile**, as follows:

```
chmod u-rw myfile
```

The command-line output is shown here:

```
packt@neptune:~$ ls -l myfile
-rw-rw-rw- 1 packt packt 0 Mar  9 17:12 myfile
packt@neptune:~$ chmod u-rw myfile
packt@neptune:~$ ls -l myfile
----rw-rw- 1 packt packt 0 Mar  9 17:12 myfile
packt@neptune:~$
```

Figure 4.33 – Removing read-write permissions for owner

We did not use `sudo` in either of the preceding examples since we carried out the operations as the current owner of the file (`packt`).

In the following example, we assume that `myfile` has read, write, and execute permissions for everyone. Then, we carry out the following changes:

- Remove the read (`r`) permission for the owner (`u`)
- Remove the write (`w`) permission for the owner (`u`) and group (`g`)
- Remove the read (`r`), write (`w`), and execute (`x`) permissions for everyone else (`o`)

This is illustrated in the following code snippet:

```
chmod u-r,ug-w,o-rwx myfile
```

The command-line output is shown here:

```
packt@neptune:~$ chmod u+rwx,ug+rwx,o+rwx myfile
packt@neptune:~$ ls -l myfile
-rwxrwxrwx 1 packt packt 0 Mar  9 17:12 myfile
packt@neptune:~$ chmod u-r,ug-w,o-rwx myfile
packt@neptune:~$ ls -l myfile
---xr-x--- 1 packt packt 0 Mar  9 17:12 myfile
packt@neptune:~$
```

Figure 4.34 – A relatively complex invocation of chmod in relative mode

Next, let's look at a second way of changing permissions: using the `chmod` command-line utility in absolute mode, by specifying the octal number corresponding to the access permissions.

Using chmod in absolute mode

The **absolute** mode invocation of `chmod` changes all permission attributes at once, using an *octal* number. The *absolute* designation of this method is due to changing permissions without any reference to existing ones, by simply assigning the octal value corresponding to the access permissions.

Here's a quick list of the octal values corresponding to effective permissions:

- **`7 rwx`**: Read, write, and execute
- **`6 rw-`**: Read and write
- **`5 r-w`**: Read and execute
- **`4 r--`**: Read
- **`3 -wx`**: Write and execute
- **`2 -w-`**: Write
- **`1 --x`**: Execute
- **`0 ---`**: No permissions

In the following example, we change the permissions of **`myfile`** to read (**`r`**), write (**`w`**), and execute (**`x`**) for everybody:

```
chmod 777 myfile
```

The related change is illustrated by the following command-line output:

```
packt@neptune:~$ chmod 777 myfile
packt@neptune:~$ ls -l myfile
-rwxrwxrwx 1 packt packt 0 Mar  9 17:12 myfile
packt@neptune:~$
```

Figure 4.35 – The chmod invocation in absolute mode

For more information about the **`chmod`** command, please refer to the related documentation (**`man chmod`**).

Let's now look at our next command-line utility, specializing in file and directory ownership changes.

Using chown

The **`chown`** command (short for *change owner*) is used to set the ownership of files and directories. Typically, the **`chmod`** command can only be run with *superuser* privileges (that is, by a *sudoer*). Regular users can only change the *group* ownership of their files, and only when they are a member of the target group.

The syntax of the **`chown`** command is shown here:

```
chown [OPTIONS] [OWNER][:[GROUP]] FILE
```

Usually, we invoke the **`chown`** command with both user *and* group ownerships—for example, like this:

```
sudo chown julian:developers myfile
```

The related command-line output is shown here:

```
packt@neptune:~$ sudo chown julian:developers myfile
[sudo] password for packt:
packt@neptune:~$ ls -l myfile
-rwxrwxrwx 1 julian developers 0 Mar  9 17:12 myfile
packt@neptune:~$
```

Figure 4.36 – A simple invocation of the chown command

One of the most common uses of **chown** is for *recursive mode* invocation, with the **-R, --recursive** option. The following example changes the ownership permissions of all files in **mydir** (directory), initially owned by **root**, to **julian**:

```
sudo chown -R julian:julian mydir/
```

The related changes are shown in the following command-line output:

```
drwxrwxr-x 2 packt packt 4096 Mar  9 17:37 subdir

mydir/subdir:
total 0
-rw-rw-r-- 1 packt packt 0 Mar  9 17:37 file1
-rw-rw-r-- 1 packt packt 0 Mar  9 17:37 file2
-rw-rw-r-- 1 packt packt 0 Mar  9 17:37 file3
packt@neptune:~$ sudo chown -R julian:julian mydir/
packt@neptune:~$ ls -lR mydir/
mydir/:
total 4
-rw-rw-r-- 1 julian julian    0 Mar  9 17:35 file1
-rw-rw-r-- 1 julian julian    0 Mar  9 17:35 file2
-rw-rw-r-- 1 julian julian    0 Mar  9 17:35 file3
drwxrwxr-x 2 julian julian 4096 Mar  9 17:37 subdir

mydir/subdir:
total 0
-rw-rw-r-- 1 julian julian 0 Mar  9 17:37 file1
-rw-rw-r-- 1 julian julian 0 Mar  9 17:37 file2
-rw-rw-r-- 1 julian julian 0 Mar  9 17:37 file3
```

Figure 4.37 – Invoking ls and chown in recursive mode

For more information about the **chown** command, please refer to the related documentation (**man chown**).

Next, let's briefly look at a similar command-line utility that specializes exclusively in group ownership changes.

Using chgrp

The **chgrp** command (short for *change group*) is used to change the *group* ownership for files and directories. In Linux, files and directories typically belong to a user (owner) or a group. We can set user ownership by using the **chown** command-line utility, while group ownership can be set with **chgrp**.

The syntax for **chgrp** is shown here:

```
chgrp [OPTIONS] GROUP FILE
```

The following example changes the group ownership of **myfile** to the **developers** group:

```
sudo chgrp developers myfile
```

The changes are shown in the following output:

```
packt@neptune:~$ sudo chgrp developers myfile
packt@neptune:~$ ls -l myfile
-rwxrwxrwx 1 julian developers 0 Mar  9 17:12 myfile
packt@neptune:~$
```

Figure 4.38 - Using chgrp to change group ownership

The preceding command has been invoked with superuser privileges (**sudo**) since the current user (**packt**) is not an admin for the **developers** group.

For more information about the **chgrp** utility, please refer to the tool's command-line help (**chgrp --help**).

Using umask

The **umask** command is used to view or set the default *file mode mask* in the system. The file mode represents the default permissions for any new files and directories created by a user. For example, the default file mode masks in Ubuntu are given here:

- **0002** for a regular user
- **0022** for the **root** user

As a general rule in Linux, the *default permissions* for new files and directories are calculated with the following formulas:

- **0666 – umask**: For a new file created by a regular user
- **0777 – umask**: For a new directory created by a regular user

According to the preceding formula, on Ubuntu, we have the following default permissions:

- File (regular user): **`0666 – 0002 = 0664`**
- File (**`root`**): **`0666 – 0022 = 0644`**
- Directory (regular user): **`0777 – 0002 = 0775`**
- Directory (**`root`**): **`0777 – 0022 = 0755`**

In the following examples, run on Ubuntu, we create a file (**`myfile`**) and a directory (**`mydir`**), using the terminal session of a regular user (**`packt`**). Then, we query the **`stat`** command for each and verify that the default permissions match the values enumerated previously for regular users (file: **`664`**, directory: **`775`**).

Let's start with the default file permissions first, as follows:

```
touch myfile2
stat --format '%a' myfile2
```

The related output is shown here:

```
664
```

Next, let's verify the default directory permissions, as follows:

```
mkdir mydir2
stat --format '%a' mydir2
```

The related output is shown here:

```
775
```

Here's a list with the typical **`umask`** values for files and directories on Linux systems:

umask value	Files		Directories	
0000	666	rw-rw-rw-	777	rwxrwxrwx
0002	664	rw-rw-r--	775	rwxrwxr-x
0022	644	rw-r--r--	755	rwxr-xr-x
0027	640	rw-r-----	750	rwxr-x---
0077	600	rw-------	700	rwx------
0277	400	r--------	500	r-x------

Figure 4.39 – Typical umask values on Linux

For more information about the **`umask`** utility, please refer to the tool's command-line help (**`umask --help`**).

File and directory permissions are critical for a secure environment. Users and processes should operate exclusively within the isolation and security constraints controlled by permissions, to avoid inadvertent or deliberate interference with the use and ownership of system resources. There are

cases, particularly in user impersonation situations, when the access rights may involve some special permission attributes. Let's have a look at them.

Special permissions

In Linux, the ownership of files and directories is usually determined by the UID and GID of the user —or group—who created them. The same principle applies to applications and processes—they are owned by the users who launch them. The special permissions are meant to change this default behavior when needed.

Here are the special permission flags, with their respective octal values:

- **`setuid`**: 2 ^ 2 = **`4`** (bit 2 set)
- **`setgid`**: 2 ^ 1 = **`2`** (bit 1 set)
- **`sticky`**: 2 ^ 0 = **`1`** (bit 0 set)

When any of these special bits are set, the overall octal number of the access permissions will have an extra digit, with the leading (high-order) digit corresponding to the special permission's octal value.

Let's look at these special permission flags, with examples for each.

The setuid permission

With the **`setuid`** bit set, when an executable file is launched, it will run with the privileges of the file owner instead of the user who launched it. For example, if the executable is owned by **`root`** and launched by a *regular* user, it will run with **`root`** privileges. The **`setuid`** permission could pose a potential security risk when used inadequately, or when vulnerabilities of the underlying process could be exploited.

In the file access permission field, the **`setuid`** bit could have either of the following representations:

- **`s`** *replacing* the corresponding executable bit (**`x`**) (when the executable bit is present)
- **`S`** (the capital letter) for a non-executable file

The **`setuid`** permission can be set via the following **`chmod`** command (for example, for the **`myscript.sh`** executable file):

```
chmod u+s myscript.sh
```

The resulting file permissions are shown here (including the octal value): **`-rwsrwxr-x`** (**`4775`**).

Here is the related command-line output:

```
packt@neptune:~$ touch myscript.sh
packt@neptune:~$ chmod +x myscript.sh
packt@neptune:~$ ls -l myscript.sh
-rwxrwxr-x 1 packt packt 0 Mar  9 17:02 myscript.sh
packt@neptune:~$ chmod u+s myscript.sh
packt@neptune:~$ ls -l myscript.sh
-rwsrwxr-x 1 packt packt 0 Mar  9 17:02 myscript.sh
packt@neptune:~$ stat --format '%a' myscript.sh
4775
packt@neptune:~$
```

Figure 4.40 – The setuid permission

In the preceding screenshot, you can see the difference in permissions. Before applying the **chmod** command, the permissions are **-rwxrwxr-x**, and after applying the **setuid** permission with the **chmod** command, an **s** (referring to **setuid**) is included in the user's permission, **-rwsrwxr-x**. For more information on **setuid**, please visit https://docs.oracle.com/cd/E19683-01/816-4883/secfile-69/index.html or refer to the **chmod** command-line utility documentation (**man chmod**).

The setgid permission

While **setuid** controls user impersonation privileges, **setgid** has a similar effect on group impersonation permissions.

When an executable file has the **setgid** bit set, it runs using the permissions of the group that owns the file, rather than the group of the user who initiated it. In other words, the GID of the process is the same as the GID of the file.

When used on a directory, the **setgid** bit changes the default ownership behavior so that files created within the directory will have group ownership of the parent directory instead of the group associated with the user who created them. This behavior could be adequate in file-sharing situations when files can be changed by all users associated with the parent directory's owner group.

The **setgid** permission can be set via the following **chmod** command (for example, for the **myscript.sh** executable file, the original file, before applying **setuid** to it):

```
chmod g+s myscript.shw
```

The resulting file permissions are shown here (including the octal value): **-rwxrwsr-x** (**2775**).

The command-line output is shown here:

```
packt@neptune:~$ chmod g+s myscript.sh
packt@neptune:~$ ls -l myscript.sh
-rwxrwsr-x 1 packt packt 0 Mar  9 17:07 myscript.sh
packt@neptune:~$ stat --format '%a' myscript.sh
2775
packt@neptune:~$
```

Figure 4.41 – The setgid permission

For more information on **setgid**, please visit https://en.wikipedia.org/wiki/Setuid or refer to the **chmod** command-line utility documentation (**man chmod**).

The sticky permission

The **sticky** bit has no effect on files. For a directory with the **sticky** permission, only the user owner or group owner of the directory can delete or rename files within the directory. Users or groups with write access to the directory, by way of user or group ownership, cannot delete or modify files in the directory. The **sticky** permission is useful when a directory is owned by a privileged group whose members share write access to files in that directory.

The **sticky** permission can be set via the following **chmod** command (for example, for the **mydir** directory):

```
chmod +t mydir
```

The resulting directory permissions are shown here (including the octal value): **drwxrwxr-t** (**1775**).

The command-line output is shown here:

```
packt@neptune:~$ mkdir mydir
packt@neptune:~$ chmod +t mydir
packt@neptune:~$ ls -ld mydir
drwxrwxr-t 2 packt packt 4096 Mar  9 17:10 mydir
packt@neptune:~$ stat --format '%a' mydir
1775
packt@neptune:~$
```

Figure 4.42 – The sticky permission

For more information on **sticky**, please visit https://en.wikipedia.org/wiki/Setuid or refer to the **chmod** command-line utility documentation (**man chmod**).

Interpreting permissions can be a daunting task. This section aimed to demystify some of the related intricacies, and we hope that you will feel more comfortable handling file and directory permissions in everyday Linux administration tasks.

Summary

In this chapter, we explored some of the essential concepts related to managing users and groups in Linux. We learned about file and directory permissions and the different access levels of a multiuser environment. For each main topic, we focused on basic administrative tasks, providing various practical examples and using typical command-line tools for everyday user access and permission management operations.

Managing users and groups, and the related filesystem permissions that come into play, is an indispensable skill of a Linux administrator. The knowledge gained in this chapter will, we hope, put you on track to becoming a proficient superuser.

In the following chapter, we continue our journey of mastering Linux internals by exploring processes, daemons, and **inter-process communication** (**IPC**) mechanisms. An important aspect to keep in mind is that processes and daemons are also *owned* by users or groups. The skills learned in this chapter will help us navigate the related territory when we look at *who runs what* at any given time in the system.

Questions

Here are a few thoughts and questions that sum up the main ideas covered in this chapter:

1. What is a superuser?

Hint: Try `sudo`

2. Think of a command-line utility for creating users. Can you think of another one?

Hint: Think about `adduser` and `useradd`

3. What is the octal value of the `-rw-rw-r—` access permission?

Hint: Remember what the values of `r`, `w`, and `x` are: `4`, `2`, and `1`

4. What is the difference between a primary group and a secondary (supplementary) group?
5. How do you change the ownership of a user's home directory?
6. Can you remove a user from the system without deleting their home directory? How?

Further reading

Here are a few Packt titles that can help you with the task of user management:

- *Mastering Ubuntu Server – Fourth Edition*, *Jay LaCroix*
- *Red Hat Enterprise Linux 9 Administration – Second Edition*, *Pablo Iranzo Gómez*, *Pedro Ibáñez Requena*, *Miguel Pérez Colino*, and *Scott McCarty*

5

Working with Processes, Daemons, and Signals

Linux is a multitasking operating system. Multiple programs or tasks can run in parallel, each with its own identity, scheduling, memory space, permissions, and system resources. **Processes** encapsulate the execution context of any such program. Understanding how processes work and communicate with each other is an important skill for any seasoned Linux system administrator and developer to have.

This chapter explores the basic concepts behind Linux processes. We'll look at different types of processes, such as **foreground** and **background** processes, with special emphasis being placed on **daemons** as a particular type of background process. We'll closely study the anatomy of a process and various inter-process communication mechanisms in Linux – **signals** in particular. Along the way, we'll learn about some of the essential command-line utilities for managing processes and daemons and working with signals. We will also introduce you to **scripts** for the first time in this book, which are described in detail later in *Chapter 8*, *Linux Shell Scripting*. If you feel like you need more information when dealing with the scripts in this chapter, take a look at *Chapter 8* in advance.

In this chapter, we will cover the following topics:

- Introducing processes
- Working with processes
- Working with daemons
- Exploring inter-process communication

IMPORTANT NOTE

As we navigate through the content, we will occasionally reference signals before their formal introduction in the second half of this chapter. In Linux, signals are almost exclusively used in association with processes, hence our approach of becoming familiar with processes first. Yet, leaving the signals out from some of the process' internals would do a disservice to understanding how processes work. Where signals are mentioned, we'll point to the related section for further reference. We hope that this approach provides you with a better grasp of the overall picture and the inner workings of processes and daemons.

Now, before we start, let's look at the essential requisites for our study.

Technical requirements

Practice makes perfect. Running the commands and examples in this chapter by hand would go a long way toward you learning about processes. As with any chapter in this book, we recommend that you have a working Linux distribution installed on a VM or PC desktop platform. We'll be using Ubuntu or Fedora, but most of the commands and examples would be similar on any other Linux platform.

Introducing processes

A **process** represents the running instance of a program. In general, a program is a combination of instructions and data, compiled as an executable unit. When a program runs, a process is created. In other words, a process is simply a program in action. Processes execute specific tasks, and sometimes, they are also referred to as **jobs** (or **tasks**).

There are many ways to create or start a process. In Linux, every command starts a process. A command could be a user-initiated task in a Terminal session, a script, or a program (executable) that's invoked manually or automatically.

Usually, the way a process is created and interacts with the system (or user) determines its process type. Let's take a closer look at the different types of processes in Linux.

Understanding process types

At a high level, there are two major types of processes in Linux:

- **Foreground** (*interactive*)
- **Background** (*non-interactive* or *automated*)

Interactive processes assume some kind of user interaction during the lifetime of the process. Non-interactive processes are unattended, which means that they are either automatically started (for example, on system boot) or are scheduled to run at a particular time and date via job schedulers (for example, using the `at` and `cron` command-line utilities).

Our approach to exploring process types mainly pivots around the preceding classification. There are various other views or taxonomies surrounding process definitions, but they could ultimately be reduced to either foreground or background processes.

For example, batch processes and daemons are essentially background processes. Batch processes are automated in the sense that they are not user-generated but invoked by a scheduled task instead. Daemons are background processes that are usually started during system boot and run indefinitely.

There's also the concept of parent and child processes. A parent process may create other subordinate child processes.

We'll elaborate on these types (and beyond) in the following sections. Let's start with the pivotal ones – foreground and background processes.

Foreground processes

Foreground processes, also known as **interactive processes**, are started and controlled through a Terminal session. Foreground processes are usually initiated by a user via an interactive command-line interface. A foreground process may output results to the console (`stdout` or `stderr`) or accept user input. The lifetime of a foreground process is tightly coupled to the Terminal session (parent process). If the user who launched the foreground process exits the Terminal while the process is still running, the process will be abruptly terminated (via a `SIGHUP` signal sent by the parent process; see *Signals* in the *Exploring inter-process communication* section for more details).

A simple example of a foreground process is invocating the system reference manual (`man`) for an arbitrary Linux command (for example, `ps`):

```
man ps
```

The `ps` command displays information about active processes. You will learn more about process management tools and command-line utilities in the *Working with processes* section.

Once a foreground process has been initiated, the user prompt is captured and controlled by the spawned process interface. The user can no longer interact with the initial command prompt until the interactive process relinquishes control to the Terminal session.

Let's look at another example of a foreground process, this time invoking a long-lived task. The following command (one-liner) runs an infinite loop while displaying an arbitrary message every few seconds:

```
while true; do echo "Wait..."; sleep 5; done
```

So long as the command runs without being interrupted, the user won't have an interactive prompt in the Terminal. Using *Ctrl + C* would stop (interrupt) the execution of the related foreground process and yield a responsive command prompt:

```
packt@neptune:~$ while true; do echo "Wait..."; sleep 5; done
Wait...
Wait...
Wait...
^C
packt@neptune:~$
```

Figure 5.1 – A long-lived foreground process

IMPORTANT NOTE

When you press Ctrl + C while a foreground process is running, a ***`SIGINT`*** *signal is sent to the running process by the current (parent) Terminal session, and the foreground process is interrupted. For more information, see the Signals section.*

If we want to maintain an interactive command prompt in the Terminal session while running a specific command or script, we should use a background process.

Background processes

Background processes – also referred to as **non-interactive** or **automatic processes** – run independently of a Terminal session, without expecting any user interaction. A user may invoke multiple background processes within the same Terminal session without waiting on any of them to complete or exit.

Background processes are usually long-lived tasks that don't require direct user supervision. The related process may still display its output in the Terminal console, but such background tasks typically write their results to different files instead (such as log files).

The simplest invocation of a background process appends an ampersand (`&`) to the end of the related command. Building on our previous example (in the *Foreground processes* section), the following command creates a background process that runs an infinite loop, echoing an arbitrary message every few seconds:

```
while true; do echo "Wait..."; sleep 10; done &
```

Note the ampersand (`&`) at the end of the command. By default, a background process would still direct the output (`stdout` and `stderr`) to the console when invoked with the ampersand (`&`), as shown previously. However, the Terminal session remains interactive. In the following figure, we are using the `echo` command while the previous process is still running:

```
packt@neptune:~$ while true; do echo "Wait..."; sleep 10; done &
[1] 983
packt@neptune:~$ Wait...
echo "Interactive Wait...
prompt"
Interactive prompt
packt@neptune:~$ Wait...
kill -9 983
packt@neptune:~$
```

Figure 5.2 – Running a background process

As shown in the preceding screenshot, the background process is given a **process ID** (**PID**) of `983`. While the process is running, we can still control the Terminal session and run a different command, like so:

```
echo "Interactive prompt..."
```

Eventually, we can force the process to terminate with the `kill` command:

```
kill -9 983
```

The preceding command *kills* our background process (with PID `983`). The corresponding signal that's sent by the parent Terminal session to terminate this process is `SIGKILL` (see the *Signals* section for more information) through the `-9` argument in our command.

Both foreground and background processes are typically under the direct control of a user. In other words, these processes are created or started manually as a result of a command or script invocation. There are some exceptions to this rule, particularly when it comes to batch processes, which are launched automatically via scheduled jobs.

There's also a select category of background processes that are automatically started during system boot and terminated at shutdown without user supervision. These background processes are also known as daemons.

Introducing daemons

A **Daemon** is a particular type of background process that is usually started upon system boot and run indefinitely or until terminated (for example, during system shutdown). A daemon doesn't have a user-controlled Terminal, even though it is associated with a system account (`root` or other) and runs with the related privileges.

Daemons usually serve client requests or communicate with other foreground or background processes. Here are some common examples of daemons, all of which are generally available on most Linux platforms:

- `systemd`: The parent of all processes (formerly known as `init`)
- `crond`: A job scheduler that runs tasks in the background
- `ftpd`: An FTP server that handles client FTP requests
- `httpd`: A web server (Apache) that handles client HTTP requests
- `sshd`: A Secure Shell server that handles SSH client requests

Typically, system daemons in Linux are named with `d` at the end, denoting a daemon process. Daemons are controlled by shell scripts usually stored in the `/etc/init.d/` or `/lib/systemd/` system directory, depending on the Linux platform. Ubuntu, for example, stores daemon script files in

`/etc/init.d/`, while Fedora stores them in `/lib/systemd/`. The location of these daemon files depends on the platform implementation of `init`, a system-wide service manager for all Linux processes.

The Linux init-style startup process generally invokes these shell scripts at system boot. But the same scripts can also be invoked via service control commands, usually run by privileged system users, to manage the lifetime of specific daemons. In other words, a privileged user or system administrator can *stop* or *start* a particular daemon through the command-line interface. Such commands would immediately return the user's control to the Terminal while performing the related action in the background.

Let's take a closer look at the `init` process.

The init process

Throughout this chapter, we'll refer to `init` as the *generic* system initialization engine and service manager on Linux platforms. Over the years, Linux distributions have evolved and gone through various `init` system implementations, such as `SysV`, `upstart`, `OpenRC`, `systemd`, and `runit`. There's an ongoing debate in the Linux community about the supremacy or advantages of one over the other. For now, we will simply regard `init` as a system process, and we will briefly look at its relationship with other processes.

`init` (or `systemd`, and others) is essentially a system daemon, and it's among the first process to start when Linux boots up. The related daemon process continues to run in the background until the system is shut down. `init` is the root (parent) process of all other processes in Linux, in the overall process hierarchy tree. In other words, it is a direct or indirect ancestor of all the processes in the system.

In Linux, the `pstree` command displays the whole process tree, and it shows the `init` process at its root – in our case, `systemd` (on Ubuntu or Fedora).

The output of the preceding command can be seen in the following screenshot:

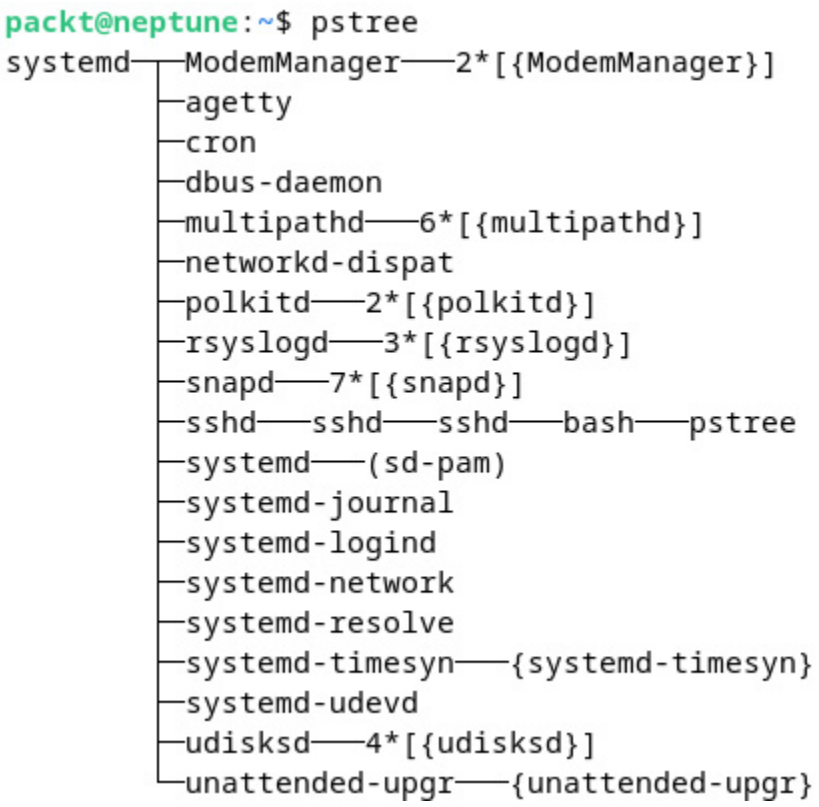

Figure 5.3 – init (systemd), the parent of all processes

The `pstree` command's output illustrates a hierarchy tree representation of the processes, where some appear as parent processes while others appear as child processes. Let's look at the parent and child process types and some of the dynamics between them.

Parent and child processes

A **parent process** creates other subordinate processes, also known as **child processes**. Child processes belong to the parent process that spawned them and usually terminate when the parent process exits (stops execution). A child process may continue to run beyond the parent process's lifetime if it's been instructed to ignore the `SIGHUP` signal that's invoked by the parent process upon termination (for example, via the `nohup` command). See the *Signals* section for more information.

In Linux, all processes except the `init` process (with its variations) are children of a specific process. Terminating a child process won't stop the related parent process from running. A good practice for terminating a parent process when the child is done processing is to exit from the parent process itself after the child process completes.

There are cases when processes run unattended, based on a specific schedule. Running a process without user interaction is known as batch processing. We'll look at batch processes next.

Batch processes

A **batch process** is typically a script or a command that's been scheduled to run at a specific date and time, usually in a periodic fashion. In other words, batch processing is a background process that's spawned by a **job scheduler**. In most common cases, batch processes are resource-intensive tasks that are usually scheduled to run during less busy hours to avoid system overload. On Linux, the most commonly used tools for job scheduling are `at` and `cron`. While `cron` is better suited to scheduled task management complexities, `at` is a more lightweight utility, better suited for one-off jobs. A detailed study of these commands is beyond the scope of this chapter. You may refer to the related system reference manuals for more information (`man at` and `man cron`).

We'll conclude our study of process types with orphan and zombie processes.

Orphan and zombie processes

When a child process is terminated, the related parent process is notified with a `SIGCHILD` signal. The parent can go on running other tasks or may choose to spawn another child process. However, there may be instances when the parent process is terminated before a related child process completes execution (or exits). In this case, the child process becomes an **orphan process**. In Linux, the `init` process – the parent of all processes – automatically becomes the new parent of the orphan process.

Zombie processes (also known as **defunct processes**) are references to processes that have completed execution (and exited) but are still lingering in the system process table (according to the `ps` command).

The main difference between the zombie and orphan processes is that a zombie process is dead (terminated), while an orphan process is still running.

As we differentiate between various process types and their behavior, a significant part of the related information is reflected in the composition or data structure of the process itself. In the next section, we'll take a closer look at the makeup of a process, which is mostly echoed through the `ps` command-line utility – an ordinary yet very useful process explorer on Linux systems.

The anatomy of a process

In this section, we will explore some of the common attributes of a Linux process through the lens of the `ps` and `top` command-line utilities. We hope that taking a practical approach based on these tools will help you gain a better understanding of process internals, at least from a Linux administrator's perspective. Let's start by taking a brief look at these commands. The `ps` command displays a current snapshot of the system processes. This command has the following syntax:

```
ps [OPTIONS]
```

The following command displays the processes owned by the current Terminal session:

```
ps
```

The output of the preceding command can be seen in the following screenshot:

```
packt@neptune:~$ ps
    PID TTY          TIME CMD
    968 pts/0    00:00:00 bash
   1008 pts/0    00:00:00 ps
packt@neptune:~$
```

Figure 5.4 – Displaying processes owned by the current shell

Let's look at each field in the top (header) row of the output and explain their meaning in the context of our relevant process – that is, the `bash` Terminal session:

- **`PID`**: Each process in Linux has a **`PID`** value automatically assigned by the kernel when the process is created. The **`PID`** value is a positive integer and is always guaranteed to be unique.

In our case, the relevant process is **`bash`** (the current shell), with a PID of **`171233`**.

- **`TTY`**: **TTY** is short for **teletype**, more popularly known as a controlling Terminal or device for interacting with a system. In the context of a Linux process, the **`TTY`** attribute denotes the type of Terminal the process interacts with. In our example, the **`bash`** process representing the Terminal session has **`pts/0`** as its TTY type. **PTS** or **`pts`** stands for **pseudo terminal slave** and indicates the input type – a Terminal console – controlling the process. **`/0`** indicates the ordinal sequence of the related Terminal session. For example, an additional SSH session would have **`pts/1`**, and so on.
- **`TIME`**: The **`TIME`** field represents the cumulative CPU utilization (or time) spent by the process (in **`[DD-]hh:mm:ss`** format). Why is it zero (**`00:00:00`**) for the **`bash`** process in our example? We may have run multiple commands in our Terminal session, yet the CPU utilization could still be zero. That's because the CPU utilization measures (and accumulates) the time spent for each command, and not the parent Terminal session overall. If the commands complete within a fraction of a second, they will not amount to a significant CPU utilization being shown in the **`TIME`** field.
- **`CMD`**: The **`CMD`** field stands for command and indicates the name or full path of the command (including the arguments) that created the process. For well-known system commands (for example, **`bash`**), **`CMD`** displays the command's name, including its arguments.

The process attributes we've explored thus far represent a relatively simple view of Linux processes. There are situations when we may need more information. For example, the following command provides additional details about the processes running in the current Terminal session:

```
ps -l
```

The **`-l`** option parameter invokes the so-called *long format* for the **`ps`** output:

```
packt@neptune:~$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000     968     967  0  80   0 -  2216 do_wai pts/0    00:00:00 bash
0 R  1000    1058     968  0  80   0 -  2517 -      pts/0    00:00:00 ps
```

Figure 5.5 – A more detailed view of processes

Here are just a few of the more relevant output fields of the **ps** command:

- **F**: Process flags (for example, **0** – none, **1** – forked, and **4** – superuser privileges)
- **S**: Process status code (for example, **R** – running, **S** – interruptible sleep, and so on)
- **UID**: The username or owner of the process (the user ID)
- **PID**: The process ID
- **PPID**: The process ID of the parent process
- **PRI**: The priority of the process (a higher number means lower priority)
- **SZ**: The virtual memory usage

There are many more such attributes and exploring them all is beyond the scope of this book. For additional information, refer to the **ps** system reference manual (**man ps**).

The **ps** command examples we've used so far have only displayed the processes that are owned by the current Terminal session. This approach, we thought, would add less complexity to analyzing process attributes.

Besides **ps**, another command that's used is **top**, and it provides a live (real-time) view of all the running processes in a system. Its syntax is as follows:

```
top [OPTIONS]
```

Many of the process output fields displayed by the **ps** command are also reflected in the **top** command, albeit some of them with slightly different notations. Let's look at the **top** command and the meaning of the output fields that are displayed. The following command displays a real-time view of running processes:

```
top
```

The output of the preceding command can be seen in the following screenshot:

```
top - 12:02:40 up 32 min,  1 user,  load average: 0.00, 0.00, 0.00
Tasks: 113 total,   1 running, 112 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  6.2 sy,  0.0 ni, 93.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   1976.0 total,   1424.6 free,    222.1 used,    329.3 buff/cache
MiB Swap:   1840.0 total,   1840.0 free,      0.0 used.   1604.8 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
      1 root      20   0  166492  11736   8288 S   0.0   0.6   0:02.13 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par+
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 slub_fl+
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
```

Figure 5.6 – A real-time view of the current processes

Here are some of the output fields, briefly explained:

- **`USER`**: The username or owner of the process
- **`PR`**: The priority of the process (a lower number means higher priority)
- **`NI`**: The nice value of the process (a sort of dynamic/adaptive priority)
- **`VIRT`**: The virtual memory size (in KB) – the total memory used by the process
- **`RES`**: The resident memory size (in KB) – the physical (non-swapped) memory used by the process
- **`SHR`**: The shared memory size (in KB) – a subset of the process memory shared with other processes
- **`S`**: The process' status (for example, **`R`** – running, **`S`** – interruptible sleep, **`I`** – idle, and so on)
- **`%CPU`**: CPU usage (percentage)
- **`%MEM`**: **`RES`** memory usage (percentage)
- **`COMMAND`**: Command name or command line

Each of these fields (and many more) are explained in detail in the `top` system reference manual (`man top`).

Every day, Linux administration tasks frequently use process-related queries based on the preceding presented fields. The *Working with processes* section will explore some of the more common usages of the `ps` and `top` commands, and beyond.

An essential aspect of a process's lifetime is the **status** (or **state**) of the process at any given time and the transition between these states. Both the `ps` and `top` commands provide information about the status of the process via the `s` field. Let's take a closer look at these states.

Process states

During its lifetime, a process may change states according to circumstances. According to the `s` (status) field of the `ps` and `top` commands, a Linux process can have any of the following states:

- `D`: Uninterruptible sleep
- `I`: Idle
- `R`: Running
- `S`: Sleeping (interruptible sleep)
- `T`: Stopped by a job control signal
- `t`: Stopped by the debugger during a trace
- `Z`: Zombie

At a high level, any of these states can be identified with the following process states:

- **Running**: The process is currently running (the `R` state) or is an idle process (the `I` state). In Linux, an idle process is a specific task that's assigned to every processor (CPU) in the system and is scheduled to run only when there's no other process running on the related CPU. The time that's spent on idle tasks accounts for the idle time that's reported by the `top` command.
- **Waiting**: The process is waiting for a specific event or resource. There are two types of waiting states: interruptible sleep (the `S` state) and uninterruptible sleep (the `D` state). Interruptible sleep can be disturbed by specific process signals, yielding further process execution. On the other hand, uninterruptible sleep is a state where the process is blocked in a system call (possibly waiting on some hardware conditions), and it cannot be interrupted.
- **Stopped**: The process has stopped executing, usually due to a specific signal – a job control signal (the `T` state) or a debugging signal (the `t` state).
- **Zombie**: The process is defunct or dead (the `Z` state) – it's terminated without being reaped by its parent. A zombie process is essentially a dead reference for an already terminated process in the system's process table. This will be discussed in more detail in the *Orphan and zombie processes* section.

To conclude our analysis of process states, let's look at the lifetime of a Linux process. Usually, a process starts with a running state (`R`) and terminates once its parent has reaped it from the zombie state (`Z`). The following diagram provides an abbreviated view of the process states and the possible transitions between them:

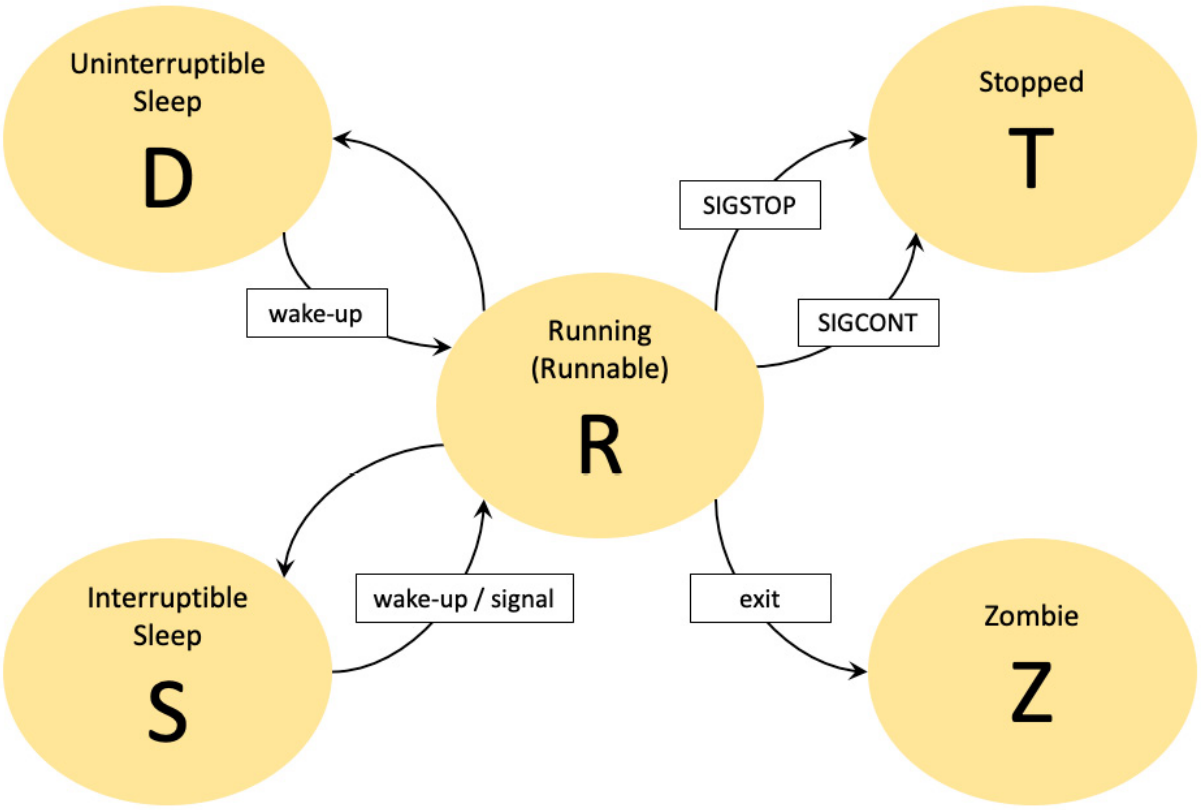

Figure 5.7 – The lifetime of a Linux process

Now that we've introduced processes and provided you with a preliminary idea of their type and structure, we're ready to interact with them. In the following sections, we will explore some standard command-line utilities for working with processes and daemons. Most of these tools operate with input and output data, which we covered in the *Anatomy of a process* section. We'll look at working with processes next.

Working with processes

This section serves as a practical guide to managing processes via resourceful command-line utilities that are used in everyday Linux administration tasks. Some of these tools were mentioned in previous sections (for example, `ps` and `top`) when we covered specific process internals. Here, we will summon most of the knowledge we've gathered so far and take it for a real-world spin by covering some hands-on examples.

Let's start with the `ps` command – the Linux process explorer.

Using the ps command

We described the **ps** command and its syntax in the *Anatomy of a process* section. The following command displays a selection of the current processes running in the system:

```
ps -e | head
```

The **-e** option (or **-A**) selects *all* the processes in the system. The **head** pipe invocation displays only the first few lines (10 by default):

```
packt@neptune:~$ ps -e | head
    PID TTY          TIME CMD
      1 ?        00:00:02 systemd
      2 ?        00:00:00 kthreadd
      3 ?        00:00:00 rcu_gp
      4 ?        00:00:00 rcu_par_gp
      5 ?        00:00:00 slub_flushwq
      6 ?        00:00:00 netns
      7 ?        00:00:01 kworker/0:0-events
      8 ?        00:00:00 kworker/0:0H-events_highpri
     10 ?        00:00:00 mm_percpu_wq
```

Figure 5.8 – Displaying the first few processes

The preceding information may not always be particularly useful. Perhaps we'd like to know more about each process, beyond just the **PID** or **CMD** fields in the **ps** command's output. (We described some of these process attributes in the *Anatomy of a process* section).

The following command lists the processes owned by the current user more elaborately:

```
ps -fU $(whoami)
```

The **-f** option specifies the full-format listing, which displays more detailed information for each process. The **-U $(whoami)** option parameter specifies the current user (**packt**) as the real user (owner) of the processes we'd like to retrieve. In other words, we want to list all the processes we own:

```
packt@neptune:~$ ps -fU $(whoami)
UID          PID    PPID  C STIME TTY          TIME CMD
packt        864       1  0 11:30 ?        00:00:00 /lib/systemd/systemd --user
packt        865     864  0 11:30 ?        00:00:00 (sd-pam)
packt        967     861  0 11:30 ?        00:00:00 sshd: packt@pts/0
packt        968     967  0 11:30 pts/0    00:00:00 -bash
packt       1079     968  0 12:31 pts/0    00:00:00 ps -fU packt
packt@neptune:~$
```

Figure 5.9 – Displaying the processes owned by the current user

There are situations when we may look for a specific process, either for monitoring purposes or to act upon them. Let's take a previous example, where we showcased a long-lived process and wrapped the related command into a simple script. The command is a simple **`while`** loop that runs indefinitely:

```
while true; do x=1; done
```

Using an editor of our preference (for example, **`nano`**), we can create a script file (for example, **`test.sh`**) with the following content:

```
packt@neptune:~$ nano test.sh
packt@neptune:~$ cat test.sh
#!/bin/bash
while true; do x=1; done
```

Figure 5.10 – A simple test script running indefinitely

We can make the test script executable and run it as a background process:

```
chmod +x test.sh
./test.sh &
```

Note the ampersand (**`&`**) at the end of the command, which invokes the background process:

```
packt@neptune:~$ chmod +x test.sh
packt@neptune:~$ ./test.sh &
[1] 1094
```

Figure 5.11 – Running a script as a background process

The background process running our script has a process ID (**`PID`**) of **`1094`**. Suppose we want to find our process by its name (**`test.sh`**). For this, we can use the **`ps`** command with a **`grep`** pipe:

```
ps -ef | grep test.sh
```

The output of the preceding command can be seen in the following screenshot:

```
packt@neptune:~$ ps -ef | grep test.sh
packt       1094     968 99 12:41 pts/0    00:02:28 /bin/bash ./test.sh
packt       1097     968  0 12:44 pts/0    00:00:00 grep --color=auto test.sh
```

Figure 5.12 – Finding a process by name using the ps command

The preceding output shows that our process has a **`PID`** value of **`1094`** and a **`CMD`** value of **`/bin/bash ./test.sh`**. The **`CMD`** field contains the full command invocation of our script, including the command-line parameters.

We should note that the first line of the `test.sh` script contains `#!/bin/bash`, which prompts the OS to invoke `bash` for the script's execution. This line is also known as the **shebang** line, and it has to be the first line in a bash script. To make more sense of the `CMD` field, the command in our case is `/bin/bash` (according to the shebang invocation), and the related command-line parameter is the `test.sh` script. In other words, `bash` executes the `test.sh` script.

The output of the preceding `ps` command also includes our `ps | grep` command's invocation, which is somewhat irrelevant. A refined version of the same command is as follows:

```
ps -ef | grep test.sh | grep -v grep
```

The output of the preceding command can be seen in the following screenshot:

```
packt@neptune:~$ ps -ef | grep test.sh | grep -v grep
packt       1094     968 99 12:41 pts/0    00:04:57 /bin/bash ./test.sh
```

Figure 5.13 – Finding a process by name using the ps command (refined)

The `grep -v grep` pipe filters out the unwanted `grep` invocation from the `ps` command's results.

If we want to find a process based on a process ID (`PID`), we can invoke the `ps` command with the `-p|--pid` option parameter. For example, the following command displays detailed information about our process with `PID` set to `1094` (running the `test.sh` script):

```
packt@neptune:~$ ps -fp 1094
UID          PID    PPID  C STIME TTY          TIME CMD
packt       1094     968 99 12:41 pts/0    00:06:50 /bin/bash ./test.sh
```

Figure 5.14 – Finding a process by PID using the ps command

The `-f` option displays the detailed (*long-format*) process information.

There are numerous other use cases for the `ps` command, and exploring them all is well beyond the scope of this book. The invocations we've enumerated here should provide a basic exploratory guideline for you. For more information, please refer to the `ps` system reference manual (`man ps`).

Using the pstree command

`pstree` shows the running processes in a hierarchical, tree-like view. In some respects, `pstree` acts as a visualizer of the `ps` command. The root of the `pstree` command's output is either the `init` process or the process with the `PID` value specified in the command. The syntax of the `pstree` command is as follows:

```
pstree [OPTIONS] [PID] [USER]
```

The following command displays the process tree of our current Terminal session:

```
pstree $(echo $$)
```

The output of the preceding command can be seen in the following screenshot:

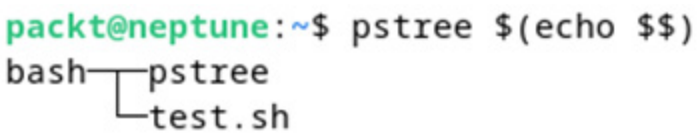

Figure 5.15 – The process tree of the current Terminal session

In the preceding command, **`echo $$`** provides the **`PID`** value of the current Terminal session. **`$$`** is a Bash built-in variable that contains the **`PID`** value of the shell that is running. The **`PID`** value is wrapped as the argument for the **`pstree`** command. To show the related PIDs, we can invoke the **`pstree`** command with the **`-p|--show-pids`** option:

```
pstree -p $(echo $$)
```

The output of the preceding command can be seen in the following screenshot:

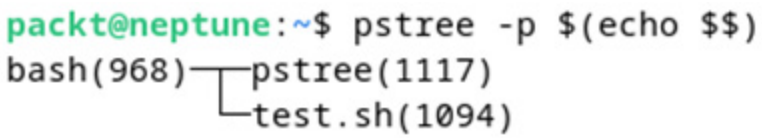

Figure 5.16 – The process tree (along with its PIDs) of the current Terminal session

The following command shows the processes owned by the current user:

```
pstree $(whoami)
```

The output of the preceding command can be seen in the following screenshot:

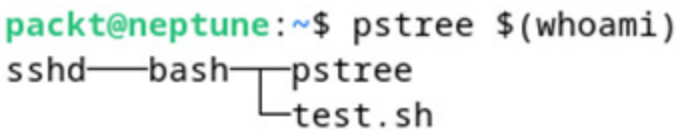

Figure 5.17 – The process tree owned by the current user

For more information about the **`pstree`** command, please refer to the related system reference manual (**`man pstree`**).

Using the top command

When it comes to monitoring processes in real time, the **`top`** utility is among the most common tool to be used by Linux administrators. The related command-line syntax is as follows:

```
top [OPTIONS]
```

The following command displays all the processes currently running in the system, along with real-time updates (on memory, CPU usage, and so on):

```
top
```

Pressing *Q* will exit the `top` command. By default, the `top` command sorts the output by CPU usage (shown in the `%CPU` field/column).

We can also choose to sort the output of the `top` command by a different field. While `top` is running, press *Shift* + *F* (`F`) to invoke interactive mode.

Using the arrow keys, we can select the desired field to sort by (for example, `%MEM`), then press *S* to set the new field, followed by *Q* to exit interactive mode. The alternative to interactive mode sorting is invoking the `-o` option parameter of the `top` command, which specifies the sorting field.

For example, the following command lists the top 10 processes, sorted by CPU usage:

```
top -b -o %CPU | head -n 17
```

Similarly, the following command lists the top 10 processes, sorted by CPU and memory usage:

```
top -b -o +%MEM | head -n 17
```

The `-b` option parameter specifies the batch mode operation (instead of the default interactive mode). The `-o +%MEM` option parameter indicates the additional (`+`) sorting field (`%MEM`) in tandem with the default `%CPU` field. The `head -n 17` pipe selects the first 17 lines of the output, accounting for the seven-line header of the `top` command:

```
packt@neptune:~$ top -b -o +%MEM | head -n 17
top - 12:59:24 up  1:29,  1 user,  load average: 1.00, 0.99, 0.71
Tasks: 115 total,   2 running, 113 sleeping,   0 stopped,   0 zombie
%Cpu(s):100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   1976.0 total,   1417.6 free,    211.7 used,    346.6 buff/cache
MiB Swap:   1840.0 total,   1840.0 free,      0.0 used.   1612.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1013 root      20   0  500400  64388  22584 S   0.0   3.2   0:00.74 fwupd
    665 root      20   0  727508  40732  20716 S   0.0   2.0   0:01.18 snapd
    421 root      rt   0  289312  27360   9072 S   0.0   1.4   0:00.49 multipa+
    720 root      20   0  109748  21292  13136 S   0.0   1.1   0:00.05 unatten+
    661 root      20   0   32652  18932  10284 S   0.0   0.9   0:00.04 network+
    381 root      19  -1   47856  15664  14572 S   0.0   0.8   0:00.12 systemd+
    669 root      20   0  392564  12880  10792 S   0.0   0.6   0:00.06 udisksd
    641 systemd+  20   0   25260  12452   8516 S   0.0   0.6   0:00.04 systemd+
    705 root      20   0  317948  11964  10060 S   0.0   0.6   0:00.03 ModemMa+
      1 root      20   0  166492  11736   8288 S   0.0   0.6   0:02.16 systemd
```

Figure 5.18 – The top 10 processes sorted by CPU and memory usage

The following command lists the top five processes by CPU usage, owned by the current user (`packt`):

```
top -u $(whoami) -b -o %CPU | head -n 12
```

The `-u $(whoami)` option parameter specifies the current user for the `top` command.

With the `top` command, we can also monitor specific processes using the `-p` PID option parameter. For example, the following command monitors our test process (with PID `243436`):

```
top -p 1094
```

The output of the preceding command can be seen in the following screenshot:

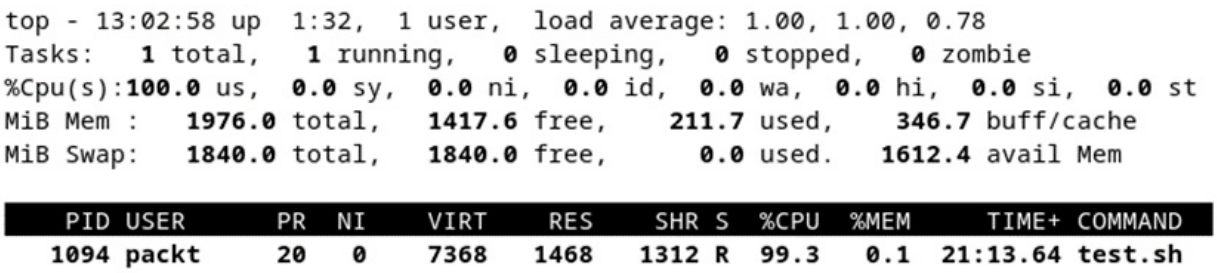

Figure 5.19 – Monitoring a specific PID with the top command

We may choose to *kill* the process by pressing *K* while using the `top` command. We'll get prompted for this by the PID of the process we want to terminate:

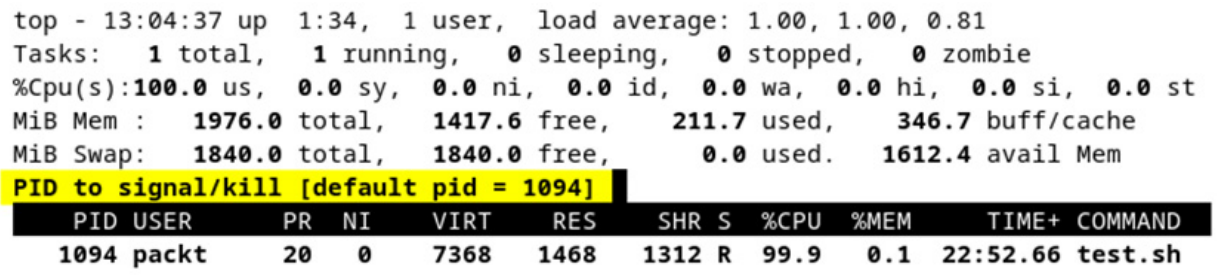

Figure 5.20 – Killing a process with the top command

The `top` utility can be used in many creative ways. We hope that the examples we've provided in this section have inspired you to explore further use cases based on specific needs. For more information, please refer to the system reference manual for the `top` command (`man top`).

Using the kill and killall commands

We use the `kill` command to terminate processes. The command's syntax is as follows:

```
kill [OPTIONS] [ -s SIGNAL | -SIGNAL ] PID [...]
```

The **kill** command sends a *signal* to a process, attempting to stop its execution. When no signal is specified, **SIGTERM** (**15**) is sent. A signal can either be specified by the signal's name without the **SIG** prefix (for example, **KILL** for **SIGKILL**) or by value (for example, **9** for **SIGKILL**).

The **kill -l** and **kill -L** commands provide a full list of signals that can be used in Linux:

```
packt@neptune:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Figure 5.21 – The Linux signals

Each signal has a numeric value, as shown in the preceding output. For example, **SIGKILL** equals **9**. The following command will kill our test process (with PID **243436**):

```
kill -9 1094
```

The following command will also do the same as the preceding command:

```
kill -KILL 1094
```

In some scenarios, we may want to kill multiple processes in one go. The **killall** command comes to the rescue here. The syntax for the **killall** command is as follows:

```
killall [OPTIONS] [ -s SIGNAL | -SIGNAL ] NAME...
```

killall sends a signal to all the processes running any of the commands specified. When no signal is specified, **SIGTERM** (**15**) is sent. A signal can either be specified by the signal name without the **SIG** prefix (for example, **TERM** for **SIGTERM**) or by value (for example, **15** for **SIGTERM**).

For example, the following command terminates all the processes running the **test.sh** script:

```
killall -e -TERM test.sh
```

The output of the preceding command can be seen in the following screenshot:

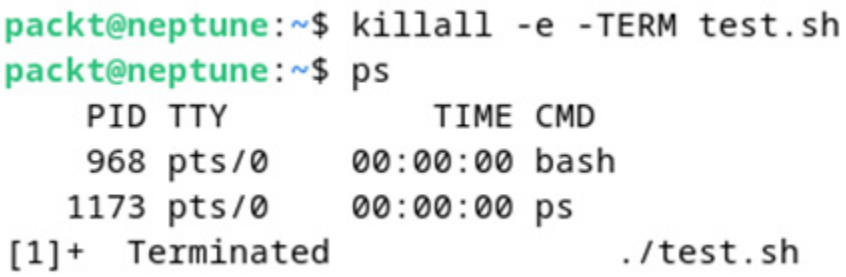

Figure 5.22 – Terminating multiple processes with killall

Killing a process will usually remove the related reference from the system process table. The terminated process won't show up anymore in the output of **ps**, **top**, or similar commands.

For more information about the **kill** and **killall** commands, please refer to the related system reference manuals (**man kill** and **man killall**).

Using the pgrep and pkill commands

pgrep and **pkill** are pattern-based lookup commands for exploring and terminating running processes. They have the following syntax:

```
pgrep [OPTIONS] PATTERN
pkill [OPTIONS] PATTERN
```

pgrep iterates through the current processes and lists the PIDs that match the selection pattern or criteria. Similarly, **pkill** terminates the processes that match the selection criteria.

The following command looks for our test process (**test.sh**) and displays the **PID** value if the related process is found. Start the process again before using the following command as we killed it in the previous section. This will lead to a different **PID** value:

```
pgrep -f test.sh
```

The output of the preceding command can be seen in the following screenshot:

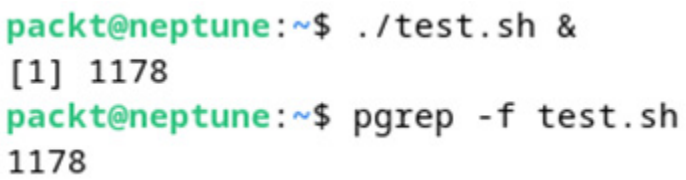

Figure 5.23 – Looking for a PID based on name using pgrep

The **-f|--full** option enforces a full name match of the process we're looking for. We may use **pgrep** in tandem with the **ps** command to get more detailed information about the process, like so:

```
pgrep -f test.sh | xargs ps -fp
```

The output of the preceding command can be seen in the following screenshot:

```
packt@neptune:~$ pgrep -f test.sh | xargs ps -fp
UID          PID    PPID  C STIME TTY          TIME CMD
packt       1178     968 99 13:15 pts/0    00:02:07 /bin/bash ./test.sh
```

Figure 5.24 – Chaining pgrep and ps for more information

In the preceding one-liner, we piped the output of the **`pgrep`** command (with PID **`243436`**) to the **`ps`** command, which has been invoked with the **`-f`** (long-format) and **`-p|--pid`** options. The **`-p`** option parameter gets the piped PID value.

The **`xargs`** command takes the input from the **`pgrep`** command and converts it into an argument for the **`ps`** command. Thus, when piping from **`pgrep`** to **`ps`**, the output of the first command was automatically converted as the argument for the second command. By default, **`xargs`** reads the standard input.

To terminate our **`test.sh`** process, we simply invoke the **`pkill`** command, as follows:

```
pkill -f test.sh
```

The preceding command will *silently* kill the related process, based on the full name lookup enforced by the **`-f|--full`** option. To get some feedback from the action of the **`pkill`** command, we need to invoke the **`-e|--echo`** option, like so:

```
pkill -ef test.sh
```

The output of the preceding command can be seen in the following screenshot:

```
packt@neptune:~$ pkill -ef test.sh
test.sh killed (pid 1178)
[1]+  Terminated              ./test.sh
```

Figure 5.25 – Killing a process by name using pkill

For more information, please refer to the **`pgrep`** and **`pkill`** system reference manuals (**`man pgrep`** and **`man pkill`**).

This section covered some command-line utilities that are frequently used in everyday Linux administration tasks involving processes. Keep in mind that in Linux, most of the time, there are many ways to accomplish a specific task. We hope that the examples in this section will help you come up with creative methods and techniques for working with processes.

Next, we'll look at some common ways of interacting with daemons.

Working with daemons

As noted in the introductory sections, daemons are a special breed of background process. Consequently, the vast majority of methods and techniques for working with processes also apply to daemons. However, there are specific commands that strictly operate on daemons when it comes to managing (or controlling) the lifetime of the related processes.

As noted in the *Introducing daemons* section, daemon processes are controlled by shell scripts, usually stored in the **`/etc/init.d/`** or **`/lib/systemd/`** system directories, depending on the Linux platform. On legacy Linux systems (for example, RHEL 6) and Ubuntu (even in the latest distros), the daemon script files are stored in **`/etc/init.d/`**. On RHEL 7/Ubuntu 18.04 and newer platforms, they are typically stored in **`/lib/systemd/`**. Feel free to do a listing of those two directories to see the contents.

The location of the daemon files and the daemon command-line utilities largely depends on the **`init`** initialization system and service manager. In *The init process* section, we briefly mentioned a variety of **`init`** systems across Linux distributions. To illustrate the use of daemon control commands, we will explore the **`init`** system called **`systemd`**, which is extensively used across various Linux platforms.

Working with systemd daemons

The **`init`** system's essential requirement is to initialize and orchestrate the launch and startup dependencies of various processes when the Linux kernel is booted. These processes are also known as **userland** or **user processes**. The **`init`** engine also controls the services and daemons while the system is running.

Over the last few years, most Linux platforms have transitioned to **`systemd`** as their default **`init`** engine. Due to its extensive adoption, being familiar with **`systemd`** and its related command-line tools is of paramount importance. With that in mind, this section's primary focus is on **`systemctl`** – the central command-line utility for managing **`systemd`** daemons.

The syntax of the **`systemctl`** command is as follows:

```
systemctl [OPTIONS] [COMMAND] [UNITS...]
```

The actions that are invoked by the **`systemctl`** command are directed at units, which are system resources that are managed by **`systemd`**. Several unit types are defined in **`systemd`** (for example, service, mount, socket, and so on). Each of these units has a corresponding file. These file types are inferred from the suffix of the related filename; for example, **`httpd.service`** is the service unit file of

the Apache web service (daemon). For a comprehensive list of **systemd** units and detailed descriptions of them, please refer to the **systemd.unit** system reference manual (**man systemd.unit**).

The following command enables a daemon (for example, **httpd**, the web server) to start at boot:

```
sudo systemctl enable httpd
```

Typically, invoking **systemctl** commands requires superuser privileges. We should note that **systemctl** does not require the **.service** suffix when we're targeting service units. The following invocation is also acceptable:

```
sudo systemctl enable httpd.service
```

The command to disable the **httpd** service from starting at boot is as follows:

```
sudo systemctl disable httpd
```

To query the status of the **httpd** service, we can run the following command:

```
sudo systemctl status httpd
```

Alternatively, we can check the status of the **httpd** service with the following command:

```
sudo systemctl is-active httpd
```

The following commands stop or start the **httpd** service:

```
sudo systemctl stop httpd
sudo systemctl start httpd
```

For more information on **systemctl**, please refer to the related system reference manual (**man systemctl**). For more information about **systemd** internals, please refer to the corresponding reference manual (**man systemd**).

Working with processes and daemons is a constant theme of everyday Linux administration tasks. Mastering the related command-line utilities is an essential skill for any seasoned user. Yet, a running process or daemon should also be considered in relationships with other processes or daemons running either locally or on remote systems. The way processes communicate with each other could be a slight mystery to some. We will address this in the next section, in which we will explain how inter-process communication works.

Explaining inter-process communication

Inter-process communication (**IPC**) is a way of interacting between processes using a shared mechanism or interface. In this section, we will take a short theoretical approach to exploring various

communication mechanisms between processes. For more details on this matter and some of the mechanisms used, head to *Chapter 8*, *Linux Shell Scripting*.

Linux processes can typically share data and synchronize their actions via the following interfaces:

- **Shared storage** (**files**): In its simplest form, the shared storage of an IPC mechanism can be a simple file that's been saved to disk. The producer then writes to a file while the consumer reads from the same file. In this simple use case, the obvious challenge is the integrity of the read/write operations due to possible race conditions between the underlying operations. To avoid race conditions, the file must be locked during write operations to prevent overlapping I/O with another read or write action. To keep things simple, we're not going to resolve this problem in our naive examples, but we thought it's worth calling it out.
- **Shared memory**: Processes in Linux typically have separate address spaces. A process can only access data in the memory of another process if the two share a common memory segment where such data would be stored. Linux provides at least a couple of **application programming interfaces** (**APIs**) to programmatically define and control shared memory between processes: a legacy System V API and the more recent POSIX API, for example. Both these APIs are written in C, though the implementation of the producer and consumer mockups is beyond the scope of this book. However, we can closely match the shared memory approach by using the **`/dev/shm`** temporary file storage system, which uses the system's RAM as its backing store (that is, RAM disk).

With **`/dev/shm`** being used as shared memory, we can reuse our producer-consumer model from the example in the previous point on *Shared storage*, where we simply point the storage file to **`/dev/shm/storage`**.

The shared memory and shared storage IPC models may not perform well with large amounts of data, especially massive data streams. The alternative would be to use IPC channels, which can be enabled through the pipe, message queue, or socket communication layers.

- **Named and unnamed pipes**: **Unnamed** or **anonymous pipes**, also known as **regular pipes**, feed the output of a process to the input of another one. Using our producer-consumer model, the simplest way to illustrate an unnamed pipe as an IPC mechanism between the two processes would be to do the following:

```
producer.sh | consumer.sh
```

The key element of the preceding code is the pipe (`|`) symbol. The left-hand side of the pipe produces an output that's fed directly to the right-hand side of the pipe for consumption.

Named pipes, also known as **First-In, First-Outs** (**FIFOs**), are similar to traditional (unnamed) pipes but substantially different in terms of their semantics. An unnamed pipe only persists for as long as the related process is running. However, a named pipe has backing storage and will last as long as the system is up, regardless of the running status of the processes attached to the related IPC channel. Typically, a named pipe acts as a file, and it can be deleted when it's no longer being used.

- **Message queues**: A message queue is an asynchronous communication mechanism that's typically used in a distributed system architecture. Messages are written and stored in a queue until they are processed and eventually deleted. A message is written (published) by a producer and is processed only once, typically by a single consumer. At a very high level, a message has a sequence, a payload, and a type. Message queues can regulate the retrieval (order) of messages (for example, based on priority or type):

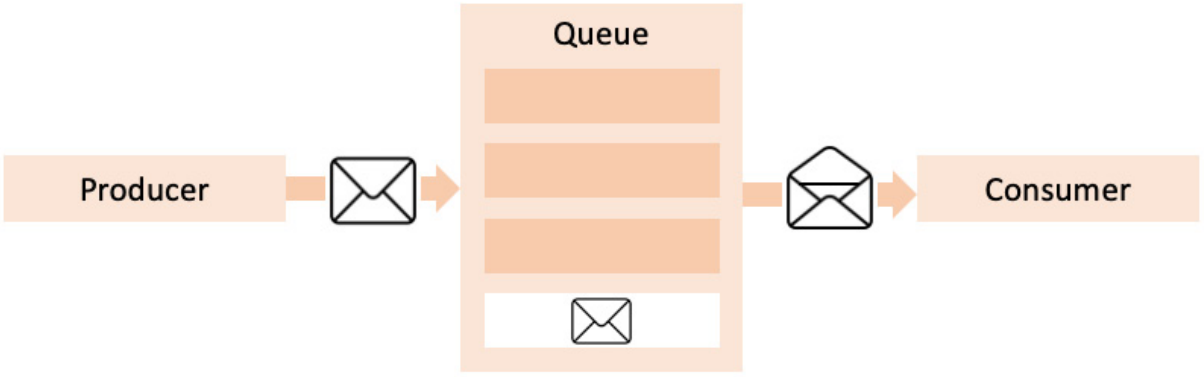

Figure 5.26 – Message queue (simplified view)

A detailed analysis of message queues or a mock implementation thereof is far from trivial, and it's beyond this chapter's scope. There are numerous open source message queue implementations available for most Linux platforms (RabbitMQ, ActiveMQ, ZeroMQ, MQTT, and so on).

IPC mechanisms based on message queues and pipes are unidirectional. One process writes the data; another one reads it. There are bidirectional implementations of named pipes, but the complexities involved would negatively impact the underlying communication layer. For bidirectional communication, you can think of using socket-based IPC channels (detailed in *Chapter 8*, *Linux Shell Scripting*).

- **Sockets**: There are two types of IPC socket-based facilities:
    - **IPC sockets**: Also known as Unix domain sockets, IPC sockets use a local file as a socket address and enable bidirectional communication between processes on the same host.
    - **Network sockets**: **Transport Control Protocol** (**TCP**) and **User Datagram Protocol** (**UDP**) sockets. They extend the IPC data connectivity layer beyond the local machine via TCP/UDP networking.

Apart from the obvious implementation differences, the IPC socket's and network socket's data communication channels behave the same.

Both sockets are configured as streams, support bidirectional communication, and emulate a client/server pattern. The socket's communication channel is active until it's closed on either end, thereby breaking the IPC connection.

- **Signals**: In Linux, a signal is a one-way asynchronous notification mechanism that's used in response to a specific condition. A signal can act in any of the following directions:
    - From the Linux kernel to an arbitrary process
    - From process to process
    - From a process to itself

We mentioned at the beginning of this section that signals are yet another IPC mechanism. Indeed, they are a somewhat limited form of IPC in the sense that through signals, processes can coordinate synchronization with each other. But signals don't carry any data payloads. They simply notify processes about events, and processes may choose to take specific actions in response to these events.

In the next section, we will detail working with signals in Linux.

Working with signals

Signals typically alert a Linux process about a specific event, such as a segmentation fault (`SIGSEGV`) that's raised by the kernel or execution being interrupted (`SIGINT`) by the user pressing *Ctrl + C*. In Linux, processes are controlled via signals. The Linux kernel defines a few dozen signals. Each signal has a corresponding non-zero positive integer value.

The following command lists all the signals that have been registered in a Linux system:

```
kill -l
```

The output of the preceding command can be seen back in *Figure 5.21*. From the output, `SIGHUP`, for example, has a signal value of `1`, and it's invoked by a Terminal session to all its child processes when it exits. `SIGKILL` has a signal value of `9` and is most commonly used for terminating processes. Processes can typically control how signals are handled, except for `SIGKILL` (`9`) and `SIGSTOP` (`19`), which always end or stop a process, respectively.

Processes handle signals in either of the following fashions:

- Perform the default action implied by the signal; for example, stop, terminate, core-dump a process, or do nothing.
- Perform a custom action (except for **`SIGKILL`** and **`SIGSTOP`**). In this case, the process catches the signal and handles it in a specific way.

When a program implements a custom handler for a signal, it usually defines a signal handler function that alters the execution of the process, as follows:

- When the signal is received, the process' execution is interrupted at the current instruction
- The process' execution immediately jumps to the signal-handler function
- The signal handler function runs
- When the signal handler function exits, the process resumes execution, starting from the previously interrupted instruction

Here's some brief terminology related to signals:

- A signal is raised by the process that generates it
- A signal is caught by the process that handles it
- A signal is ignored if the process has a corresponding **no-operation** or **no-op** (**NOOP**) handler

- A signal is handled if the process implements a specific action when the signal is caught

Out of all the signals, `SIGKILL` and `SIGSTOP` are the only ones that cannot be caught or ignored.

Let's explore a few use cases for handling signals:

- When the kernel raises a **`SIGKILL`**, **`SIGFPE`** (floating-point exception), **`SIGSEGV`** (segmentation fault), **`SIGTERM`**, or similar signals, typically, the process that receives the signal immediately terminates execution and may generate a core dump – the image of the process that's used for debugging purposes.
- When a user types *Ctrl + C* – otherwise known as an **interrupt character** (**INTR**) – while a foreground process is running, a **`SIGINT`** signal is sent to the process. The process will terminate unless the underlying program implements a special handler for **`SIGINT`**.
- Using the **`kill`** command, we can send a signal to any process based on its PID. The following command sends a **`SIGHUP`** signal to a Terminal session with a PID of **`3741`**:

```
kill -HUP 3741
```

In the preceding command, we can either specify the signal value (for example, `1` for **`SIGHUP`**) or just the signal name without the **`SIG`** prefix (for example, **`HUP`** for **`SIGHUP`**).

With **`killall`**, we can signal that multiple processes are running a specific command (for example, **`test.sh`**). The following command terminates all processes running the **`test.sh`** script and outputs the result to the console (via the **`-e`** option):

```
killall -e -TERM test.sh
```

The output of this command can be seen in *Figure 5.22*.

Linux processes and signals are a vast domain. The information we've provided here is far from a comprehensive guide on the topic. We hope that this short spin and hands-on approach to presenting some common use cases has inspired you to take on and possibly master more challenging issues.

Summary

A detailed study of Linux processes and daemons could be a major undertaking. Where worthy volumes on the topic have admirably succeeded, a relatively brief chapter may pale in comparison. Yet in this chapter, we tried to put on a real-world, down-to-earth, practical coat on everything we've considered to make up for our possible shortcomings in the abstract or scholarly realm.

At this point, we hope you are comfortable working with processes and daemons. The skills you've gathered so far should include a relatively good grasp of process types and internals, with a reasonable understanding of process attributes and states. Special attention has been paid to inter-process communication mechanisms, and signals in particular. For each of these topics, we will take

a more detailed approach in *Chapter 8*. For now, we consider the information we've provided to be sufficient for understanding how inter-process communication works.

The next chapter will take our journey further into working with Linux disks and filesystems. We'll explore the Linux storage, disk partitioning, and **Logical Volume Management** (**LVM**) concepts. Rest assured that everything we've learned so far will be immediately put to good use in the chapters that follow.

Questions

If you managed to skim through some parts of this chapter, you might want to recap a few essential details about Linux processes and daemons:

1. Think of a few process types. How would they compare to each other?
2. Think of the anatomy of a process. Can you come up with a few essential process attributes (or fields in the `ps` command-line output) that you may look for when inspecting processes?

Hint: What would be relevant for you, except CPU, RAM, or disk usage, for example?

3. Can you think of a few process states and some of the dynamics or possible transitions between them?
4. If you are looking for a process that takes up most of the CPU on your system, how would you proceed?
5. Can you write a simple script and make it a long-lived background process?

Hint: Take a peek at *Chapter 8*, where we will teach you how to create and use shell scripts.

6. Enumerate at least four process signals that you can think of. When or how would those signals be invoked?

Hint: Use the `kill -l` command. For more information, read the manual.

7. Think of a couple of IPC mechanisms. Try to come up with some pros and cons for them.

Hint: The information in *Chapter 8* could help you.

Further reading

For more information about what was covered in this chapter, you can refer to the following Packt titles:

- *Linux Administration Best Practices*, by Scott Alan Miller
- *Linux Service Management Made Easy with systemd*, by Donald A. Tevault

Part 2:Advanced Linux Administration

In this second part, you will learn about advanced Linux system administration tasks, including working with disks and configuring networking, hardening Linux security, and system-specific troubleshooting and diagnostics.

This part has the following chapters:

6

Working with Disks and Filesystems

In this chapter, you will learn how to manage disks and filesystems, how to use the **Logical Volume Management** (**LVM**) system, and how to mount and partition the hard drive, as well as gain an understanding of storage in Linux. You will also learn how to partition and format a disk, as well as how to create logical volumes, and you will gain a deeper understanding of filesystem types. In this chapter, we're going to cover the following main topics:

- Understanding devices in Linux
- Understanding filesystem types in Linux
- Understanding disks and partitions
- Introducing LVM in Linux

Technical requirements

A basic knowledge of disks, partitions, and filesystems is preferred. No other special technical requirements are needed, just a working installation of Linux on your system. We will mainly use Ubuntu or Debian for this chapter's exercises. All the commands used in this chapter can be replicated on any Linux distribution, even if you don't use Debian or Ubuntu.

Understanding devices in Linux

As already stated on several occasions in this book, everything in Linux is a file. This also includes devices. **Device files** are special files in Unix and Linux operating systems. Those special files are interfaces to device drivers, and they are present in the filesystem as a regular file.

With no further ado, let's see how Linux abstraction layers work. This will give you an overview of how hardware and software are related and interconnected.

Linux abstraction layers

Now is as good a time as any to discuss Linux system abstraction layers and how devices fit into the overall picture. Any computer is generally organized into two layers (or levels) – the hardware and the software levels:

- **Hardware level**: This level contains the hardware components of your machine, such as the memory (RAM), **central processing unit** (**CPU**), and devices, including disks, network interfaces, ports, and controllers.
- **Software level**: For all these hardware components to work, the operating system (Linux, in our case) uses **abstraction layers**. Those layers exist in the **kernel**, which is the main software component of Linux. Without diving into more information, it is sufficient for you to know that Linux has these layers that are responsible for accessing low-level resources and providing the specific drivers for different hardware components. When the computer is booted up, the Linux kernel is loaded from the disk into the system's memory (RAM). Thus, inside the memory, there will be two separate regions, called **kernel space** and **user space**, and this would be the **software level**:
    - The kernel is the beating heart of the Linux operating system. The kernel resides inside the memory (RAM) and manages all the hardware components. It is the *interface* between the software and hardware on your Linux system.
    - The user space level is the level where user processes are executed. As presented in *Chapter 5*, *Working with Processes, Daemons, and Signals*, a process is a running instance of a program.

Where are devices in this grand scheme of things? Devices are managed by the *kernel*. To sum up, the kernel is in charge of managing processes, system calls, memory, and devices. When dealing with devices, the kernel manages **device drivers**, which are the interface between hardware components and software. All devices are accessible only in kernel mode, for a more secure and streamlined operation.

How does all this work? Well, the memory, known as RAM, consists of cells that are used to store information temporarily. Those cells are accessed by different programs that are executed and function as an intermediary between the CPU and the storage. The speeds of accessing memory are very high to secure a seamless process of execution. The management of user processes inside the user space is the kernel's job. The kernel makes sure that none of the processes will interfere with each other. The kernel space is usually accessed only by the kernel, but there are times when user processes need to access this space. This is done through **system calls**. A system call is the way a user process requests a kernel service through an active process inside the kernel space, for anything such as **input/output** (**I/O**) requests to internal or external devices. All those requests transfer data to and from the CPU, through RAM, to get the job done.

In the following section, we will introduce you to the naming convention in Linux and how device files are managed.

Device files and naming conventions

After seeing how the abstraction layers work, you may be wondering how Linux manages devices. Well, it does that with the help of **userspace /dev** (**udev**), which is a device manager for the kernel. It works with **device nodes**, which are special files (also called **device files**) that are used as an interface to the driver.

Device files in Linux

udev runs as a daemon that listens to the user space calls that the kernel is sending, so it is aware of what kinds of devices are used and how they are used. The daemon is called **udevd** and its configurations are currently available under **/etc/udev/udev.conf**. You can concatenate the **/etc/udev/udev.conf** file to see its contents by running the following command:

```
cat /etc/udev/udev.conf
```

Each Linux distribution has a default set of rules that governs **udevd**. Those rules are normally stored under the **/etc/udev/rules.d/** directory, as shown in the following screenshot:

```
packt@neptune:~$ ls -l /etc/udev/rules.d/
total 68
-rw-r--r-- 1 root root 63312 Aug  9  2022 70-snap.snapd.rules
-rw-r--r-- 1 root root   148 Feb 27 08:50 ubuntu--vg-ubuntu--lv.rules
```

Figure 6.1 – udevd rules location

NOTE

*The kernel sends calls for events using the **Netlink** socket. The netlink socket is an interface for inter-process communication that's used for both userspace and kernel space processes alike.*

The **/dev** directory is the interface between user processes and devices managed by the kernel. If you were to use the **ls -la /dev** command, you would see a lot of files inside, each with different names. If you were to do a long listing, you would see different file types. Some of the files will start with the letters *b* and *c*, but the letters *p* and *s* may also be present, depending on your system. Files starting with those letters are device files. The ones starting with *b* are **block devices**, and those starting with the letter *c* are **character devices**, as shown in the following screenshot:

```
packt@neptune:~$ ls -la /dev
total 4
drwxr-xr-x 20 root root        4080 Mar 16 07:09 .
drwxr-xr-x 19 root root        4096 Feb 27 08:54 ..
crw-r--r--  1 root root     10, 235 Mar 16 07:09 autofs
drwxr-xr-x  2 root root         360 Mar 16 07:09 block
crw-rw----  1 root disk     10, 234 Mar 16 07:09 btrfs-control
drwxr-xr-x  3 root root          60 Mar 16 07:09 bus
drwxr-xr-x  2 root root        3500 Mar 16 07:09 char
crw--w----  1 root tty        5,   1 Mar 16 07:09 console
lrwxrwxrwx  1 root root          11 Mar 16 07:09 core -> /proc/kcore
drwxr-xr-x  3 root root          60 Mar 16 07:09 cpu
crw-------  1 root root     10, 124 Mar 16 07:09 cpu_dma_latency
crw-------  1 root root     10, 203 Mar 16 07:09 cuse
drwxr-xr-x  7 root root         140 Mar 16 07:09 disk
brw-rw----  1 root disk    253,   0 Mar 16 07:09 dm-0
drwxr-xr-x  2 root root          60 Mar 16 07:09 dma_heap
drwxr-xr-x  3 root root          80 Mar 16 07:09 dri
crw-------  1 root root     10, 126 Mar 16 07:09 ecryptfs
crw-rw----  1 root video    29,   0 Mar 16 07:09 fb0
lrwxrwxrwx  1 root root          13 Mar 16 07:09 fd -> /proc/self/fd
```

Figure 6.2 – Device files inside the /dev directory

Let's see how disk devices are presented inside the `/dev` directory. But first, we've provided a few words about our working setup in the following note.

IMPORTANT NOTE

For most of the exercises in this book, we will be using virtual machines with planet names as hostnames, running different Linux-based operating systems. For example, ***`neptune`*** *is running on Ubuntu 22.04.2 LTS Server, so when you see the* ***`neptune`*** *hostname on the shell's prompt, you will know we are on an Ubuntu-based system. We also have* ***`jupiter`****, running on the openSUSE 15.4 Leap server,* ***`saturn`*** *running on Fedora 37 Workstation,* ***`venus`*** *running on AlmaLinux, and* ***`mars`*** *running on a Debian 11.6 server. Inside a virtual machine, device drivers are presented with different names than on bare-metal systems. We will provide details when we discuss device naming conventions in the following section. For some examples, though, which are marked accordingly, we will also use our primary workstation, which is running on Debian 12 GNU/Linux.*

As shown in *Figure 6.3*, the disk device, `sda`, is represented as a block device. Block devices have a fixed size that can easily be indexed. Character devices, on the other hand, can be accessed using data streams, as they don't have a size like block devices. For example, printers are represented as character devices. In the following screenshot, `sg0` is an SCSI generic device, and not assigned to any disks in our case. We used our primary workstation running on Debian GNU/Linux and the device presented as `sda` is an external USB device:

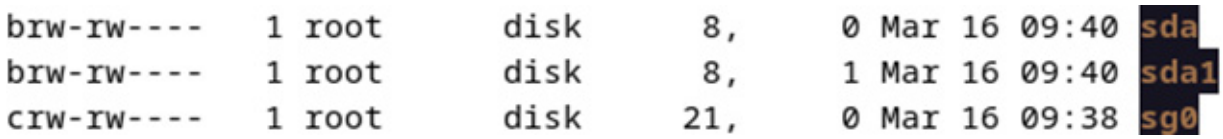

Figure 6.3 – Disk drives inside the /dev directory

In comparison, when listing devices from our `neptune` virtual machine, we will have the output presented in the following screenshot:

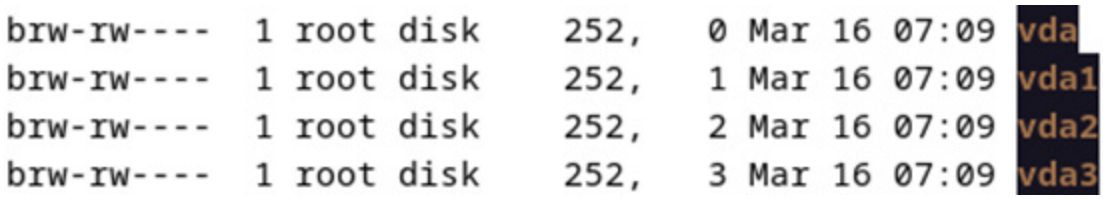

Figure 6.4 – Virtual devices inside a virtual machine

Device blocks presented with `vdaX` are virtual devices inside the virtual machine. You will learn more about virtual machines in *Chapter 11*, *Working with Virtual Machines*.

But for now, let's find out more about device naming conventions in Linux.

Understanding device naming conventions

Linux uses a device naming convention that makes device management easier and more consistent throughout the Linux ecosystem. `udev` uses several specific naming schemes that, by default, assign fixed names to devices. Those names are standardized for device categories. For example, when naming network devices, the kernel uses information compiled from sources such as firmware, topology, and location. On a Red Hat-based system, five schemes are used for naming a network interface, and we encourage you to look at these on the Red Hat customer portal official documentation website: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9.

On a Debian-based system, the naming convention is similar in that it's based on hardware buses' names for predictability. This is similar to all modern Linux-based operating systems.

You could also check what `udev` rules are active on your system. On Debian and Red Hat-based distributions, they are stored in the `/lib/udev/rules.d/` directory.

When it comes to hard drives or external drives, the conventions are more streamlined. Here are some examples:

- **For classic IDE drivers used for ATA drives**: `hda` (the master device), `hdb` (the slave device on the first channel), `hdc` (the master device on the second channel), and `hdd` (the slave device on the second channel)
- **For NVMe drivers**: `nvme0` (the first device controller – character device), `nvme0n1` (first namespace – block device), and `nvme0n1p1` (first namespace, first partition – block device)

- **For MMC drivers**: `mmcblk` (for SD cards using eMMC chips), `mmcblk0` (first device), and `mmcblk0p1` (first device, first partition)
- **For SCSI drivers used for modern SATA or USB**: `sd` (for mass storage devices), `sda` (for the first registered device), `sdb` (for the second registered device), `sdc` (for the third registered device), and so on, and `sg` (for generic SCSI layers – character device)

The devices that we are most interested in regarding this chapter are the mass storage devices. Those devices are usually **hard disk drives** (**HDDs**) or **solid-state drives** (**SSDs**), which are used inside your computer to store data. These drives are most likely divided into partitions with a specific structure provided by the filesystem. We talked a little bit about filesystems earlier in this book in *Chapter 2*, *The Linux Shell and Filesystem*, when we referred to the Linux directory structure, but now, it is time to get into more details about filesystem types in Linux.

Understanding filesystem types in Linux

When talking about physical media, such as hard drives or external drives, we are *not* referring to the directory structure. Here, we are talking about the structures that are created on the physical drive when formatting and/or partitioning it. These structures, depending on their type, are known as filesystems, and they determine how the files are managed when stored on the drive.

There are several types of filesystems, some being native to the Linux ecosystem, while others are not, such as specific Windows or macOS filesystems. In this section, we will describe only the Linux-native filesystems.

The most widely used filesystems in Linux are the **extended filesystems**, known as `Ext`, `Ext2`, `Ext3`, and `Ext4`, the `XFS` filesystem, `ZFS`, and `btrfs` (short for **B-tree filesystem**). Each of these have their strengths and weaknesses, but they are all able to do the job they were designed for. The extended filesystems are the ones that were most widely used in Linux, and they have proven trustworthy all this time. `Ext4`, the latest iteration, is similar to `Ext3`, but better, with improved support for larger files, fragmentation, and performance. The `Ext3` filesystem uses 32-bit addressing, while `Ext4` uses 48-bit addressing, thus supporting files up to 16 TB in size. It also offers support for unlimited subdirectories as `Ext3` only supports 32k subdirectories. Also, support for extended timestamps was added in `Ext4`, offering two more bits for up to the year 2446 AD, and online defragmentation at the kernel level.

Nonetheless, `Ext4` is not a truly next-gen filesystem; rather, it is an improved, trustworthy, robust, and stable *workhorse* that failed the data protection and integrity test. Its journaling system is not suitable for detecting and repairing data corruption and degradation. That is why other filesystems, such as `XFS` and `ZFS`, started to resurface by being used in Red Hat Enterprise Linux, starting from version 7 (`XFS`) and in Ubuntu since version 16.04 (`ZFS`).

The case of `btrfs` is somewhat controversial. It is considered a modern filesystem, but it is still used as a single-disk filesystem and not used in multiple disk volume managers due to several performance issues compared to other filesystems. It is used in SUSE Linux Enterprise and openSUSE, is no longer supported by Red Hat, and has been voted as the future default filesystem in Fedora, starting with version 33.

Here are some more details on the major filesystem features:

- **Ext4**: The `Ext4` filesystem was designed for Linux right from the outset. Even though it is slowly being replaced with other filesystems, this one still has powerful features. It offers block size selection, with values between 512 and 4,096 bytes. There is also a feature called inode reservation, which saves a couple of inodes when you create a directory, for improved performance when creating new files.

  The layout is simple, written in **little-endian** order (for more details on this, visit https://www.section.io/engineering-education/what-is-little-endian-and-big-endian/), with block groups containing inode data for lower access times. Each file has data blocks pre-allocated for reduced fragmentation. There are also many enhancements that `Ext4` takes advantage of. Among them, we will bring the following into the discussion: a maximum filesystem size of 1 **exabyte** (**EB**), the ability to use multi-block allocation, splitting large files into the largest possible sizes for better performance, application of the allocate-on-flush technique for better performance, the use of the handy `fsck` command for speedy filesystem checks, the use of checksums for journaling and better reliability, and the use of improved timestamps.

- **ZFS**: This filesystem was created at Sun Microsystems and combines a file system and a logical volume manager into one solution. It was announced in 2004, with development starting in 2001, and was first integrated into the Solaris operating system, then used (not the default though) by Debian, FreeBSD, and others. ZFS is a highly scalable 128-bit system that offers simple administration, data integrity, scalability, and performance. Development of this filesystem is done through the **OpenZFS** open source project. ZFS offers a complex structure by using a copy-on-write mechanism, different from traditional filesystems. For more detailed information about ZFS, we recommend the following link: https://openzfs.github.io/openzfs-docs/Getting%20Started/index.html.

- **XFS**: Enterprise Linux is starting to change by moving away from `Ext4` to other competent filesystem types. Among those is `XFS`. This filesystem was first created by Silicon Graphics, Inc and used in the IRIX operating system. Its most important key design element is performance as it is capable of dealing with large datasets. Furthermore, it is designed to handle parallel I/O tasks with a guaranteed high I/O rate. The filesystem supports up to 16 EB with support for individual files up to 8 EB. `XFS` has a feature to journal quota information, together with online maintenance tasks such as defragmenting, enlarging, and restoring. There are also specific tools for backup and restore, including `xfsdump` and `xfsrestore`.

- **btrfs**: The B-tree filesystem (`btrfs`) is still under development, but it addresses issues associated with existing filesystems, including the lack of snapshots, pooling, checksums, and multi-device spanning. These are features that are required in an enterprise Linux environment. The ability to take snapshots of the filesystem and maintain its internal framework for managing new partitions makes `btrfs` a viable newcomer in terms of the critical enterprise ecosystem.

There are other filesystems that we did not discuss here, including **ReiserFS** and **GlusterFS**, **Network File System** (**NFS**), **Samba CIFS File System** (**SMB**), **ISO9660** for CD-ROMs and Joliet

extensions, and non-native Linux ones, including **FAT**, **NTFS**, **exFAT**, and **APFS**, or **MacOS Extended**, among others. If you want to learn about these in more detail, feel free to investigate further; a good starting point is Wikipedia: https://en.wikipedia.org/wiki/File_system. To check the list of supported filesystems on your Linux distribution, run the `cat /proc/filesystems` command.

Linux implements a special software system that is designed to run specific functions of the filesystems. It is known as the **virtual file system** and acts as a bridge between the kernel and the filesystem types and hardware. Therefore, when an application wants to open a file, the action is delivered through the Virtual File System as an abstraction layer:

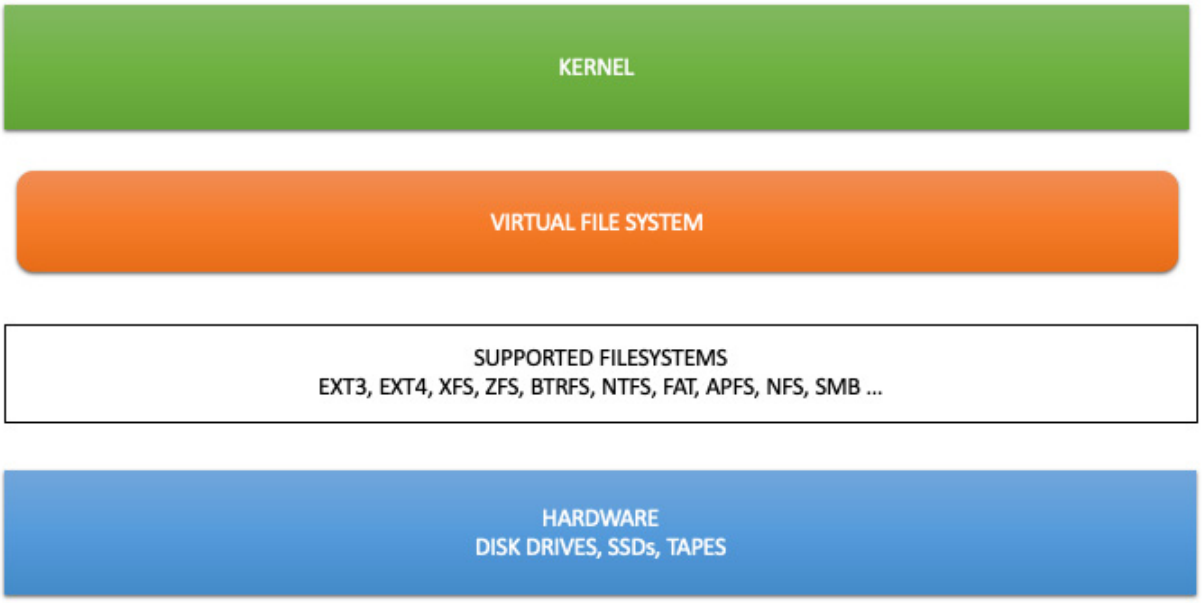

Figure 6.5 – The Linux Virtual File System abstraction layer

Basic filesystem functions include provisioning namespaces, metadata structures as a logical foundation for hierarchical directory structures, disk block usage, file size and access information, and high-level data for logical volumes and partitions. There is also an **application programming interface** (**API**) available for every filesystem. Thus, developers can access system function calls for filesystem object manipulation with specific algorithms for creating, moving, and deleting files, or for indexing, searching, and finding files. Furthermore, every modern filesystem provides a special access rights scheme that's used to determine the rules governing a user's access to files.

At this point, we have already covered the principal Linux filesystems, including `EXT4`, `btrfs`, and `XFS`. In the next section, we will teach you the basics of disks and partition management in Linux.

Understanding disks and partitions

Understanding disks and partitions is a key asset for any system administrator. Formatting and partitioning disks is critical, starting with system installation. Knowing the type of hardware available on your system is important, and it is therefore imperative to know how to work with it. One of these is the disk; let's look at this in further detail.

Common disk types

A **disk** is a hardware component that stores your data. It comes in various types and uses different interfaces. The main disk types are the well-known **spinning HDD**, the SSD, and the **non-volatile memory express** (**NVMe**). SSDs and NVMes use RAM-like technologies, with better energy consumption and higher transfer rates than original spinning hard drives. The following interfaces are used:

- **Integrated Drive Electronics** (**IDE**): This is an old standard that's used on consumer hardware with small transfer rates. It's now deprecated.
- **Serial Advanced Technology Attachment** (**SATA**): This replaced IDEs and has transfer rates of up to 16 GB/s.
- **Small Computer Systems Interface** (**SCSI**): This is used mostly in enterprise servers with RAID configurations with sophisticated hardware components.
- **Serial Attached SCSI** (**SAS**): This is a point-to-point serial protocol interface with transfer rates similar to SATA. It is mostly used in enterprise environments for their reliability.
- **Universal Serial Bus** (**USB**): This is used for external hard drives and memory drives.

Each disk has a specific geometry that consists of heads, cylinders, tracks, and sectors. On a Linux system, to see the information regarding a disk's geometry, you can use the `fdisk -l` command.

On our primary workstation, we have a single SSD running Debian 12 GNU/Linux and a USB device inserted in one of the ports. We will run the following command to obtain information about the drives on our machine:

```
sudo fdisk -l
```

The following screenshot shows excerpts of the `fdisk` command's output for both drives:

```
alexandru@debian:~$ sudo fdisk -l
Disk /dev/nvme0n1: 953.87 GiB, 1024209543168 bytes, 2000409264 sectors
Disk model: Lexar SSD NM620 1TB
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: A6049E7C-127A-40D3-9D3F-F5CF9D463B88

Device           Start        End    Sectors   Size Type
/dev/nvme0n1p1    2048    1050623    1048576   512M EFI System
/dev/nvme0n1p2 1050624    2050047     999424   488M Linux filesystem
/dev/nvme0n1p3 2050048 2000408575 1998358528 952.9G Linux filesystem

Disk /dev/sda: 7.5 GiB, 8053063680 bytes, 15728640 sectors
Disk model: Flash Disk
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x24248bcf

Device     Boot Start      End  Sectors  Size Id Type
/dev/sda1        2048 15728639 15726592  7.5G 83 Linux
```

Figure 6.6 – The output of the fdisk -l command showing disk information

The output of the **`fdisk`** utility may look intimidating at first, but rest assured that we will explain it to you so that it will look friendlier from now on. By using the **`fdisk`** utility without a specific partition as an argument, all the partition information available inside **`/proc/partitions`** will be shown. In the example shown in the preceding screenshot, you have details on two disks that are available on our system: a 1 TB Lexar NM620 SSD and an 8 GB USB flash drive attached. Let's explain how the 1 TB drive is shown:

- First, you have **`Disk model`** with the name of the drive, **`Units`** as sectors, each of which has a size of 512 bytes, **`Disklabel type`** as GPT, and **`Disk identifier`**, which is unique for each drive.
- Next is a table of the partitions available on the disk. This table has six columns (sometimes seven columns, as in the case of the USB flash drive, shown on the lower side of the screenshot). The first column has the **`Device`** header and shows the partition naming scheme. The second and third columns (in our example) show the starting and ending sectors. The fourth column shows the total number of sectors on the partition. The fifth column shows the size of the partition in human-readable format and the last column shows the type of the filesystem.

Knowing basic information about disk devices on your system is merely the starting point for working with disks and partitions on Linux. Disks are just a big chunk of metal if we don't format and partition them so that the system can use them. This is why, in the next section, we will teach you what partitions are.

Partitioning disks

Commonly, disks use **partitions**. To understand partitions, knowing a disk's geometry is essential. This legacy knowledge base is useful even when dealing with SSDs. Partitions are contiguous sets of sectors and/or cylinders, and they can be of several types: **primary**, **extended**, and **logical** partitions. A maximum number of 15 partitions can exist on a disk. The first four will be either primary or extended, and the remaining are logical partitions. Furthermore, there can only be a single extended partition, but they can be divided into several logical partitions until the maximum number is reached.

Partition types

There are two major partition types – the **Master Boot Record** (**MBR**) and the **GUID Partition Table** (**GPT**). MBR was intensively used up to around 2010. Its limitations include the maximum number of primary partitions (four) and the maximum size of a partition (2 TB). MBR uses hexadecimal codes for different types of partitions, such as `0x0c` for FAT, `0x07` for NTFS, `0x83` for a Linux filesystem type, and `0x82` for swap. GPT became a part of the **Unified Extensible Firmware Interface** (**UEFI**) standard as a solution to some issues with MBR, including partition limitations, addressing methods, using only one copy of the partition table, and so on. It supports up to 128 partitions and disk sizes of up to 75.6 **Zettabytes** (**ZB**).

The partition table

The **partition table** of a disk is stored inside the disk's MBR. MBR is the first 512 bytes of a drive. Out of these, the partition table is 64 bytes and is stored after the first 446 bytes of records. At the end of MBR, there are 2 bytes known as the end of sector marker. The first 446 bytes are reserved for code that usually belongs to a bootloader program. In the case of Linux, the bootloader is called **GRand Unified Bootloader** (**GRUB**).

When you boot up a Linux system, the bootloader looks for the active partition. There can only be one active partition on a single disk. When the active partition is located, the bootloader loads items. The partition table has 4 entries, each of which is 16 bytes in size, with each belonging to a possible primary partition on the system. Furthermore, each entry contains information regarding the beginning address of `cylinder/head/sectors`, the partition type code, the end address of `cylinder/head/sectors`, the starting sector, and the number of sectors inside one partition.

Naming partitions

The kernel interacts with the disk at a low level. This is done through device nodes that are stored inside the **`/dev`** directory. Device nodes use a simple naming convention that tells you which disk is the one that requires your attention. Looking at the contents of the **`/dev`** directory, you can see all the available disk nodes, also referred to as disk drives, in *Figure 6.2* and *Figure 6.3* earlier in this section. A short explanation is always useful, so disks and partitions are recognized as follows:

- The first hard drive is always **`/dev/sda`** (for an SCSI or SATA device)
- The second hard drive is **`/dev/sdb`**, the third is **`/dev/sdc`**, and so on
- The first partition of the first disk is **`/dev/sda1`**
- The first partition of the second disk is **`/dev/sdb1`**
- The second partition of the second disk is **`/dev/sdb2`**, and so on

We specified that this is true in the case of an SCSI and SATA, and we need to explain this in a little more detail. The kernel gives the letter designation, such as *a*, *b*, and *c*, based on the ID number of the SCSI device, and not based on the position of the hardware bus.

Partition attributes

To learn about your partition's attributes, you can use the **`lsblk`** command. We will run it on our Debian system, as shown in the following screenshot:

```
alexandru@debian:~$ lsblk
NAME        MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINTS
sda           8:0    1   7.5G  0 disk
└─sda1        8:1    1   7.5G  0 part  /media/alexandru/9d7587da-3e17-464c-b01d-
d044b82f3736
nvme0n1     259:0    0 953.9G  0 disk
├─nvme0n1p1 259:1    0   512M  0 part  /boot/efi
├─nvme0n1p2 259:2    0   488M  0 part  /boot
└─nvme0n1p3 259:3    0 952.9G  0 part
  └─nvme0n1p3_crypt
            254:0    0 952.9G  0 crypt
    ├─debian--vg-root
    │       254:1    0 951.9G  0 lvm   /
    └─debian--vg-swap_1
            254:2    0   976M  0 lvm   [SWAP]
```

Figure 6.7 – The output of lsblk

The **`lsblk`** command shows the device's name (the node's name from **`sysfs`** and the **`udev`** database), the major and minor device number, the removable state of the device (`0` for a non-removable device and `1` for a removable device), the size in human-readable format, the read-only state (again, using `0`

for the ones that are not read-only and `1` for the read-only ones), the type of device, and the device's mount point (where available).

Now that we know more about the drive, let's learn how to alter a disk's partition table.

Partition table editors

In Linux, there are several tools we can use when managing partition tables. Among the most commonly used ones are the following:

- **`fdisk`**: A command-line partition editor, perhaps the most widely used one
- **`Sfdisk`**: A non-interactive partition editor, used mostly in scripting
- **`parted`**: The GNU (the recursive acronym for GNU is *GNU's Not Unix*) partition manipulation software
- **`gparted`**: The graphical interface for **`parted`**

Of these, we will only detail how to use **`fdisk`** as this is the most widely used command-line partition editor in Linux. It is found in both Ubuntu/Debian and RHEL/Fedora or openSUSE and many other distributions too.

But before we use **`fdisk`**, we would like to see the partitions that the operating system knows about. If you are not sure about the operations you just completed, you can always visualize the contents of the **`/proc/partitions`** file with the **`cat`** command:

```
alexandru@debian:~$ cat /proc/partitions
major minor  #blocks  name

 259        0 1000204632 nvme0n1
 259        1     524288 nvme0n1p1
 259        2     499712 nvme0n1p2
 259        3  999179264 nvme0n1p3
 254        0  999162880 dm-0
 254        1  998162432 dm-1
 254        2     999424 dm-2
   8        0    7864320 sda
   8        1    7863296 sda1
```

Figure 6.8 – Listing the /proc/partitions file

To use **`fdisk`**, you must be the root user. We advise you to use caution when using **`fdisk`** as it can damage your existing partitions and disks. **`fdisk`** can be used on a particular disk as follows:

Figure 6.9 – Using fdisk for the first time

You will notice that when using `fdisk` for the first time, you are warned that changes will be done to the disk only when you decide to write them to it. You will also be prompted to introduce a command, and you will be shown the `m` option for help. We advise you to always use the help menu, even if you already know the most used commands.

When you type `m`, you will be shown the entire list of commands available for `fdisk`. You will see options to manage partitions, create new boot records, save changes, and others. Partition table editors are important tools for managing disks in Linux. Their use is incomplete if you do not know how to format a partition. In the next section, we will show you how to partition a disk drive.

Creating and formatting partitions

We will use the `fdisk` utility to create a new partition table on a USB memory stick plugged into our primary workstation running Debian GNU/Linux. We will create an MBR partition table using the following command:

```
sudo fdisk /dev/sda
```

We will use the `o` option to create an empty MBR partition table and then the `w` option to save the changes to disk. The output of the command is shown in the following screenshot:

```
alexandru@debian:~$ sudo fdisk /dev/sda

Welcome to fdisk (util-linux 2.38.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Command (m for help): o
Created a new DOS (MBR) disklabel with disk identifier 0xc1fa6ca8.

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
```

Figure 6.10 – Creating a new MBR partition table with fdisk

With that, the partition table has been created, but there is no partition defined on the disk. While still inside the `fdisk` command-line interface, you can use the `v` option to verify the newly created partition table and the `I` option to see information about existing partitions. You will see some output saying that no partitions have been defined yet. So, it is time to set up a new partition.

To create a new partition, we will use the following series of options:

- The `n` option to start the creation processes
- The `p` option when asked to create either a primary (`p`) or extended (`e`) partition type
- Enter the partition number (use the default of `1`)
- Enter the first sector (use the default of `2048`)
- Enter the last sector – if you want a specific size for the partition, you can use size values in KB, MB, GB, and so on, or sector values (the default is the maximum size of the disk)
- If asked to remove any signatures, type `Y` to remove them
- `w` to write changes to disk

The output of the previous series of actions is shown in the following screenshot:

```
Command (m for help): n
Partition type
   p   primary (0 primary, 0 extended, 4 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-15728639, default 2048):
Last sector, +/-sectors or +/-size{K,M,G,T,P} (2048-15728639, default 15728639):


Created a new partition 1 of type 'Linux' and of size 7.5 GiB.
Partition #1 contains a ext4 signature.

Do you want to remove the signature? [Y]es/[N]o: Y

The signature will be removed by a write command.

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
```

Figure 6.11 – Creating a new partition with fdisk

With that, the partition has been created, but it hasn't been formatted. Before we learn how to format partitions, let's learn how to back up a partition table.

There are situations when you will need to back up and restore your **partition tables**. As partitioning could go sideways sometimes, a good backup strategy could help you. To do this, you can use the `dd` utility. The command to use is as follows:

```
sudo dd if=/dev/sda of=mbr-backup bs=512 count=1
```

This program is very useful and powerful as it can clone disks or wipe data. Here is an example showing the output of the command:

```
alexandru@debian:~$ pwd
/home/alexandru
alexandru@debian:~$ sudo dd if=/dev/sda
sda   sda1
alexandru@debian:~$ sudo dd if=/dev/sda of=mbr-backup bs=512 count=1
1+0 records in
1+0 records out
512 bytes copied, 0.00138593 s, 369 kB/s
alexandru@debian:~$ ls
Desktop    Downloads   Music     Public     Videos
Documents  mbr-backup  Pictures  Templates
alexandru@debian:~$
```

Figure 6.12 – Backing up MBR with the dd command

The `dd` command has a clear syntax. By default, it uses the standard input and standard output, but you can change those by specifying new input files with the `if` option, and output files with the `of` option. We specified the input file as the device file for the disk we wanted to back up and gave a name for the backup output file. We also specified the block size using the `bs` option, and the `count` option to specify the number of blocks to read.

To restore the bootloader, we can use the `dd` command, as follows:

```
sudo dd if=~/mbr-backup of=/dev/sda bs=512 count=1
```

Now that you have learned how to use `dd` to back up a partition table, let's format the partition we created earlier. The most commonly used program for formatting a filesystem on a partition is `mkfs`. Formatting a partition is also known as *making* a filesystem, hence the name of the utility. It has specific tools for different filesystems, all using the same frontend utility. The following is a list of all filesystems supported by `mkfs`:

```
alexandru@debian:~$ ls -lh /sbin/mkfs*
-rwxr-xr-x 1 root root  15K Feb 13 10:48 /sbin/mkfs
-rwxr-xr-x 1 root root  35K Feb 13 10:48 /sbin/mkfs.bfs
-rwxr-xr-x 1 root root  43K Feb 13 10:48 /sbin/mkfs.cramfs
-rwxr-xr-x 1 root root  51K Oct 28 15:48 /sbin/mkfs.exfat
lrwxrwxrwx 1 root root    6 Feb  2 07:38 /sbin/mkfs.ext2 -> mke2fs
lrwxrwxrwx 1 root root    6 Feb  2 07:38 /sbin/mkfs.ext3 -> mke2fs
lrwxrwxrwx 1 root root    6 Feb  2 07:38 /sbin/mkfs.ext4 -> mke2fs
-rwxr-xr-x 1 root root  63K Feb  8  2021 /sbin/mkfs.fat
-rwxr-xr-x 1 root root 111K Feb 13 10:48 /sbin/mkfs.minix
lrwxrwxrwx 1 root root    8 Feb  8  2021 /sbin/mkfs.msdos -> mkfs.fat
lrwxrwxrwx 1 root root    6 Oct 31 16:14 /sbin/mkfs.ntfs -> mkntfs
lrwxrwxrwx 1 root root    8 Feb  8  2021 /sbin/mkfs.vfat -> mkfs.fat
```

Figure 6.13 – Details regarding the mkfs utility

To format the target disk as having the **`Ext4`** filesystem, we will use the **`mkfs`** utility. The commands to execute are shown here:

1. First, we will run the **`fdisk`** utility to make sure that we select the largest disk correctly. Run the following command:

   ```
   sudo fdisk -l
   ```

2. Then, check the output with extreme caution and select the correct disk name.
3. Now that we know which disk to work with, we will use **`mkfs`** to format it as an **`Ext4`** filesystem. The output is shown here:

```
alexandru@debian:~$ sudo mkfs.ext4 /dev/sda
mke2fs 1.46.6 (1-Feb-2023)
Found a dos partition table in /dev/sda
Proceed anyway? (y,N) y
Creating filesystem with 1966080 4k blocks and 491520 inodes
Filesystem UUID: 9dfc7fe4-e4d4-4223-92d6-94dd7e8e6e69
Superblock backups stored on blocks:
  32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done
```

Figure 6.14 – Formatting an Ext4 partition using mkfs

When using **`mkfs`**, there are several options available. To create an **`Ext4`** type partition, you can either use the command shown in *Figure 6.14* or you can use the **`-t`** option followed by the filesystem type. You can also use the **`-v`** option for a more verbose output, and the **`-c`** option for bad sector scanning

while creating the filesystem. You can also use the `-L` option if you want to add a label for the partition right from the command. The following is an example of creating an `Ext4` filesystem partition with the name `newpartition`:

```
sudo mkfs -t ext4 -v -c -L newpartition /dev/sda
```

Once a partition is formatted, it's advised that you check it for errors. Similar to `mkfs`, there is a tool called `fsck`. This is a utility that sometimes runs automatically following an abnormal shutdown or on set intervals. It has specific programs for the most commonly used filesystems, just like `mkfs`. The following is the output of running `fsck` on one of our partitions. After running, it will show whether there are any problems. In the following screenshot, the output shows that checking the partition resulted in no errors:

```
alexandru@debian:~$ sudo fsck -t ext4 /dev/sda
fsck from util-linux 2.38.1
e2fsck 1.46.6 (1-Feb-2023)
/dev/sda: clean, 11/491520 files, 55879/1966080 blocks
```

Figure 6.15 – Using fsck to check a partition

After partitions are created, they need to be mounted; otherwise, they cannot be used.

IMPORTANT NOTE

***Mounting** is an important action in Linux, and any other operating system for that matter. By mounting, you give the operating system access to the disk resource in such a way that it looks like it is using a local disk. On Linux, the external disk that is mounted is linked to a mount point, which is a directory on the local filesystem. Mount points are essential to POSIX-compatible operating systems, such as Linux. Mounting a disk makes it accessible to the entire operating system through mount points. For more information on mounting, visit https://docs.oracle.com/cd/E19455-01/805-7228/6j6q7ueup/index.html.*

Each partition will be mounted inside the existing filesystem structure. Mounting is allowed at any point in the tree structure. Each filesystem is mounted under certain directories, created inside the directory structure. We will explore mounting and unmounting partitions in the next section.

Mounting and unmounting partitions

The **mounting** utility in Linux is simply called `mount`, and the **unmounting** utility is called `umount`. To see whether a certain partition is mounted, you can simply type `mount` and see the output, which will be of a significant size. You can use `grep` to filter it:

```
mount | grep /dev/sda
```

We are looking for `/dev/sda` in the output, but it is not shown. This means that the drive is not mounted.

To mount it, we need to make a new directory. For simplicity, we will show all the steps required until you mount and use the partition:

1. Create a new directory to mount the partition. In our case, we created a new directory called **USB** inside the **/home/alexandru** directory:

   ```
   mkdir USB
   ```

2. Mount the partition using the following command:

   ```
   sudo mount /dev/sda /home/alexandru/USB
   ```

3. Start using the new partition from the new location. As an example, we will copy the **mbr-backup** file we created a few steps back to the newly mounted USB memory stick using the following command:

   ```
   sudo cp mbr-backup USB/
   ```

The following is the output for all the commands from the preceding list:

```
alexandru@debian:~$ pwd
/home/alexandru
alexandru@debian:~$ mkdir USB
alexandru@debian:~$ sudo mount /dev/sda /home/alexandru/USB
alexandru@debian:~$
alexandru@debian:~$ sudo cp mbr-backup USB/
alexandru@debian:~$ ls -l USB/
total 4
-rw-r--r-- 1 root root 512 Mar 16 15:44 mbr-backup
alexandru@debian:~$
```

Figure 6.16 – Mounting an external memory stick

The **mount** command needs to be used with superuser permission. If you try to mount an external USB device without **sudo**, you will get the following message:

```
alexandru@debian:~$ mount /dev/sda /home/alexandru/USB
mount: /home/alexandru/USB: must be superuser to use mount.
       dmesg(1) may have more information after failed mount system call.
```

Figure 6.17 – Error for not using sudo with mount

The **mount** utility has many options available. Use the help menu to see everything that it has under the hood. Now that the partition has been mounted, you can start using it. If you want to unmount it, you can use the **umount** utility. You can use it as follows:

```
sudo umount /dev/sda
```

When unmounting a filesystem, you may receive errors if that partition is still in use. Being in use means that certain programs from that filesystem are still running in memory, using files from that partition. Therefore, you first have to close all running applications, and if other processes are using that filesystem, you will have to kill them, too. Sometimes, the reason a filesystem is busy is not clear at first, and to know which files are open and running, you can use the `lsof` command:

```
sudo lsof | grep /dev/sda
```

Mounting a filesystem only makes it available until the system is shut down or rebooted. If you want the changes to be persistent, you will have to edit the `/etc/fstab` file accordingly. First, open the file with your favorite text editor:

```
sudo nano /etc/fstab
```

Add a new line similar to the one that follows:

```
/dev/sda /mnt/sdb ext4 defaults 0 0
```

The `/etc/fstab` file is a configuration file for the filesystem table. It consists of a set of rules needed to control how the filesystems are used. This simplifies the need to manually mount and unmount each disk when used, by drastically reducing possible errors. The table has a six-column structure, with each column designated with a specific parameter. There is only one correct order for the parameters to work:

- **Device name**: Either by using UUID or the mounted device name
- **Mount point**: The directory where the device is, or will be, mounted
- **Filesystem type**: The filesystem type used
- **Options**: The options shown, with multiple ones separated by commas
- **Backup operation**: This is the first digit from the last two digits in the file; `0` = no backup, `1` = dump utility backup
- **Filesystem check order**: This is the last digit inside the file; `0` = no `fsck` filesystem check, with `1` for the root filesystem, and `2` for other partitions

By updating the `/etc/fstab` file, the mounting is permanent and is not affected by any shutdown or system reboot. Usually, the `/etc/fstab` file only stores information about the internal hard drive partitions and filesystems. The external hard drives or USB drives are automatically mounted under `/media` by the kernel's **hardware abstraction layer** (**HAL**).

By now, you should be comfortable with managing partitions in Linux, but there is still one type of partition we have not discussed: the **swap partition**. In the next section, we will introduce you to how swap works on Linux.

Swap partition

Linux uses a robust swap implementation. The virtual memory uses hard drive space when physical memory is full through swap. This additional space is made available either for the programs that do not use all the memory they are given, or when memory pressure is high. Swapping is usually done using one or more dedicated partitions as Linux permits multiple swap areas. The recommended swap size is at least the total RAM on the system. To check the actual swap used on the system, you can concatenate the `/proc/swaps` file or use the `free` command to see swap utilization, as shown in the following screenshot:

```
alexandru@debian:~$ cat /proc/swaps
Filename                                Type            Size            Used            Priority
/dev/dm-2                               partition       999420          300             -2
alexandru@debian:~$ free
               total        used        free      shared  buff/cache   available
Mem:        48124468     7763840    15675640       69204    25344948    40360628
Swap:         999420         300      999120
```

Figure 6.18 – Checking the currently used swap

If swap is not set up on your system, you can format a partition as swap and activate it. The commands to do that are as follows:

```
mkswap /dev/sda1
swapon /dev/sda1
```

The operating system is caching file contents inside the memory to prevent the use of swap as much as possible. This happens because memory is working at much higher speeds compared to hard drives or hard disk drives. Only when available memory is limited will swap be used. However, the memory that the kernel uses is never swapped; only the memory that the user space is using gets to be swapped. This assures data integrity for the kernel. Refer to the utilities we applied in *Chapter 5* to show memory usage in Linux.

Filesystems and partitions are the bare bones of any disk management task, but there are still several hiccups that an administrator needs to overcome, and this can be solved by using logical volumes. This is why, in the next section, we will introduce you to LVM.

Introducing LVM in Linux

Some of you may have already heard of **LVM**. For those who do not know what it is, we will explain it briefly in this section. Imagine a situation where your disks run out of space. You can always move it to a larger disk and then replace the smaller one, but this implies system restarts and unwanted downtimes. As a solution, you can consider LVM, which offers more flexibility and efficiency. By using LVM, you can add more physical disks to your existing volume groups while they're still in

use. This still offers the possibility to move data to a new hard drive but with no downtime – everything is done while filesystems are online.

The utilities used in Linux for LVM management are called **pvcreate**, **vgcreate**, **vgdisplay**, **lvcreate**, **lvextend**, and **lvdisplay**. Let's learn how to use them.

As we don't have a system with LVM set up just yet, we will show you the steps that are necessary to create new LVM volumes by using another system with two internal drives: one with the operating system installed on it, and a second, internal one that's available. We'll be using Debian GNU/Linux, but the commands are the same for any other Linux-based operating system.

Follow these steps to create an LVM volume:

1. Use the **fdisk** command to verify the names of the available disks (you can also use **lsblk** for this step):

   ```
   sudo fdisk -l
   ```

   In our case, the second drive is **/dev/sda**.

2. Create the LVM physical volume with the **pvcreate** command:

   ```
   sudo pvcreate /dev/sda
   ```

   Using this command might not work from the beginning and give an error message regarding the status of the drive where you want to create a physical volume. It happened to us with one of the drives; here, we had to wipe the filesystem information from the drive using the **wipefs** utility. The output is shown in the following screenshot:

```
alexandru@debian:~$ sudo pvcreate /dev/sda
  Cannot use /dev/sda: device is partitioned
alexandru@debian:~$ wipefs --all /dev/sda
bash: wipefs: command not found
alexandru@debian:~$ sudo wipefs --all /dev/sda
/dev/sda: 8 bytes were erased at offset 0x00000200 (gpt): 45 46 49 20 50 41 52 5
4
/dev/sda: 8 bytes were erased at offset 0x1bf2975e00 (gpt): 45 46 49 20 50 41 52
 54
/dev/sda: 2 bytes were erased at offset 0x000001fe (PMBR): 55 aa
/dev/sda: calling ioctl to re-read partition table: Success
alexandru@debian:~$ sudo pvcreate /dev/sda
  Physical volume "/dev/sda" successfully created.
```

Figure 6.19 – Using pvcreate to create an LVM physical volume

3. Create a new volume group to add the new physical volume to using the **vgcreate** command:

   ```
   sudo vgcreate newvolume /dev/sda
   ```

4. You can see the new volume group by running the **vgdisplay** command:

```
alexandru@debian:~$ sudo vgcreate newvolume /dev/sda
  Volume group "newvolume" successfully created
alexandru@debian:~$ sudo vgdisplay newvolume
\\  --- Volume group ---
  VG Name               newvolume
  System ID
  Format                lvm2
```

Figure 6.20 – Creating and viewing details of the new volume

5. Now, let's create a logical volume using some of the space available from the volume group, using **`lvcreate`**. Use the **`-n`** option to add a name for the logical volume and **`-L`** to set the size in a human-readable manner (we created a 5 GB logical volume named **`projects`**):

```
alexandru@debian:~$ sudo lvcreate -n projects -L 5G newvolume
WARNING: ext4 signature detected on /dev/newvolume/projects at offset 1080. Wipe
 it? [y/n]: y
  Wiping ext4 signature on /dev/newvolume/projects.
  Logical volume "projects" created.
```

Figure 6.21 – Creating a logical volume using lvcreate

6. Check whether the logical volume exists:

```
sudo ls /dev/mapper/newvolume-projects
```

7. The newly created device can only be used if it's formatted using a known filesystem and mounted afterward, in the same way as a regular partition. First, let's format the new volume:

```
alexandru@debian:~$ sudo ls /dev/mapper/newvolume-projects
/dev/mapper/newvolume-projects
alexandru@debian:~$ sudo mkfs -t ext4 /dev/mapper/newvolume-projects
mke2fs 1.46.6 (1-Feb-2023)
Discarding device blocks: done
Creating filesystem with 1310720 4k blocks and 327680 inodes
Filesystem UUID: 1af332be-46a6-41f9-b343-271078ee6503
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done
```

Figure 6.22 – Formatting the new logical volume as an Ext4 filesystem

8. Now, it's time to mount the logical volume. First, create a new directory and mount the logical volume there. Then, check its size using the **`df`** command:

```
alexandru@debian:~$ pwd
/home/alexandru
alexandru@debian:~$ mkdir LVM
alexandru@debian:~$ sudo mount /dev/mapper/newvolume-projects /home/alexandru/LVM
alexandru@debian:~$ df -h /home/alexandru/LVM/
Filesystem                      Size  Used Avail Use% Mounted on
/dev/mapper/newvolume-projects  4.9G   24K  4.6G   1% /home/alexandru/LVM
```

Figure 6.23 – Mounting the logical volume

9. All changes implemented hitherto are not permanent. To make them permanent, you will have to edit the **`/etc/fstab`** file by adding the following within the file:

   ```
   /dev/mapper/newvolume-projects /home/alexandru/LVM ext4 defaults 1 2
   ```

10. You can now check the space available on your logical volume and grow it if you want. Use the **`vgdisplay`** command to see the following details:

    ```
    sudo vgdisplay newvolume
    ```

11. You can now expand the logical volume by using the **`lvextend`** command. We will extend the initial size by 5 GB, for a total of 10 GB. The following is an example:

```
alexandru@debian:~$ sudo lvextend -L +5G /dev/mapper/newvolume-projects
  Size of logical volume newvolume/projects changed from 5.00 GiB (1280 extents) t
o 10.00 GiB (2560 extents).
  Logical volume newvolume/projects successfully resized.
```

Figure 6.24 – Extending the logical volume using lvextend

12. Now, resize the filesystem so that it fits the new size of the logical volume using **`resize2fs`** and check for the size with **`df`**:

```
alexandru@debian:~$ sudo resize2fs -p /dev/mapper/newvolume-projects
resize2fs 1.46.6 (1-Feb-2023)
Filesystem at /dev/mapper/newvolume-projects is mounted on /home/alexandru/LVM; on
-line resizing required
old_desc_blocks = 1, new_desc_blocks = 2
The filesystem on /dev/mapper/newvolume-projects is now 2621440 (4k) blocks long.
```

Figure 6.25 – Resizing the logical volume with resize2fs and checking for the size with df

LVM is an advanced topic that will prove essential for any Linux system administrator to have. The brief hands-on examples we provided in this section only show the basic operations that you need to work with LVM. Feel free to dig deeper into this topic if you need to.

In the following section, we will discuss several more advanced LVM topics, including how to take full filesystem snapshots.

LVM snapshots

What is an LVM snapshot? It is a frozen instance of an LVM logical volume. More specifically, it uses a **copy-on-write** technology. This technology monitors each block of the existing volume, and when blocks change, due to new writings, that block's value is copied to the snapshot volume.

The snapshots are created constantly and instantly and persist until they are deleted. This way, you can create backups from any snapshot. As snapshots are constantly changing due to the copy-on-write technology, initial thoughts on the size of the snapshot should be given when creating one. Take into consideration, if possible, how much data is going to change during the existence of the snapshot. Once the snapshot is full, it will be automatically disabled.

Creating a new snapshot

To create a new snapshot, you can use the `lvcreate` command, with the `-s` option. You can also specify the size with the `-L` option and add a name for the snapshot with the `-n` option, as follows:

```
alexandru@debian:~$ sudo lvcreate -s -L 5G -n linux-snapshot-01 /dev/mapper/newvol
ume-projects
[sudo] password for alexandru:
  Logical volume "linux-snapshot-01" created.
```

Figure 6.26 – Creating an LVM snapshot with the lvcreate command

In the preceding command, we set a size of 5 GB and used the name `linux-snapshot-01`. The last part of the command contains the destination of the volume for which we created the snapshot. To list the new snapshot, use the `lvs` command:

```
alexandru@debian:~$ sudo lvs
  LV               VG        Attr       LSize  Pool Origin   Data%  Meta%  Move Log Cpy%Sync Convert
  linux-snapshot-01 newvolume swi-a-s---  5.00g      projects 0.01
  projects         newvolume owi-aos--- 10.00g
```

Figure 6.27 – Listing the available volume and the newly created snapshot

For more information on the logical volumes, run the `lvdisplay` command. The output will show information about all the volumes, and among them, you will see the snapshot we just created.

When we created the snapshot, we gave it a size of 5 GB. Now, we would like to extend it to the size of the source, which was 10 GB. We will do this with the `lvextend` command:

```
alexandru@debian:~$ sudo lvextend -L +5G /dev/mapper/newvolume-linux--snapshot--01
  Size of logical volume newvolume/linux-snapshot-01 changed from 5.00 GiB (1280 extents)
 to 10.00 GiB (2560 extents).
  Logical volume newvolume/linux-snapshot-01 successfully resized.
```

Figure 6.28 – Extending the snapshot from 5 to 10 GB

As shown in the preceding screenshot, the name that the snapshot volume is using is different from the one we used. Even though we used the name `linux-snapshot-01` for the snapshot volume, if we

do a listing of the `/dev/mapper/` directory, we will see that the name uses two more dashes instead. This is a convention that's used to represent logical volume files.

Now that you know how to create snapshots, let's learn how to restore a snapshot.

Restoring a snapshot

To restore a snapshot, first, you would need to unmount the filesystem. To unmount, we will use the `umount` command:

```
sudo umount /home/alexandru/LVM
```

Then, we can proceed to restore the snapshot with the `lvconvert` command. After the snapshot is merged into the source, we can check this by using the `lvs` command. The output of the two commands is shown in the following screenshot:

```
alexandru@debian:~$ sudo umount /home/alexandru/LVM
[sudo] password for alexandru:
alexandru@debian:~$ sudo lvconvert --merge /dev/mapper/newvolume-linux--snapshot--01
  Merging of volume newvolume/linux-snapshot-01 started.
  newvolume/projects: Merged: 100.00%
alexandru@debian:~$
alexandru@debian:~$ sudo lvs
  LV       VG        Attr       LSize  Pool Origin Data%  Meta%  Move Log Cpy%Sync Convert
  projects newvolume -wi-a----- 10.00g
```

Figure 6.29 – Restoring and checking the snapshot

Following the merge, the snapshot is automatically removed.

We have now covered all the basics of LVM in Linux. LVM is more complicated than normal disk partitioning. It might be intimidating to many, but it can show its strengths when needed. Nevertheless, it also comes with several drawbacks – for example, it can add unwanted complexity in a disaster recovery scenario or when a hardware failure occurs. But all this aside, it is still worth learning about.

Summary

Managing filesystems and disks is an important task for any Linux system administrator. Understanding how devices are managed in Linux, and how to format and partition disks, is essential. Furthermore, it is important to learn about LVM as it offers a flexible way to manage partitions.

Mastering those skills will give you a strong foundation for any basic administration task. In the following chapter, we will introduce you to the vast domain of **networking** in Linux.

Questions

If you managed to skim through some parts of this chapter, you might want to recap a few essential details about Linux filesystem and disk management:

1. Think of another tool to use for working with disks and install it.

   Hint: Try installing `parted` and use it from the command line. You can also use GParted from the GUI.

2. Experiment with using Disks (in GNOME) and KDE Partition Manager (in KDE) and use the command-line interface side by side.

   Hint: Keep both applications open and use the command-line utilities side by side. Try to format and mount a disk from the command line while keeping the GUI apps open.

3. Format new partitions using different filesystems.

   Hint: Use `btrfs` instead of `ext4`.

4. Explore your filesystem and disks.

   Hint: Use tools such as `lsblk`, `df`, and `fdisk`.

Further reading

For more information about what was covered in this chapter, please refer to the following Packt titles:

- *Linux Administration Best Practices*, by Scott Alan Miller
- *Mastering Ubuntu Server – Fourth Edition*, by Jay LaCroix

7

Networking with Linux

Linux networking is a vast domain. The last few decades have seen countless volumes and references written about Linux network administration internals. Sometimes, the mere assimilation of essential concepts can be overwhelming for both novice and advanced users. This chapter provides a relatively concise overview of Linux networking, focusing on network communication layers, sockets and ports, network services and protocols, and network security.

We hope that the content presented in this chapter is both a comfortable introduction to basic Linux networking principles for a novice user and a good refresher for an advanced Linux administrator.

In this chapter, we'll cover the following topics:

- Exploring basic networking – focusing on computer networks, networking models, protocols, network addresses, and ports. We'll also cover some practical aspects of configuring Linux network settings using the command-line Terminal.
- Working with network services – introducing common networking servers that run on Linux.
- Understanding network security.

Technical requirements

Throughout this chapter, we'll be using the Linux command line to some extent. A working Linux distribution, installed on either a **virtual machine** (**VM**) or a desktop platform, is highly recommended. If you don't have one already, go back to *Chapter 1*, *Installing Linux*, which will guide you through the installation process. Most of the commands and examples illustrated in this chapter use Ubuntu and Fedora, but the same would apply to any other Linux platform.

Exploring basic networking

Today, it's almost inconceivable to imagine a computer not connected to some sort of network or the internet. Our ever-increasing online presence, cloud computing, mobile communications, and **Internet of Things** (**IoT**) would not be possible without the highly distributed, high-speed, and scalable networks serving the underlying data traffic; yet the basic networking principles behind the driving force of the modern-day internet are decades old. Networking and communication paradigms will continue to evolve, but some of the original primitives and concepts will still have a long-lasting effect in shaping the building blocks of future communications.

This section will introduce you to a few of these networking essentials and, hopefully, spark your curiosity for further exploration. Let's start with computer networks.

Computer networks

A **computer network** is a group of two or more computers (or nodes) connected via a physical medium (cable, wireless, optical) and communicating with each other using a standard set of agreed-upon communication protocols. At a very high level, a **network communication infrastructure** includes computers, devices, switches, routers, Ethernet or optical cables, wireless environments, and all sorts of network equipment.

Beyond the *physical* connectivity and arrangement, networks are also defined by a *logical* layout via network topologies, tiers, and the related data flow. An example of a logical networking hierarchy is the three-tiered layering of the **demilitarized zone** (**DMZ**), *firewall*, and *internal* networks. The DMZ is an organization's outward-facing network, with an extra security layer against the public internet. A firewall controls the network traffic between the DMZ and the internal network.

Network devices are identified by the following aspects:

- **Network addresses**: These assist with locating nodes on the network using **communication protocols**, such as the **Internet Protocol** (**IP**) (see more on IP in the *TCP/IP protocols* section, later in this chapter)
- **Hostnames**: These are user-friendly labels associated with devices that are easier to remember than network addresses

A common classification criterion looks at the *scale* and *expansion* of computer networks. Let's introduce you to **local area networks** (**LANs**) and **wide area networks** (**WANs**):

- **LANs**: A LAN represents a group of devices connected and located in a single physical location, such as a private residence, school, or office. A LAN can be of any size, ranging from a home network with only a few devices to large-scale enterprise networks with thousands of users and computers.

  Regardless of the network's size, a LAN's essential characteristic is that it connects devices in a single, limited area. Examples of LANs include the home network of a single-family residence or your local coffee shop's free wireless service.

  For more information about LANs, you can refer to https://www.cisco.com/c/en/us/products/switches/what-is-a-lan-local-area-network.html.

  When a computer network spans multiple regions or multiple interconnected LANs, WANs come into play.

- **WANs**: A WAN is usually a network of networks, with multiple or distributed LANs communicating with each other. In this sense, we regard the internet as the world's largest WAN. An example of a WAN is the computer network of a multinational company's geographically distributed offices worldwide. Some WANs are built by service providers, to be leased to various businesses and institutions around the world.

WANs have several variations, depending on their type, range, and use. Typical examples of WANs include **personal area networks** (**PANs**), **metropolitan area networks** (**MANs**), and **cloud** or **internet area networks** (**IANs**).

For more information about WANs, you can refer to https://www.cisco.com/c/en/us/products/switches/what-is-a-wan-wide-area-network.html.

We think that an adequate introduction to basic networking principles should always include a brief presentation of the theoretical model governing network communications in general. We'll look at this next.

The OSI model

The **Open Systems Interconnection** (**OSI**) model is a theoretical representation of a multilayer communication mechanism between computer systems interacting over a network. The OSI model was introduced in 1983 by the **International Organization for Standardization** (**ISO**) to provide a standard for different computer systems to communicate with each other.

We can regard the OSI model as a universal framework for network communications. As the following figure shows, the OSI model defines a stack of seven layers, directing the communication flow:

Application Layer	7	data	User interaction and high-level APIs
Presentation Layer	6		Data translation into a usable format (e.g. encrypt/decrypt, encode/decode, compress/deflate)
Session Layer	5		Communication sessions (connections, sockets, ports)
Transport Layer	4	segments, datagrams	Reliable data segments using transmission control protocols (e.g. TCP, UDP)
Network Layer	3	packets	Data packets with addressing, routing and traffic control information (i.e. data path)
Data Link Layer	2	frames	Formatting the data as reliable data frames
Physical Layer	1	bits	Transmission/reception of raw bit streams over a physical medium

Figure 7.1 – The OSI model

In the layered view shown in the preceding figure, the communication flow moves from top to bottom (on the transmitting end) or bottom to top (on the receiving end). Before we look at each layer

in more detail, let's briefly explain how the OSI model and data encapsulation and decapsulation work.

Data encapsulation and decapsulation in the OSI model

When using the network to transfer data from one computer to another, some specific rules are followed. Inside the OSI model, the network data flows in two different ways. One way is down, from Layer 7 to Layer 1, and this is known as **data encapsulation**. The other way, up from Layer 1 to Layer 7, is known as **data decapsulation**. Data encapsulation represents the process of sending data from one computer to another, while data decapsulation represents the process of receiving data. Let's look at these in greater detail:

- **Encapsulation**: When sending data, data from one computer is converted to be sent through the network, and it receives extra information as it is being sent though all the layers of the stack. The application layer (Layer 7) is the place where the user directly interacts with the application. Then, data is sent through the presentation (Layer 6) and session (Layer 5) layers, where data is transformed into a usable format. In the transport layer (Layer 4), data is broken into smaller chunks (segments) and receives a new TCP header. Inside the network layer (Layer 3), the data is called a packet, receives an IP header, and is sent to the data link layer (Layer 2), where it is called a frame and contains both TCP and IP headers. At Layer 2, each frame receives information about the hardware addresses of the source and destination (**media access control** (**MAC**) addresses) and information about the protocols to be used in the network layer (created by the **logical link control** (**LLC**) data communication protocol). At this point, a new field is added, called **Frame Check Sequence** (**FCS**), which is used for checking errors. Then, the frames are passed through the physical layer (Layer 1).
- **Decapsulation**: When data is received, the process is identical, but in reverse order. This starts from the physical layer (Layer 1), where the first synchronization happens, after which the frame is sent through the data link layer (Layer 2), where an error check is done, by verifying the FCS field. This process is called a **Cyclic Redundancy Check** (**CRC**). The data, which is now a packet, is sent through all the other layers. Here, the headers that were added during the encapsulation process are stripped off until they reach the upper layers and become ready to be used on the target computer. A graphic explanation is provided in the following figure:

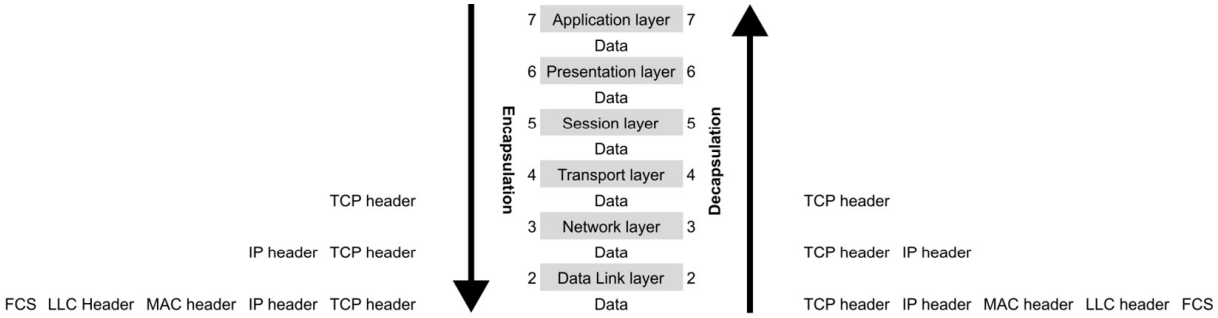

Figure 7.2 – Encapsulation and decapsulation in the OSI model

Let's look at each of these layers in detail and describe their functionality in shaping network communication.

The physical layer

The **physical layer** (or *Layer 1*) consists of the networking equipment or infrastructure connecting the devices and serving the communication, such as cables, wireless or optical environments,

connectors, and switches. This layer handles the conversion between raw bit streams and the communication medium (which includes electrical, radio, or optical signals) while regulating the corresponding bit-rate control.

Examples of protocols operating at the physical layer include Ethernet, **Universal Serial Bus** (**USB**), and **Digital Subscriber Line** (**DSL**).

The data link layer

The **data link layer** (or *Layer 2*) establishes a reliable data flow between two directly connected devices on a network, either as adjacent nodes in a WAN or as devices within a LAN. One of the data link layer's responsibilities is flow control, adapting to the physical layer's communication speed. On the receiving device, the data link layer corrects communication errors that originated in the physical layer. The data link layer consists of the following subsystems:

- **Media access control** (**MAC**): This subsystem uses MAC addresses to identify and connect devices on the network. It also controls the device access permissions to transmit and receive data on the network.
- **Logical link control** (**LLC**): This subsystem identifies and encapsulates network layer protocols and performs error checking and frame synchronization while transmitting or receiving data.

The protocol data units controlled by the data link layer are also known as **frames**. A frame is a data transmission unit that acts as a container for a single network packet. Network packets are processed at the next OSI level (*network layer*). When multiple devices access the same physical layer simultaneously, frame collisions may occur. Data link layer protocols can detect and recover from such collisions and further reduce or prevent their occurrence.

There are also Ethernet frames, for example, which are encapsulated data that is defined for MAC implementations. The original IEEE 802.3 Ethernet format, the 802.3 **SubNetwork Access Protocol** (**SNAP**), and the Ethernet II (extended) frame formats are also available.

One more example of the data link protocol is the **Point-to-Point Protocol** (**PPP**), a binary networking protocol that's used in high-speed broadband communication networks.

The network layer

The **network layer** (or *Layer 3*) discovers the optimal communication path (or route) between devices on a network. This layer uses a routing mechanism based on the IP addresses of the devices involved in the data exchange to move data packets from source to destination.

On the transmitting end, the network layer disassembles the data segments that originated in the *transport layer* into network packets. On the receiving end, the data frames are reassembled from the layer below (*data link layer*) into packets.

A protocol that operates at the network layer is the **Internet Control Message Protocol** (**ICMP**). ICMP is used by network devices to diagnose network communication issues. ICMP reports an error when a requested endpoint is not available by sending messages such as *destination network unreachable*, *timer expired*, *source route failed*, and others.

The transport layer

The **transport layer** (or *Layer 4*) operates with **data segments** or **datagrams**. This layer is mainly responsible for transferring data from a source to a destination and guaranteeing a specific **quality of service** (**QoS**). On the transmitting end, data that originated from the layer above (*session layer*) is disassembled into segments. On the receiving end, the transport layer reassembles the data packets received from the layer below (*network layer*) into segments.

The transport layer maintains the reliability of the data transfer through flow-control and error-control functions. The flow-control function adjusts the data transfer rate between endpoints with different connection speeds, to avoid a sender overwhelming the receiver. When the data received is incorrect, the error-control function may request the retransmission of data.

Examples of transport layer protocols include the **Transmission Control Protocol** (**TCP**) and the **User Datagram Protocol** (**UDP**).

The session layer

The **session layer** (or *Layer 5*) controls the lifetime of the connection channels (or sessions) between devices communicating on a network. At this layer, sessions or network connections are usually defined by network addresses, sockets, and ports. We'll explain each of these concepts in the *Sockets and ports* and *IP addresses* sections. The session layer is responsible for the integrity of the data transfer within a communication channel or session. For example, if a session is interrupted, the data transfer resumes from a previous checkpoint.

Some typical session layer protocols are the **Remote Procedure Call** (**RPC**) protocol, which is used by interprocess communications, and **Network Basic Input/Output System** (**NetBIOS**), which is a file-sharing and name-resolution protocol.

The presentation layer

The **presentation layer** (or *Layer 6*) acts as a data translation tier between the *application layer* above and the *session layer* below. On the transmitting end, this layer formats the data into a system-independent representation before sending it across the network. On the receiving end, the presentation layer transforms the data into an application-friendly format. Examples of such transformations are encryption and decryption, compression and decompression, encoding and decoding, and serialization and deserialization.

Usually, there is no substantial distinction between the presentation and application layers, mainly due to the relatively tight coupling of the various data formats with the applications consuming them. Standard data representation formats include the **American Standard Code for Information Interchange** (**ASCII**), **Extensible Markup Language** (**XML**), **JavaScript Object Notation** (**JSON**), **Joint Photographic Experts Group** (**JPEG**), ZIP, and others.

The application layer

The **application layer** (or *Layer 7*) is the closest to the end user in the OSI model. This layer collects or provides the input or output of application data in some meaningful way. This layer does not contain or run the applications themselves. Instead, it acts as an abstraction between applications, implementing a communication component and the underlying network. Typical examples of applications that interact with the application layer are web browsers and email clients.

A few examples of Layer 7 protocols are the DNS protocol, the **HyperText Transfer Protocol** (**HTTP**), the **File Transfer Protocol** (**FTP**), and email messaging protocols, such as the **Post Office Protocol** (**POP**), **Internet Message Access Protocol** (**IMAP**), and **Simple Mail Transfer Protocol** (**SMTP**).

Before wrapping up, we should note that the OSI model is a generic representation of networking communication layers and provides the theoretical guidelines for how network communication works. A similar – but more practical – illustration of the networking stack is the TCP/IP model. Both these models are useful when it comes to network design, implementation, troubleshooting, and diagnostics. The OSI model gives network operators a good understanding of the full networking stack, from the physical medium to the application layer, and each level has **Protocol Data Units** (**PDUs**) and communication internals. However, the TCP/IP model is somewhat simplified, with a few of the OSI model layers collapsed into one, and it takes a rather protocol-centric approach to network communications. We'll explore this in more detail in the next section.

The TCP/IP network stack model

The **TCP/IP model** is a four-layer interpretation of the OSI networking stack, where some of the equivalent OSI layers appear consolidated, as shown in the following figure:

	OSI Model	TCP/IP Model	TCP/IP Protocols
7	Application Layer	Application Layer	DNS, HTTP, FTP, SMTP, SNMP, Telnet, …
6	Presentation Layer		
5	Session Layer		
4	Transport Layer	Transport Layer	TCP, UDP, …
3	Network Layer	Internet Layer	IP, ARP, ICMP, IGMP, …
2	Data Link Layer	Network Interface Layer	Ethernet, Token Ring, Frame Relay, ATM, …
1	Physical Layer		

Figure 7.3 – The OSI and TCP/IP models

Chronologically, the TCP/IP model is older than the OSI model. It was first suggested by the US **Department of Defense** (**DoD**) as part of an internetwork project developed by the **Defense Advanced Research Projects Agency** (**DARPA**). This project eventually became the modern-day internet.

The TCP/IP model layers encapsulate similar functions to their counterpart OSI layers. Here's a summary of each layer in the TCP/IP model.

The network interface layer

The **network interface layer** is responsible for data delivery over a physical medium (such as wire, wireless, or optical). Networking protocols operating at this layer include Ethernet, Token Ring, and Frame Relay. This layer maps to the composition of the *physical and data link layers* in the OSI model.

The internet layer

The **internet layer** provides *connectionless* data delivery between nodes on a network. Connectionless protocols describe a network communication pattern where a sender transmits data to a receiver without a prior arrangement between the two. This layer is responsible for disassembling data into network packets at the transmitting end and reassembling them on the receiving end. The internet layer uses routing functions to identify the optimal path between the network nodes. This layer maps to the *network layer* in the OSI model.

The transport layer

The **transport layer** (also known as the **transmission layer** or the **host-to-host layer**) is responsible for maintaining the communication sessions between connected network nodes. The transport layer implements error-detection and correction mechanisms for reliable data delivery between endpoints. This layer maps to the *transport layer* in the OSI model.

The application layer

The **application layer** provides the data communication abstraction between software applications and the underlying network. This layer maps to the composition of the *session, presentation, and application layers* in the OSI model.

As discussed earlier in this chapter, the TCP/IP model is a protocol-centric representation of the networking stack. This model served as the foundation of the internet by gradually defining and developing networking protocols required for internet communications. These protocols are collectively referred to as the *IP suite*. The following section describes some of the most common networking protocols.

TCP/IP protocols

In this section, we'll describe some widely used networking protocols. The reference provided here should not be regarded as an all-encompassing guide. There are a vast number of TCP/IP protocols, and a comprehensive study is beyond the scope of this chapter. Nevertheless, there are a handful of protocols worth exploring, frequently at work in everyday network communication and administration workflows.

The following list briefly describes each TCP/IP protocol and its related **Request for Comments** (**RFC**) identifier. The RFC represents the detailed technical documentation – of a protocol, in our case – that's usually authored by the **Internet Engineering Task Force** (**IETF**). For more information about RFC, please refer to https://www.ietf.org/standards/rfcs/. Here are the most widely used protocols:

- **IP**: IP (*RFC 791*) identifies network nodes based on fixed-length addresses, also known as IP addresses. IP addresses will be described in more detail in the next section. The IP protocol uses datagrams as the data transmission unit and provides fragmentation and reassembly capabilities of large datagrams to accommodate small-packet networks (and avoid transmission delays). The IP protocol also provides routing functions to find the optimal data path between network nodes. IP operates at the network layer (*Layer 3*) in the OSI model.
- **ARP**: The **Address Resolution Protocol** (**ARP**) (*RFC 826*) is used by the IP protocol to map IP network addresses (specifically, **IP version 4** or **IPv4**) to device MAC addresses used by a data link protocol. ARP operates at the data link layer (*Layer 2*) in the OSI model.
- **NDP**: The **Neighbor Discovery Protocol** (**NDP**) (*RFC 4861*) is like the ARP protocol, and it also controls **IP version 6** (**IPv6**) address mapping. NDP operates within the data link layer (*Layer 2*) in the OSI model.

- **ICMP**: ICMP (*RFC 792*) is a supporting protocol for checking transmission issues. When a device or node is not reachable within a given timeout, ICMP reports an error. ICMP operates at the network layer (*Layer 3*) in the OSI model.
- **TCP**: TCP (*RFC 793*) is a connection-oriented, highly reliable communication protocol. TCP requires a logical connection (such as a *handshake*) between the nodes before initiating the data exchange. TCP operates at the transport layer (*Layer 4*) in the OSI model.
- **UDP**: UDP (*RFC 768*) is a connectionless communication protocol. UDP has no handshake mechanism (compared to TCP). Consequently, with UDP, there's no guarantee of data delivery. It is also known as a *best-effort protocol*. UDP uses datagrams as the data transmission unit, and it's suitable for network communications where error checking is not critical. UDP operates at the transport layer (*Layer 4*) in the OSI model.
- **Dynamic Host Configuration Protocol** (**DHCP**): The DHCP (*RFC 2131*) provides a framework for requesting and passing host configuration information required by devices on a TCP/IP network. DHCP enables the automatic (dynamic) allocation of reusable IP addresses and other configuration options. DHCP is considered an application layer (*Layer 7*) protocol in the OSI model, but the initial DHCP discovery mechanism operates at the data link layer (*Layer 2*).
- **Domain Name System** (**DNS**): The DNS (*RFC 2929*) is a protocol that acts as a network address book, where nodes in the network are identified by human-readable names instead of IP addresses. According to the IP protocol, each device on a network is identified by a unique IP address. When a network connection specifies the remote device's hostname (or domain name) before the connection is established, DNS translates the domain name (such as **`dns.google.com`**) into an IP address (such as **`8.8.8.8`**). The DNS protocol operates at the application layer (*Layer 7*) in the OSI model.
- **HTTP**: HTTP (*RFC 2616*) is the vehicular language of the internet. HTTP is a stateless application-level protocol based on the request and response between a client application (for example, a browser) and a server endpoint (for example, a web server). HTTP supports a wide variety of data formats, ranging from text to images and video streams. HTTP operates at the application layer (*Layer 7*) in the OSI model.
- **FTP**: FTP (*RFC 959*) is a standard protocol for transferring files requested by an FTP client from an FTP server. FTP operates at the application layer (*Layer 7*) in the OSI model.
- **TELNET**: The **Terminal Network protocol** (**TELNET**) (*RFC 854*) is an application-layer protocol that provides a bidirectional text-oriented network communication between a client and a server machine, using a virtual terminal connection. TELNET operates at the application layer (*Layer 7*) in the OSI model.
- **SSH**: **Secure Shell** (**SSH**) (*RFC 4253*) is a secure application-layer protocol that encapsulates strong encryption and cryptographic host authentication. SSH uses a virtual terminal connection between a client and a server machine. SSH operates at the application layer (*Layer 7*) in the OSI model.
- **SMTP**: SMTP (*RFC 5321*) is an application-layer protocol for sending and receiving emails between an email client (for example, Outlook) and an email server (such as Exchange Server). SMTP supports strong encryption and host authentication. SMTP acts at the application layer (*Layer 7*) in the OSI model.
- **SNMP**: The **Simple Network Management Protocol** (**SNMP**) (*RFC 1157*) is used for remote device management and monitoring. SNMP operates at the application layer (*Layer 7*) in the OSI model.
- **NTP**: The **Network Time Protocol** (**NTP**) (*RFC 5905)* is an internet protocol that's used for synchronizing the system clock of multiple machines across a network. NTP operates at the application layer (*Layer 7*) in the OSI model.

Most of the internet protocols enumerated previously use the IP protocol to identify devices participating in the communication. Devices on a network are uniquely identified by an IP address. Let's examine these network addresses more closely.

IP addresses

An **IP address** is a fixed-length **unique identifier** (**UID**) of a device in a network. Devices locate and communicate with each other based on IP addresses. The concept of an IP address is very similar to a postal address of a residence, whereby mail or a package would be sent to that destination based on its address.

Initially, IP defined the IP address as a 32-bit number known as an **IPv4 address**. With the growth of the internet, the total number of IP addresses in a network has been exhausted. To address this issue, a new version of the IP protocol devised a 128-bit numbering scheme for IP addresses. A 128-bit IP address is also known as an **IPv6 address**.

In the next few sections, we'll take a closer look at the networking constructs that play an important role in IP addresses, such as IPv4 and IPv6 address formats, network classes, subnetworks, and broadcast addresses.

IPv4 addresses

An **IPv4 address** is a 32-bit number (4 bytes) usually expressed as four groups of 1-byte (8 bits) numbers, separated by a dot (`.`). Each number in these four groups is an integer between `0` and `255`. An example of an IPv4 address is `192.168.1.53`.

The following figure shows a binary representation of an IPv4 address:

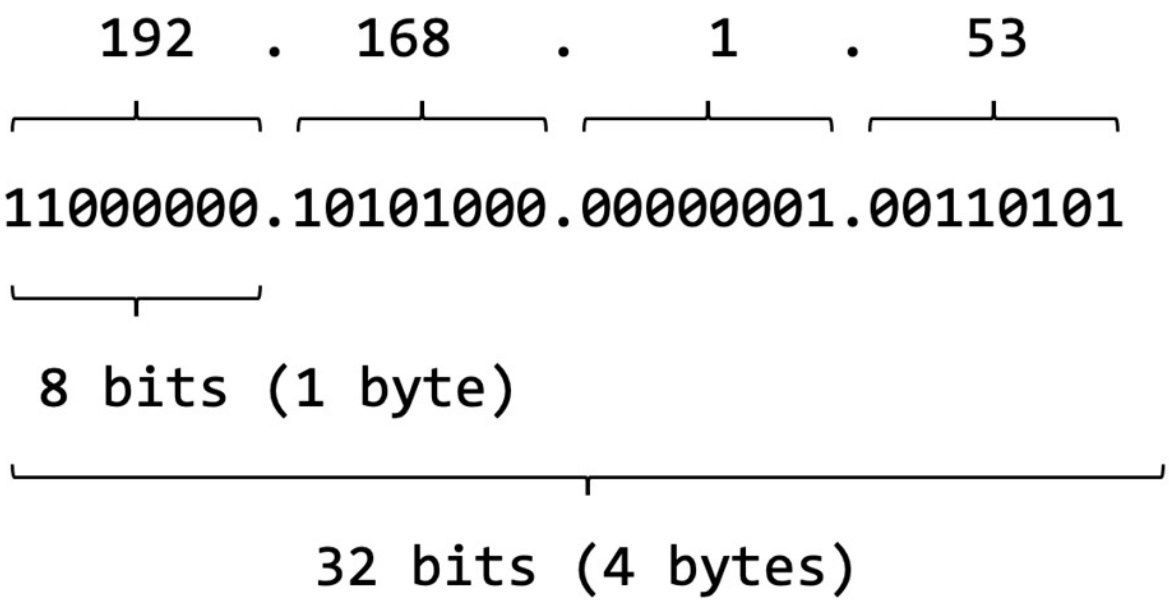

Figure 7.4 – Network classes

The IPv4 address space is limited to 4,294,967,296 (2^32) addresses (roughly 4 billion). Of these, approximately 18 million are reserved for special purposes (for example, private networks), and about 270 million are multicast addresses.

A **multicast address** is a logical identifier of a group of IP addresses. For more information on multicast addresses, please refer to *RFC 6308* (https://tools.ietf.org/html/rfc6308).

Network classes

In the early stages of the internet, the highest-order byte (first group) in the IPv4 address indicated the **network number**. The subsequent bytes further express the network hierarchy and subnetworks, with the lowest-order byte identifying the device itself. This scheme soon proved insufficient for network hierarchies and segregations as it only allowed for 256 (2^8) networks, denoted by the leading byte of the IPv4 address. As additional networks were added, each with its own identity, the IP address specification needed a special revision to accommodate a standard model. The *Classful Network* specification, which was introduced in 1981, addressed this problem by dividing the IPv4 address space into five classes based on the leading 4 bits of the address, as illustrated in the following figure:

Class	Leading bits	Start address	End address	Default subnet mask
Class A	0	0.0.0.0	127.255.255.255	255.0.0.0
Class B	10	128.0.0.0	191.255.255.255	255.255.0.0
Class C	110	192.0.0.0	223.255.255.255	255.255.255.0
Class D (multicast)	1110	224.0.0.0	239.255.255.255	Not defined
Class E (reserved)	1111	240.0.0.0	255.255.255.255	Not defined

Figure 7.5 – Network classes

For more information on network classes, please refer to *RFC 870* (https://tools.ietf.org/html/rfc870). In the preceding figure, the last column specifies the default subnet mask for each of these network classes. We'll look at subnets (or subnetworks) next.

Subnetworks

Subnetworks (or **subnets**) are logical subdivisions of an IP network. Subnets were introduced to identify devices that belong to the same network. The IP addresses of devices in the same network have an identical most-significant group. The subnet definition yields a logical division of an IP address into two fields: the **network identifier** and the **host identifier**. The numerical representation

of the subnet is called a **subnet mask** or **netmask**. The following figure provides an example of a network identifier and a host identifier:

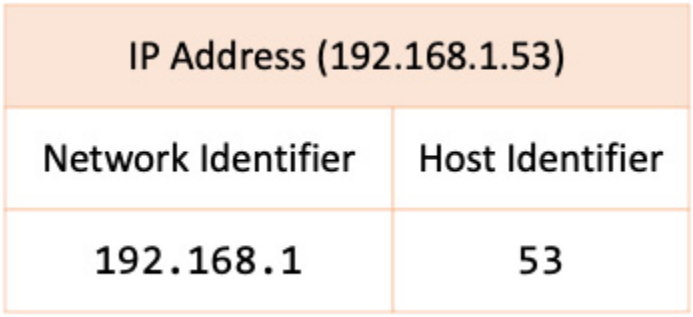

Figure 7.6 – Subnet with network and host identifiers

With our IPv4 address (`192.168.1.53`), we could devise a network identifier of `192.168.1`, where the host identifier is `53`. The resulting subnet mask would be as follows:

```
192.168.1.0
```

We dropped the least significant group in the subnet mask, representing the host identifier (`53`), and replaced it with `0`. Here, `0` indicates the starting address in the subnet. In other words, any host identifier value in the range of `0` to `255` is allowed in the subnetwork. For example, `192.168.1.92` is a valid (and accepted) IP address in the `192.168.1.0` network.

An alternative representation of subnets uses so-called **Classless Inter-Domain Routing** (**CIDR**) notation. CIDR represents an IP address as the network address (*prefix*) followed by a slash (`/`) and the *bit-length* of the prefix. In our case, the CIDR notation of the `192.168.1.0` subnet is this:

```
192.168.1.0/24
```

The first three groups in the network address make up *3 x 8 = 24* bits, hence the `/24` notation.

Usually, **subnets** are planned with the host identifier address as a starting point. Back to our example, suppose we wanted our host identifier addresses in the network to start with `100` and end with `125`. Let's look at how we can achieve this.

First, let's see the binary representation of `192.168.1.100`:

```
11000000.10101000.00000001.01100100
```

As you can see, the last group, highlighted in the preceding sequence, represents the host identifier (`100`). Remember that we want the network to start with 100 and end with 125. This means that the closest binary value to the reserved 99 addresses that would not be permitted in our subnet is *96 = 64 + 32*. The equivalent binary value for it is as follows:

```
111
```
00000

In other words, the three most significant bits in the host identifier are reserved. **Reserved bits** in the subnet representation are shown as `1`. These bits would be added to the 24 already reserved bits of the network address (`192.168.1`), accounting in total for *27 = 24 + 3* bits. Here's the equivalent representation:

```
11111111.11111111.11111111.11100000
```

Consequently, the resulting netmask is as follows:

```
255.255.255.224
```

The CIDR notation of the corresponding subnet is shown here:

```
192.168.1.96/27
```

The remaining five bits in the host identifier's group account for *2^5 = 32* possible addresses in the subnet, starting with `97`. This would limit the maximum host identifier value to *127 = 96 + 32 – 1* (we subtract 1 to account for the starting number of 97 included in the total of 32). In this range of 32 addresses, the last IP address is reserved as a **broadcast address**, as shown here:

```
192.168.1.127
```

A broadcast address is reserved as the highest number in a network or subnet, when applicable. Back to our example, excluding the broadcast address, the maximum host IP address in the subnet is as follows:

```
192.168.1.126
```

You can learn more about subnets in *RFC 1918* (https://tools.ietf.org/html/rfc1918). Since we mentioned the broadcast address, let's have a quick look at it.

Broadcast addresses

A **broadcast address** is a reserved IP address in a network or subnetwork that's used to transmit a collective message (data) to all devices belonging to the network. The broadcast address is the last IP address in the network or subnet, when applicable.

For example, the broadcast address of the `192.168.1.0/24` network is `192.168.1.255`. In our example in the previous section, the broadcast address of the `192.168.1.96/27` subnet is `192.168.1.127` (*127 = 96 + 32 – 1*).

For more information on broadcast addresses, you can refer to https://www.sciencedirect.com/topics/computer-science/broadcast-address.

IPv6 addresses

An IPv6 address is a 128-bit number (16 bytes) that's usually expressed as up to eight groups of 2-byte (16 bits) numbers, separated by a column (`:`). Each number in these eight groups is a hexadecimal number, with values between `0000` and `FFFF`. Here's an example of an IPv6 address:

```
2001:0b8d:8a52:0000:0000:8b2d:0240:7235
```

An equivalent representation of the preceding IPv6 address is shown here:

```
2001:b8d:8a52::8b2d:240:7235/64
```

In the second representation, the leading zeros are omitted, and the all-zero groups (`0000:0000`) are collapsed into an empty group (`::`). The `/64` notation at the end represents the **prefix length** of the IPv6 address. The IPv6 prefix length is the equivalent of the CIDR notation of IPv4 subnets. For IPv6, the prefix length is expressed as an integer value between `1` and `128`.

In our case, with a prefix length of 64 (*4 x 16*) bits, the subnet looks like this:

```
2001:b8d:8a52::
```

The subnet represents the leading four groups (`2001`, `0b8d`, `8a52`, and `0000`), which results in a total of *4 x 16 = 64* bits. In the shortened representation of the IPv6 subnet, the leading zeros are omitted and the all-zero group is collapsed to `::`.

Subnetting with IPv6 is very similar to IPv4. We won't go into the details here since the related concepts were presented in the *IPv4 addresses* section. For more information about IPv6, please refer to *RFC 2460* (https://tools.ietf.org/html/rfc2460).

Now that you've become familiar with IP addresses, it is fitting to introduce some of the related network constructs that serve the software implementation of IP addresses – that is, sockets and ports.

Sockets and ports

A **socket** is a software data structure representing a network node for communication purposes. Although a programming concept, in Linux, a **network socket** is ultimately a file descriptor that's controlled via a network **application programming interface** (**API**). A socket is used by an application process for transmitting and receiving data. An application can create and delete sockets. A socket cannot be active (send or receive data) beyond the lifetime of the process that created the socket.

Network sockets operate at the *transport-layer* level in the OSI model. There are two endpoints to a socket connection – a sender and a receiver. Both the sender and receiver have an IP address.

Consequently, a critical piece of information in the socket data structure is the *IP address* of the endpoint that owns the socket.

Both endpoints create and manage their sockets via the network processes using these sockets. The sender and receiver may agree upon using multiple connections to exchange data. Some of these connections may even run in parallel. How do we differentiate between these socket connections? The IP address by itself is not sufficient, and this is where **ports** come into play.

A **network port** is a logical construct that's used to identify a specific process or network service running on a host. A port is an integer value in the range of `0` and `65535`. Usually, ports in the range of `0` and `1024` are assigned to the most used services on a system. These ports are also called **well-known ports**. Here are a few examples of well-known ports and the related network service for each:

- `25`: SMTP
- `21`: FTP
- `22`: SSH
- `53`: DNS
- `67`, `68`: DHCP (client = `68`, server = `67`)
- `80`: HTTP
- `443`: **HTTP Secure** (**HTTPS**)

Port numbers beyond `1024` are for general use and are also known as **ephemeral ports**.

A port is always associated with an IP address. Ultimately, a socket is a combination of an IP address and a port. For more information on network sockets, you can refer to *RFC 147* (https://tools.ietf.org/html/rfc147). For well-known ports, see *RFC 1340* (https://tools.ietf.org/html/rfc1340).

Now, let's put the knowledge we've gained so far to work by looking at how to configure the local networking stack in Linux.

Linux network configuration

This section describes the TCP/IP **network configuration** for Ubuntu and Fedora platforms, using their latest released versions to date. The same concepts would apply to most Linux distributions, albeit some of the network configuration utilities and files involved could be different.

We can use the `ip` command-line utility to retrieve the system's current IP addresses, as follows:

```
ip addr show
```

An example of the output is shown here:

```
packt@neptune:~$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP gr
oup default qlen 1000
    link/ether 52:54:00:d6:7d:69 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.117/24 metric 100 brd 192.168.122.255 scope global dynamic
enp1s0
       valid_lft 3071sec preferred_lft 3071sec
    inet6 fe80::5054:ff:fed6:7d69/64 scope link
       valid_lft forever preferred_lft forever
```

Figure 7.7 – Retrieving the current IP addresses with the ip command

We've highlighted some relevant information here, such as the network interface ID (`2: enp1s0`) and the IP address with the subnet prefix (`192.168.122.117/24`).

We'll look at Ubuntu's network configuration next. At the time of writing this book, the released version of Ubuntu is 22.04.2 LTS.

Ubuntu network configuration

Ubuntu 22.04 provides the `netplan` command-line utility for easy network configuration. `netplan` uses a **YAML Ain't Markup Language** (**YAML**) configuration file to generate the network interface bindings. The `netplan` configuration file(s) is in the `/etc/netplan/` directory, and we can access it by using the following command:

```
ls /etc/netplan/
```

In our case, the configuration file is `00-installer-config.yaml`.

Changing the network configuration involves editing the `netplan` YAML configuration file. As good practice, we should always make a backup of the current configuration file before making changes. Changing a network configuration would most commonly involve setting up either a dynamic or a static IP address. We will show you how to configure both types in the next few sections. We'll look at dynamic IP addressing first.

Dynamic IP configuration

To enable a dynamic (DHCP) IP address, we must edit the `netplan` configuration file and set the `dhcp4` attribute to `true` (as shown in *Figure 7.8*) for the network interface of our choice (`ens33`, in our case). Open the `00-installer-config.yaml` file with your text editor of choice (nano, in our case):

```
sudo nano /etc/netplan/00-installer-config.yaml
```

Here's the related configuration excerpt, with the relevant points highlighted:

Figure 7.8 – Enabling DHCP in the netplan configuration

After saving the configuration file, we can test the related changes with the following command:

```
sudo netplan try
```

We'll get the following response:

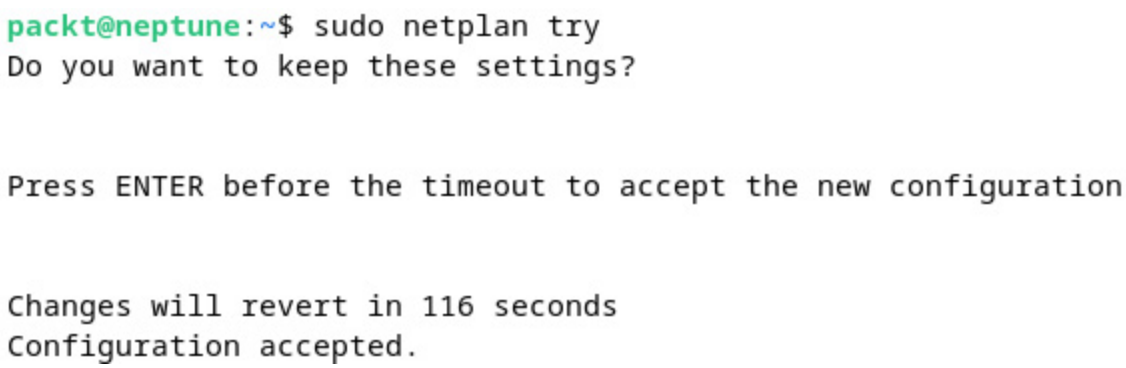

Figure 7.9 – Testing and accepting the netplan configuration changes

The `netplan` keyword validates the new configuration and prompts the user to accept the changes. The following command applies the current changes to the system:

```
sudo netplan apply
```

Next, we will configure a static IP address using `netplan`.

Static IP configuration

To set the static IP address of a network interface, we start by editing the `netplan` configuration YAML file, as follows:

```
sudo nano /etc/netplan/00-installer-config.yaml
```

Here's a configuration example with a static IP address of `192.168.122.22/24`:

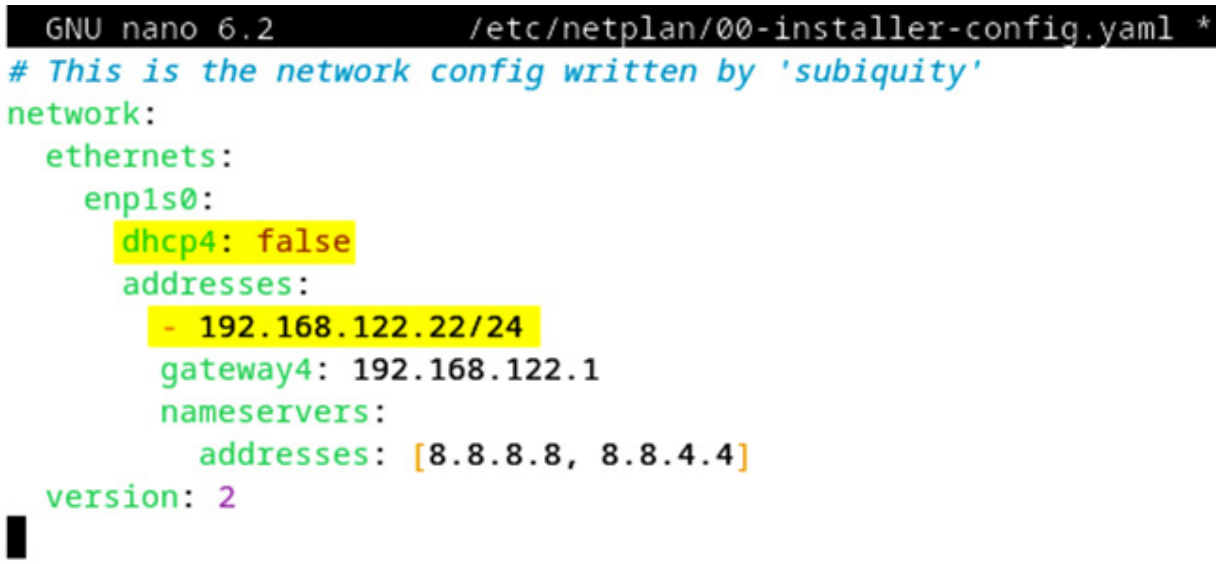

Figure 7.10 – Static IP configuration example with netplan

After saving the configuration, we can test and accept it, and then apply changes, as we did in the *Dynamic IP* section, with the following commands:

```
sudo netplan try
sudo netplan apply
```

For more information on the `netplan` command-line utility, see `netplan --help` or the related system manual (`man netplan`).

We'll look at the Fedora network configuration next. At the time of writing, the current released version of Fedora is 37.

Fedora/RHEL network configuration

Starting with Fedora 33 and RHEL 9, network configuration files are *no longer* kept in the `/etc/sysconfig/network-scripts/` directory. To learn more about the new configuration options, read the following file:

```
cat /etc/sysconfig/network-scripts/readme-ifcfg-rh.txt
```

The preferred method to configure the network in Fedora/RHEL is to use the `nmcli` utility. This location is deprecated and no longer used by NetworkManager in Fedora; it can still be used, but we

do not recommend it. The new NetworkManager keyfiles are stored inside the `/etc/NetworkManager/system-connections/` directory.

Let's use some basic `nmcli` commands to view information about our connections. To learn about `nmcli`, read the related manual pages. First, let's find information about our active connection using the following command:

```
nmcli connection show
```

The output will show basic information about the name of the connection, UUID, type, and the device used. The following screenshot shows the relevant information on our Fedora 37 VM:

```
[packt@saturn ~]$ nmcli connection show
NAME                UUID                                  TYPE      DEVICE
Wired connection 1  ce007bd0-cc92-3a30-a367-95ecbf728ab7  ethernet  enp1s0
```

Figure 7.11 – Using nmcli to view connection information

Similar to Ubuntu, when using Fedora/RHEL, changing a network configuration would most commonly involve setting up either a dynamic or a static IP address. We will show you how to configure both types in the following sections. Let's look at dynamic IP addressing first.

Dynamic IP configuration

To configure a dynamic IP address using `ncmli`, we can run the following command:

```
sudo nmcli connection modify 'Wired connection 1' ipv4.method auto
```

The `ipv4.method auto` directive enables DHCP. There is no output from the command; after execution, you will be returned to the prompt again. You can check if the command worked by viewing the `/etc/NetworkManager/system-connections/` directory. In our case, there is a new keyfile inside. It has the same name as our connection. The following is an excerpt:

```
[packt@saturn ~]$ sudo cat /etc/NetworkManager/system-connections/Wired\ connect
ion\ 1.nmconnection
[sudo] password for packt:
[connection]
id=Wired connection 1
uuid=ce007bd0-cc92-3a30-a367-95ecbf728ab7
type=ethernet
autoconnect-priority=-999
interface-name=enp1s0
timestamp=1679137326

[ethernet]

[ipv4]
method=auto

[ipv6]
addr-gen-mode=default
method=auto
```

Figure 7.12 – New configuration keyfile

Next, we'll configure a static IP address.

Static IP configuration

To perform the equivalent static IP address changes using `ncmli`, we need to run multiple commands. First, we must set the static IP address, as follows:

```
sudo nmcli connection modify 'Wired connection 1' ipv4.address 192.168.122.3/24
```

If no previous static IP address has been configured, we recommend saving the preceding change before proceeding with the next steps. The changes can be saved with the following code:

```
sudo nmcli connection down 'Wired connection 1'
sudo nmcli connection up 'Wired connection 1'
```

Next, we must set the gateway and DNS IP addresses, as follows:

```
sudo nmcli connection modify 'Wired connection 1' ipv4.gateway 192.168.122.1
sudo nmcli connection modify 'Wired connection 1' ipv4.dns 8.8.8.8
```

Finally, we must disable DHCP with the following code:

```
sudo nmcli connection modify 'Wired connection 1' ipv4.method manual
```

After making these changes, we need to restart the `'Wired connection 1'` network interface with the following code:

```
sudo nmcli connection down 'Wired connection 1'
sudo nmcli connection up 'Wired connection 1'
```

Now, let's see the results of all the commands we've performed. After bringing the connection up again, let's check the new IP address and the contents of the network keyfile. The following figure shows the new IP we assigned to the system (`192.168.122.3`) by using the `ip addr show` command:

```
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP gr
oup default qlen 1000
    link/ether 52:54:00:37:a2:73 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.3/24 brd 192.168.122.255 scope global noprefixroute enp1s0
       valid_lft forever preferred_lft forever
    inet6 fe80::d78a:3dd9:7af7:c8d2/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

Figure 7.13 – Checking the new IP address

Now, let's view the contents of the network keyfile to see the changes that were made for static IP configuration. Remember that the location of the file is `/etc/NetworkManager/system-connections/`:

```
[connection]
id=Wired connection 1
uuid=ce007bd0-cc92-3a30-a367-95ecbf728ab7
type=ethernet
autoconnect-priority=-999
interface-name=enp1s0
timestamp=1679147993

[ethernet]

[ipv4]
address1=192.168.122.3/24,192.168.122.1
dns=8.8.8.8;
method=manual
```

Figure 7.14 – New keyfile configuration for the static IP address

The `nmcli` utility is a powerful and useful one. At the end of this chapter, we will provide you with some useful links for learning more about it. Next, we'll take a look at how to configure network services on openSUSE.

openSUSE network configuration

openSUSE provides several tools for network configuration: **Wicked** and NetworkManager. According to the official SUSE documentation, Wicked is used for all types of machines, from servers to laptops and workstations, whereas NetworkManager is used only for laptop and workstation setup and is not used for server setup. However, in openSUSE Leap, Wicked is set up by

default on both desktop or server configurations, and NetworkManager is set up by default on laptop configurations.

For example, on our main workstation (which is a laptop), if we want to see which service is running by default on openSUSE Leap, we can use the following command:

```
sudo systemctl status network
```

The output will show us which service is running, and in our case, it is NetworkManager:

```
alexandru@localhost:~> sudo systemctl status network
[sudo] password for root:
● NetworkManager.service - Network Manager
     Loaded: loaded (/usr/lib/systemd/system/NetworkManager.service; enabled; v>
    Drop-In: /usr/lib/systemd/system/NetworkManager.service.d
             └─NetworkManager-ovs.conf
     Active: active (running) since Sun 2023-03-19 12:06:46 EET; 33min ago
       Docs: man:NetworkManager(8)
   Main PID: 1387 (NetworkManager)
```

Figure 7.15 – Checking which network service is running in openSUSE

When running the same command inside an openSUSE Leap server VM, the result is different. The output shows that Wicked is running by default. The following screenshot shows an example:

```
packt@localhost:~> sudo systemctl status network
● wicked.service - wicked managed network interfaces
     Loaded: loaded (/usr/lib/systemd/system/wicked.service; enabled; vendor pr>
     Active: active (exited) since Sun 2023-03-19 13:42:07 EET; 2min 18s ago
   Main PID: 769 (code=exited, status=0/SUCCESS)
```

Figure 7.16 – Wicked is running inside the openSUSE server

As a result, we will perform all the examples in this section on an openSUSE Leap server VM, thus using Wicked as the default network configuration tool. In the next section, we will configure dynamic IP on an openSUSE machine.

Dynamic IP configuration

Before setting anything up, let's check for active connections and devices. We can do this by using the following command:

```
wicked show all
```

The output will show all the active devices. The following screenshot shows an excerpt from the output on our machine:

```
eth0            up
      link:     #2, state up, mtu 1500
      type:     ethernet, hwaddr 52:54:00:dc:4e:e0
      config:   compat:suse:/etc/sysconfig/network/ifcfg-eth0
      leases:   ipv4 dhcp granted
      leases:   ipv6 dhcp requesting
      addr:     ipv4 192.168.122.146/24 [dhcp]
      route:    ipv4 default via 192.168.122.1 proto dhcp
```

Figure 7.17 – Information about active devices

There are two active connections, one is loopback (`lo`) and the other is on the ethernet port (`eth0`). We will only show information related to `eth0`. The location where Wicked stores configuration files in openSUSE is `/etc/sysconfig/network`. If we do a listing of that directory, we will see that it is already populated with configuration files for existing connections:

```
packt@localhost:~> ls -la /etc/sysconfig/network/
total 64
drwxr-xr-x 1 root root   176 Mar  3 21:00 .
drwxr-xr-x 1 root root   584 Mar 11 14:18 ..
-rw-r--r-- 1 root root  9691 Mar  3 21:00 config
-rw-r--r-- 1 root root 15678 Mar  3 21:00 dhcp
-rw------- 1 root root    46 Mar  3 21:00 ifcfg-eth0
-rw-r--r-- 1 root root    34 Mar  3 21:00 ifcfg-eth0.bak
-rw------- 1 root root   147 Mar  3 21:00 ifcfg-lo
-rw-r--r-- 1 root root 21738 Aug  1  2022 ifcfg.template
drwxr-xr-x 1 root root     0 Mar 15  2022 if-down.d
drwxr-xr-x 1 root root     0 Mar 15  2022 if-up.d
drwx------ 1 root root     0 Mar 15  2022 providers
drwxr-xr-x 1 root root    38 Mar  3 20:59 scripts
```

Figure 7.18 – Location of the Wicked configuration files

In our case, and maybe it will be the same for you, the file we are interested in is called `ifcfg-eth0`; we can open it with a text editor or concatenate it. Let's take a look at the contents of the file:

```
  GNU nano 6.3                /etc/sysconfig/network/ifcfg-eth0
BOOTPROTO='dhcp'
STARTMODE='auto'
ZONE=public
```

Figure 7.19 – Information provided by the configuration file

As shown in the preceding screenshot, the information provided is rather scarce but relevant. For a more detailed output, we can use the following command:

```
sudo wicked show-config
```

This will provide much more relevant information directly to the monitor. The following screenshot provides an excerpt from our output, with detailed IPv4 DHCP information:

```
<ipv4>
  <enabled>true</enabled>
  <arp-verify>true</arp-verify>
</ipv4>
<ipv4:dhcp>
  <enabled>true</enabled>
  <flags>group</flags>
  <update>default-route,hostname,dns,nis,ntp,nds,mtu,tz,boot</update>
  <defer-timeout>15</defer-timeout>
  <recover-lease>true</recover-lease>
  <release-lease>false</release-lease>
</ipv4:dhcp>
```

Figure 7.20 – Detailed information provided by Wicked

To better understand this output, we recommend reading the `config` file and, if available, the `dhcp` files inside the `/etc/sysconfig/network` directory. These files provide information about specific variables and default parameters needed for configuring the network devices.

Dynamic IP addresses are usually set up by default when installing the operating system. If yours is not configured, all you need to do is create a configuration file inside `/etc/sysconfig/network` and give it a relevant name based on the device you are using to connect to the network, such as `ifcfg-eth0`. Inside that file, you will have to provide just three lines, as seen in *Figure 7.19*. The following commands are needed for the actions described in this paragraph:

- Check your device name using the `ip addr` command:

  ```
  ip addr show
  ```

- Create a configuration file inside the `/etc/sysconfig/network` directory:

  ```
  sudo nano /etc/sysconfig/network/ifcfg-eth0
  ```

- Provide relevant information for the DHCP configuration:

  ```
  BOOTPROTO='dhcp'
  STARTMODE='auto'
  ZONE=public
  ```

- Restart the Wicked service:

```
sudo systemctl restart wicked
```

- Check for connectivity by using either a web browser on your system or the **ping** command. Here's an example:

```
ping google.com
```

In the following section, we will show you how to set up a static IP configuration.

Static IP configuration

To set up a static IP configuration, you would need to manually provide variables for the configuration files. These files are the same ones that were presented in the previous section. The location of the files is **/etc/sysconfig/network**. For example, you can create a new file for the **eth0** device connection and provide the information you want. Let's look at an example of using our openSUSE Leap server VM. However, before doing this, we advise you to open the manual pages for the **ifcfg** utility as they provide valuable information on the variables used for this exercise.

Therefore, we will set the **ifcfg** configuration file as follows:

- First, we will check for the IP address and the network device name; in our case, the dynamically allocated IP is **192.168.122.146** and the device's name is **eth0**:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP gr
oup default qlen 1000
    link/ether 52:54:00:dc:4e:e0 brd ff:ff:ff:ff:ff:ff
    altname enp1s0
    inet 192.168.122.146/24 brd 192.168.122.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::5054:ff:fedc:4ee0/64 scope link
       valid_lft forever preferred_lft forever
```

Figure 7.21 – IP and device information

- Go to the **/etc/sysconfig/network** directory and edit the **ifcfg-eth0** file; we will use the following variables for static IP configuration:
    - **BOOTPROTO='static'**: This allows us to use a fixed IP address that will be provided by using the IPADDR variable
    - **STARTMODE='auto'**: The interface will be automatically enabled on boot
    - **IPADDR='192.168.122.144'**: The IP address we choose for the machine
    - **ZONE='public'**: The zone used by the **firewalld** utility
    - **PREFIXLEN='24'**: The number of bits in the **IPADDR** variable

    This will look as follows:

```
  GNU nano 6.3                          ifcfg-eth0
BOOTPROTO='static'
STARTMODE='auto'
ZONE=public
IPADDR='192.168.122.144'
PREFIXLEN='24'
```

Figure 7.22 – New variables for static IP configuration

- Save the changes to the new file and restart the Wicked daemon:

```
sudo systemctl restart wickedd.service
```

- Enable the interface so that the changes appear:

```
sudo wicked ifup eth0
```

- Run the **`ip addr show`** command again to check for the new IP address:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP gr
oup default qlen 1000
    link/ether 52:54:00:dc:4e:e0 brd ff:ff:ff:ff:ff:ff
    altname enp1s0
    inet 192.168.122.146/24 brd 192.168.122.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet 192.168.122.144/24 brd 192.168.122.255 scope global secondary eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::5054:ff:fedc:4ee0/64 scope link
       valid_lft forever preferred_lft forever
```

Figure 7.23 – New IP address assigned to eth0

At this point, you know how to configure networking devices in all major Linux distributions, using their preferred utilities. We've merely scratched the surface of this matter, but we've provided sufficient information for you to start working with network interfaces in Linux. For more information, feel free to read the manual pages freely provided with your operating system. In the next section, we will approach the matter of hostname configuration.

Hostname configuration

To retrieve the current hostname on a Linux machine, we can use either the **`hostname`** or **`hostnamectl`** command, as follows:

```
hostname
```

The most convenient way to change the hostname is with the **`hostnamectl`** command. We can change the hostname to **`earth`** using the **`set-hostname`** parameter of the command:

```
sudo hostnamectl set-hostname earth
```

Let's verify the hostname change with the `hostname` command again. You could use the `hostnamectl` command to verify the hostname. The output of the `hostnamectl` command provides more detailed information compared to the `hostname` command, as shown in the following screenshot:

```
packt@neptune:~$ hostname
neptune
packt@neptune:~$ sudo hostnamectl set-hostname earth
packt@neptune:~$ hostname
earth
packt@neptune:~$ hostnamectl
 Static hostname: earth
       Icon name: computer-vm
         Chassis: vm
      Machine ID: 7c7f082fb11d431a9823a11ba9e87ad0
         Boot ID: f7eb64629ddf4424836760dfab2e9146
  Virtualization: kvm
Operating System: Ubuntu 22.04.2 LTS
          Kernel: Linux 5.15.0-67-generic
    Architecture: x86-64
 Hardware Vendor: QEMU
  Hardware Model: Standard PC _Q35 + ICH9, 2009_
```

Figure 7.24 – Retrieving the current hostname with the different commands

Alternatively, we can use the `hostname` command to change the hostname *temporarily*, as follows:

```
sudo hostname jupiter
```

However, this change will not survive a reboot unless we also change the hostname in the `/etc/hostname` and `/etc/hosts` files. When editing these two files, change your hostname accordingly. The following screenshot shows the succession of commands:

```
packt@neptune:~$ hostname
earth
packt@neptune:~$ sudo hostname neptune
packt@neptune:~$ cat /etc/hostname
earth
packt@neptune:~$ sudo nano /etc/hostname
packt@neptune:~$ sudo nano /etc/hosts
packt@neptune:~$ hostname
neptune
```

Figure 7.25 – The /etc/hostname and /etc/hosts files

After the hostname reconfiguration, a logout followed by a login would usually reflect the changes. Hostnames are important for coherent network management, where each system on the network should have relevant hostnames set up. In the following section, you'll learn about network services in Linux.

Working with network services

In this section, we'll enumerate some of the most common network services running on Linux. Not all the services mentioned here are installed or enabled by default on your Linux platform of choice. *Chapter 9*, *Securing Linux*, and *Chapter 10*, *Disaster Recovery, Diagnostics, and Troubleshooting*, will dive deeper into how to install and configure some of them. Our focus in this section remains on what these network services are, how they work, and the networking protocols they use for communication.

A **network service** is typically a system process that implements application layer (OSI *Layer 7*) functionality for data communication purposes. Network services are usually designed as peer-to-peer or client-server architectures.

In peer-to-peer networking, multiple network nodes each run their own equally privileged instance of a network service while sharing and exchanging a common set of data. Take, for example, a network of DNS servers, all sharing and updating their domain name records.

Client-server networking usually involves one or more server nodes on a network and multiple clients communicating with any of these servers. An example of a client-server network service is SSH. An SSH client connects to a remote SSH server via a secure Terminal session, perhaps for remote administration purposes.

Each of the following subsections briefly describes a network service, and we encourage you to explore topics related to these network services in *Chapter 13* or other relevant titles recommended at the end of this chapter. Let's start with DHCP servers.

DHCP servers

A **DHCP server** uses the DHCP protocol to enable devices on a network to request an IP address that's been assigned dynamically. The DHCP protocol was briefly described in the *TCP/IP protocols* section earlier in this chapter.

A computer or device requesting a DHCP service sends out a broadcast message (or query) on the network to locate a DHCP server, which, in turn, provides the requested IP address and other

information. Communication between the DHCP client (device) and the server uses the DHCP protocol.

The DHCP protocol's initial *discovery* workflow between a client and a server operates at the data link layer (*Layer 2*) in the OSI model. Since Layer 2 uses network frames as PDUs, the DHCP discovery packets cannot transcend the local network boundary. In other words, a DHCP client can only initiate communication with a *local* DHCP server.

After the initial *handshake* (on Layer 2), DHCP turns to UDP as its transport protocol, using datagram sockets (*Layer 4*). Since UDP is a connectionless protocol, a DHCP client and server exchange messages without a prior arrangement. Consequently, both endpoints (client and server) require a well-known DHCP communication port for the back-and-forth data exchange. These are the well-known ports `68` (for a DHCP server) and `67` (for a DHCP client).

A DHCP server maintains a collection of IP addresses and other client configuration data (such as MAC addresses and domain server addresses) for each device on the network requesting a DHCP service.

DHCP servers use a **leasing mechanism** to assign IP addresses dynamically. Leasing an IP address is subject to a **lease time**, either finite or infinite. When the lease of an IP address expires, the DHCP server may reassign it to a different client upon request. A device would hold on to its dynamic IP address by regularly requesting a **lease renewal** from the DHCP server. Failing to do so would result in the potential loss of the device's dynamic IP address. A late (or post-lease) DHCP request would possibly result in a new IP address being acquired if the previous address had already been allocated by the DHCP server.

A simple way to query the DHCP server from a Linux machine is by invoking the following command:

```
ip route
```

This is the output of the preceding command:

```
packt@neptune:~$ ip route
default via 192.168.122.1 dev enp1s0 proto dhcp src 192.168.122.117 metric 100
192.168.122.0/24 dev enp1s0 proto kernel scope link src 192.168.122.117 metric 1
00
192.168.122.1 dev enp1s0 proto dhcp scope link src 192.168.122.117 metric 100
```

Figure 7.26 – Querying the IP route for DHCP information

The first line of the output provides the DHCP server (`192.168.122.1`).

Chapter 13, *Configuring Linux Servers*, will further go into the practical details of installing and configuring a DHCP server.

For more information on DHCP, please refer to *RFC 2131* (https://tools.ietf.org/html/rfc2131).

DNS servers

A **Domain Name Server** (**DNS**), also known as a **name server**, provides a name-resolution mechanism by converting a hostname (such as `wikipedia.org`) into an IP address (such as `208.80.154.224`). The name-resolution protocol is DNS, briefly described in the *TCP/IP protocols* section earlier in this chapter. In a DNS-managed TCP/IP network, computers and devices can also identify and communicate with each other via hostnames, not just IP addresses.

As a reasonable analogy, DNS very much resembles an address book. Hostnames are relatively easier to remember than IP addresses. Even in a local network, with only a few computers and devices connected, it would be rather difficult to identify (or memorize) any of the hosts by simply using their IP address. The internet relies on a globally distributed network of DNS servers.

There are four different types of DNS servers: **recursive servers**, **root servers**, **top-level domain** (**TLD**) **servers**, and **authoritative servers**. All these DNS server types work together to bring you the internet as you experience it in your browser.

A **recursive DNS server** is a resolver that helps you find the destination (IP) of a website you search for. When you perform a lookup operation, a recursive DNS server is connected to different DNS servers to find the IP address that you are looking for and return it to you in the form of a website. Recursive DNS lookups are faster as they cache every query that they perform. In a recursive type of query, the DNS server calls itself and does the recursion while still sending the request to another DNS server to find the answer.

In contrast, an **iterative DNS** lookup is done by every DNS server directly, without using caching. For example, in an iterative query, each DNS server responds with the address of another DNS server, until one of them has the matching IP address for the hostname in question and responds to the client. For more details on DNS server types, please check out the following Cloudflare learning solution: https://www.cloudflare.com/learning/dns/what-is-dns/.

DNS servers maintain (and possibly share) a collection of **database files**, also known as **zone files**, which are typically simple plaintext ASCII files that store the name and IP address mapping. In Linux, one such DNS resolver file is `/etc/resolv.conf`.

To query the DNS server managing the local machine, we can query the `/etc/resolv.conf` file by running the following code:

```
cat /etc/resolv.conf | grep nameserver
```

The preceding code yields the following output:

```
packt@neptune:~$ cat /etc/resolv.conf | grep nameserver
nameserver 127.0.0.53
packt@neptune:~$
```

Figure 7.27 – Querying DNS server using /etc/resolv.conf

A simple way to query name-server data for an arbitrary host on a network is by using the `nslookup` tool. If you don't have the `nslookup` utility installed on your system, you may do so with the commands outlined here.

On Ubuntu/Debian, run the following command:

```
sudo apt install dnsutils
```

On Fedora, run this command:

```
sudo dnf install bind-utils
```

For example, to query the name-server information for a computer named `neptune.local` in our local network, we can run the following command:

```
nslookup neptune.local
```

The output is shown here:

```
packt@neptune:~$ nslookup neptune.local
Server:         127.0.0.53
Address:        127.0.0.53#53

Name:   neptune.local
Address: 192.168.122.117
```

Figure 7.28 – Querying name-server information with nslookup

We can also use the `nslookup` tool interactively. For example, to query the name-server information for `wikipedia.org`, we can simply run the following command:

```
nslookup
```

Then, in the interactive prompt, we must enter `wikipedia.org`, as illustrated here:

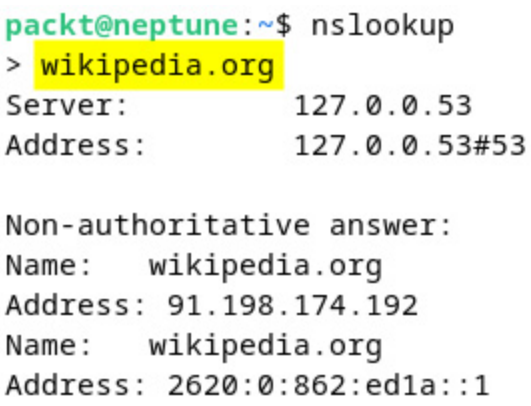

Figure 7.29 – Using the nslookup tool interactively

To exit the interactive shell mode, press *Ctrl* + *C*. Here's a brief explanation of the information shown in the preceding output:

- **Server (address)**: The loopback address (`127.0.0.53`) and port (`53`) of the DNS server running locally
- **Name**: The internet domain we're looking up (`wikipedia.org`)
- **Address**: The IPv4 (`91.198.174.192`) and IPv6 (`2620:0:862:ed1a::1`) addresses that correspond to the lookup domain (`wikipedia.org`)

`nslookup` is also capable of reverse DNS search when providing an IP address. The following command retrieves the name server (`dns.google`) corresponding to the IP address `8.8.8.8`:

```
nslookup 8.8.8.8
```

The preceding command yields the following output:

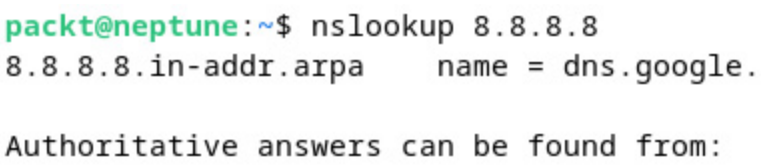

Figure 7.30 – Reverse DNS search with nslookup

For more information on the `nslookup` tool, you can refer to the `nslookup` system reference manual (`man nslookup`).

Alternatively, we can use the `dig` command-line utility. If you don't have the `dig` utility installed on your system, you can do so by installing the `dnsutils` package on Ubuntu/Debian or `bind-utils` on Fedora platforms. The related commands for installing the packages were shown previously with `nslookup`.

For example, the following command retrieves the name-server information for the `google.com` domain:

```
dig google.com
```

This is the result (see the highlighted `ANSWER SECTION`):

```
;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             39      IN      A       172.217.169.206
```

Figure 7.31 – DNS lookup with dig

To perform a reverse DNS lookup with `dig`, we must specify the `-x` option, followed by an IP address (for example, `8.8.4.4`), as follows:

```
dig -x 8.8.4.4
```

This command yields the following output (see the highlighted `ANSWER SECTION`):

```
;; QUESTION SECTION:
;4.4.8.8.in-addr.arpa.          IN      PTR

;; ANSWER SECTION:
4.4.8.8.in-addr.arpa.   25900   IN      PTR     dns.google.
```

Figure 7.32 – Reverse DNS lookup with dig

For more information about the `dig` command-line utility, please refer to the related system manual (`man dig`).

The DNS protocol operates at the application layer (*Layer 7*) in the OSI model. The standard DNS service's well-known port is `53`.

Chapter 8, *Linux Shell Scripting*, will cover the practical details of installing and configuring a DNS server in more detail. For more information on DNS, you can refer to *RFC 1035* (https://www.ietf.org/rfc/rfc1035.txt).

The DHCP and DNS network services are arguably the closest to the TCP/IP networking stack while playing a crucial role when computers or devices are attached to a network. After all, without proper IP addressing and name resolution, there's no network communication.

There's a lot more to distributed networking and related application servers than just strictly the pure network management stack performed by DNS and DHCP servers. In the following sections, we'll take a quick tour of some of the most relevant application servers running across distributed Linux systems.

Authentication servers

Standalone Linux systems typically use the default authentication mechanism, where user credentials are stored in the local filesystem (such as `/etc/passwd` and `/etc/shadow`). We explored the related user authentication internals in *Chapter 4*, *Managing Users and Groups*. However, as we extend the authentication boundary beyond the local machine – for example, accessing a file or email server – having the user credentials shared between the remote and localhosts would become a serious security issue.

Ideally, we should have a centralized authentication endpoint across the network that's handled by a secure authentication server. User credentials should be validated using robust encryption mechanisms before users can access remote system resources.

Let's consider the secure access to a network share on an arbitrary file server. Suppose the access requires **Active Directory** (**AD**) user authentication. Creating the related mount (share) locally on a user's client machine will prompt for user credentials. The authentication request is made by the file server (on behalf of the client) to an authentication server. If the authentication succeeds, the server share becomes available to the client. The following diagram represents a simple remote authentication flow between a client and a server, using a **Lightweight Directory Access Protocol** (**LDAP**) authentication endpoint:

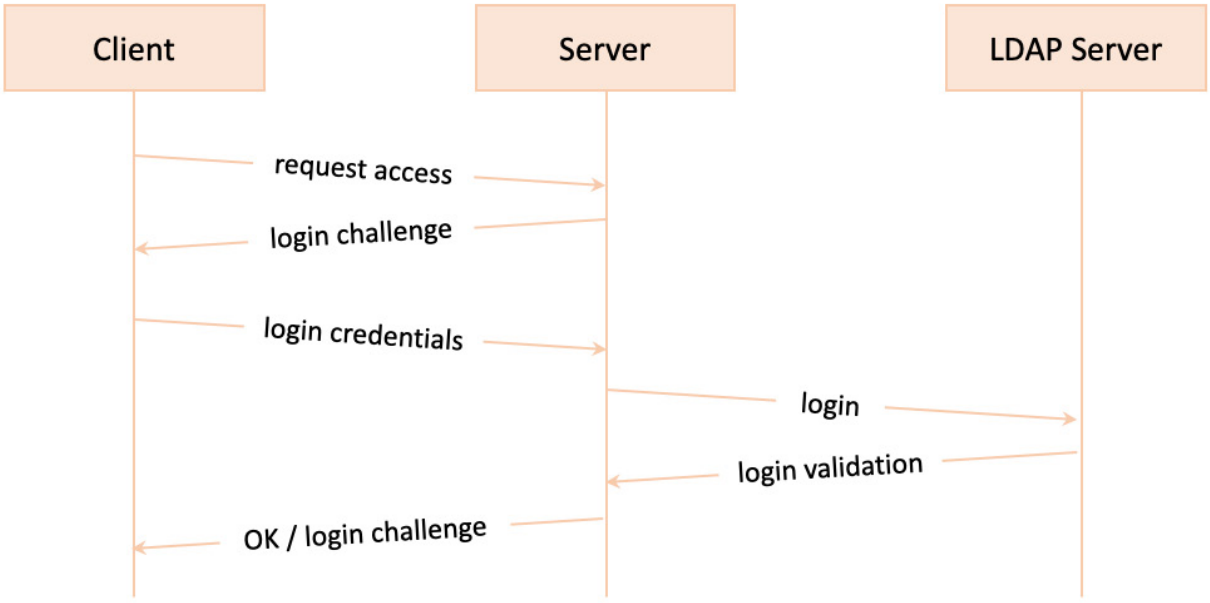

Figure 7.33 – Authentication workflow with LDAP

Here are some examples of standard secure authentication platforms (available for Linux):

- **Kerberos** (https://web.mit.edu/kerberos/)

- **LDAP** (https://www.redhat.com/en/topics/security/what-is-ldap-authentication)
- **Remote Authentication Dial-In User Service** (**RADIUS**) (https://freeradius.org/documentation/)
- **Diameter** (https://www.f5.com/glossary/diameter-protocol)
- **Terminal Access Controller Access-Control System** (**TACACS+**) (https://datatracker.ietf.org/doc/rfc8907/)

A Linux LDAP authentication server can be configured using OpenLDAP, which was covered in the first edition of this book.

In this section, we illustrated the authentication workflow with an example of using a file server. To remain on topic, we'll look at network file-sharing services next.

File sharing

In common networking terms, **file sharing** represents a client machine's ability to *mount* and access a remote filesystem belonging to a server, as if it were local. Applications running on the client machine would access the shared files directly on the server. For example, a text editor can load and modify a remote file, and then save it back to the same remote location, all in a seamless and transparent operation. The underlying remoting process – the appearance of a remote filesystem acting as local – is made possible by file-sharing services and protocols.

For every file-sharing network protocol, there is a corresponding client-server file-sharing platform. Although most network file servers (and clients) have cross-platform implementations, some operating system platforms are better suited for specific file-sharing protocols, as we'll see in the following subsections. Choosing between different file-server implementations and protocols is ultimately a matter of compatibility, security, and performance.

Here are some of the most common file-sharing protocols, with some brief descriptions for each:

- **Server Message Block** (**SMB**): The SMB protocol provides network discovery and file- and printer-sharing services. SMB also supports interprocess communication over a network. SMB is a relatively old protocol, developed by **International Business Machines Corporation** (**IBM**) in the 1980s. Eventually, Microsoft took over and made some considerable alterations to what became the current version through multiple revisions (SMB 1.0, 2.0, 2.1, 3.0, 3.0.2, and 3.1.1).
- **Common Internet File System** (**CIFS**): This protocol is a particular implementation of the SMB protocol. Due to the underlying protocol similarity, SMB clients can communicate with CIFS servers and vice versa. Though SMB and CIFS are idiomatically the same, their internal implementation of file locking, batch processing, and – ultimately – performance is quite different. Apart from legacy systems, CIFS is rarely used these days. SMB should always be preferred over CIFS, especially with the more recent revisions of SMB 2 or SMB 3.
- **Samba**: As with CIFS, Samba is another implementation of the SMB protocol. Samba provides file- and print-sharing services for Windows clients on a variety of server platforms. In other words, Windows clients can seamlessly access directories, files, and printers on a Linux Samba server, just as if they were communicating with a Windows server.

As of version 4, Samba natively supports Microsoft AD and Windows NT domains. Essentially, a Linux Samba server can act as a domain controller on a Windows AD network. Consequently, user credentials on the Windows domain can transparently be used on the Linux server without being recreated, and then manually kept in sync with the AD users.

- **Network File System** (**NFS**): This protocol was developed by Sun Microsystems and essentially operates on the same premise as SMB – accessing files over a network as if they were local. NFS is not compatible with CIFS or SMB, meaning that NFS clients cannot communicate directly with SMB servers or vice versa.
- **Apple Filing Protocol** (**AFP**): The AFP is a proprietary file-sharing protocol designed by Apple and exclusively operates in macOS network environments. We should note that besides AFP, macOS systems also support standard file-sharing protocols, such as SMB and NFS.

Most of the time, NFS is the file-sharing protocol of choice within Linux networks. For mixed networking environments – such as Windows, Linux, and macOS interoperability – Samba and SMB are best suited for file sharing.

Some file-sharing protocols (such as SMB) also support print sharing and are used by print servers. We'll take a closer look at print sharing next.

Printer servers

A **printer server** (or **print server**) connects a printer to client machines (computers or mobile devices) on a network using a printing protocol. Printing protocols are responsible for the following remote printing tasks over a network:

- Discovering printers or print servers
- Querying printer status
- Sending, receiving, queueing, or canceling print jobs
- Querying print job status

Common printing protocols include the following:

- **Line Printer Daemon** (**LPD**) protocol
- **Generic protocols**, such as SMB and TELNET
- **Wireless printing**, such as AirPrint by Apple
- **Internet printing protocols**, such as Google Cloud Print

Among the generic printing protocols, SMB (also a file-sharing protocol) was previously described in the *File sharing* section. The TELNET communication protocol was described in the *Remote access* section.

File- and printer-sharing services are mostly about *sharing* documents, digital or printed, between computers on a network. When it comes to *exchanging* documents, additional network services come into play, such as *file transfer* and *email* services. We'll look at file transfer next.

File transfer

FTP is a standard network protocol for transferring files between computers on a network. FTP operates in a client-server environment, where an FTP client initiates a remote connection to an FTP server, and files are transferred in either direction. FTP maintains a **control connection** and one or more **data connections** between the client and the server. The control connection is generally established on the FTP server's port `21`, and it's used for exchanging commands between the client and the server. Data connections are exclusively used for data transfer and are negotiated between the client and the server (through the control connection). Data connections usually involve ephemeral ports for inbound traffic, and they only stay open during the actual data transfer, closing immediately after the transfer completes.

FTP negotiates data connections in one of the following two modes:

- **Active mode**: The FTP client sends a **`PORT`** command to the FTP server, signaling that the client *actively* provides the inbound port number for data connections
- **Passive mode**: The FTP client sends a **`PASV`** command to the FTP server, indicating that the client *passively* awaits the server to supply the port number for inbound data connections

FTP is a relatively *messy* protocol when it comes to firewall configurations due to the dynamic nature of the data connections involved. The control connection port is usually well known (such as port `21` for insecure FTP) but data connections originate on a different port (usually `20`) on either side, while on the receiving end, the inbound sockets are opened within a preconfigured ephemeral range (`1024` to `65535`).

FTP is most often implemented securely through either of the following approaches:

- **FTP over SSL** (**FTPS**): SSL/TLS-encrypted FTP connection. The default FTPS control connection port is **`990`**.
- **SSH File Transfer Protocol** (**SFTP**): **FTP** over **SSH**. The default SFTP control connection port is **`22`**. For more information on the SSH protocol and client-server connectivity, refer to *SSH* in the *Remote access* section, later in this chapter.

Next, we'll look at mail servers and the underlying email exchange protocols.

Mail servers

A **mail server** (or **email server**) is responsible for email delivery over a network. A mail server can either exchange emails between clients (users) on the same network (domain) – within a company or organization – or deliver emails to other mail servers, possibly beyond the local network, such as the internet.

An email exchange usually involves the following actors:

- An **email client** application (such as Outlook or Gmail)
- One or more **mail servers** (Exchange or Gmail server)
- The **recipients** involved in the email exchange – a *sender* and one or more *receivers*
- An **email protocol** that controls the communication between the email client and the mail servers

The most used email protocols are **POP3**, **IMAP**, and **SMTP**. Let's take a closer look at each of these protocols.

POP3

POP version 3 (**POP3**) is a standard email protocol for receiving and downloading emails from a remote mail server to a local email client. With POP3, emails are available for reading offline. After being downloaded, emails are usually removed from the POP3 server, thus saving up space. Modern-day POP3 mail client-server implementations (Gmail, Outlook, and others) also have the option of keeping email copies on the server. Persisting emails on the POP3 server becomes very important when users access emails from multiple locations (client applications).

The default POP3 ports are outlined here:

- `110`: For insecure (non-encrypted) POP3 connections
- `995`: For secure POP3 using SSL/TLS encryption

POP3 is a relatively old email protocol that's not always suitable for modern-day email communications. When users access their emails from multiple devices, IMAP is a better choice. We'll look at the IMAP email protocol next.

IMAP

IMAP is a standard email protocol for accessing emails on a remote IMAP mail server. With IMAP, emails are always retained on the mail server, while a copy of the emails is available for IMAP clients. A user can access emails on multiple devices, each with their IMAP client application.

The default IMAP ports are outlined here:

- `143`: For insecure (non-encrypted) IMAP connections
- `993`: For secure IMAP using SSL/TLS encryption

Both POP3 and IMAP are standard protocols for receiving emails. To send emails, SMTP comes into play. We'll take a look at the SMTP email protocol next.

SMTP

SMTP is a standard email protocol for sending emails over a network or the internet.

The default SMTP ports are outlined here:

- **`25`**: For insecure (non-encrypted) SMTP connections
- **`465`** or **`587`**: For secure SMTP using SSL/TLS encryption

When using or implementing any of the standard email protocols described in this section, it is always recommended to use the corresponding secure implementation with the most up-to-date TLS encryption, if possible. POP3, IMAP, and SMTP also support user authentication, an added layer of security – this is also recommended in commercial or enterprise-grade environments.

To get an idea of how the SMTP protocol operates, let's go through some of the initial steps for initiating an SMTP handshake with Google's Gmail SMTP server.

We will start by connecting to the Gmail SMTP server, using a secure (TLS) connection via the **`openssl`** command, as follows:

```
openssl s_client -starttls smtp -connect smtp.gmail.com:587
```

Here, we invoked the **`openssl`** command, simulated a client (**`s_client`**), started a TLS SMTP connection (**`-starttls smtp`**), and connected to the remote Gmail SMTP server on port **`587 (-connect smtp.gmail.com:587`**).

The Gmail SMTP server responds with a relatively long TLS handshake block that ends with the following code:

```
---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: ECDSA
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 4578 bytes and written 429 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 256 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
250 SMTPUTF8
```

Figure 7.34 – Initial TLS handshake with a Gmail SMTP server

While still inside the `openssl` command's interactive prompt, we initiate the SMTP communication with a `HELO` command (spelled precisely as such). The `HELO` command *greets* the server. It is a specific SMTP command that starts the SMTP connection between a client and a server. There is also an `EHLO` variant, which is used for ESMTP service extensions. Google expects the following `HELO` greeting:

```
HELO hellogoogle
```

Another handshake follows, ending with `250 smtp.gmail.com at your service`, as illustrated here:

```
    Start Time: 1679236390
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
    Extended master secret: no
    Max Early Data: 0
---
read R BLOCK
250 smtp.gmail.com at your service
```

Figure 7.35 – The Gmail SMTP server is ready for communication

Next, the Gmail SMTP server requires authentication via the `AUTH LOGIN` SMTP command. We won't go into further details, but the key point to be made here is that the SMTP protocol follows a plaintext command sequence between the client and the server. It's very important to adopt a secure (encrypted) SMTP communication channel using TLS. The same applies to any of the other email protocols (POP3 and IMAP).

So far, we've covered several network services, some of them spanning multiple networks or even the internet. Network packets carry data and destination addresses within the payload, but there are also synchronization signals between the communication endpoints, mostly to discern between sending and receiving workflows. The synchronization of network packets is based on timestamps. Reliable network communications would not be possible without a highly accurate time-synchronization between network nodes. We'll look at network timekeepers next.

NTP servers

NTP is a standard networking protocol for clock synchronization between computers on a network. NTP attempts to synchronize the system clock on participating computers within a few milliseconds of **Coordinated Universal Time** (**UTC**) – the world's time reference.

The NTP protocol's implementation usually assumes a client-server model. An NTP server acts as a time source on the network by either broadcasting or sending updated **timestamp datagrams** to clients. An NTP server continually adjusts its system clock according to well-known accurate time servers worldwide, using specialized algorithms to mitigate network latency.

A relatively easy way to check the NTP synchronization status on our Linux platform of choice is by using the `ntpstat` utility. `ntpstat` may not be installed by default on our system. On Ubuntu, we can install it with the following command:

```
sudo apt install ntpstat
```

On Fedora, we can install `ntpstat` with the following command:

```
sudo dnf install ntpstat
```

`ntpstat` requires an NTP server to be running locally. To set up a local NTP server, you will need to do the following (all examples shown here are for Ubuntu 22.04.2 LTS):

- Install the `ntp` package with the following command:

  ```
  sudo apt install ntp
  ```

- Check for the `ntp` service's status:

  ```
  sudo systemctl status ntp
  ```

- Enable the `ntp` service:

  ```
  sudo systemctl enable ntp
  ```

- Modify the firewall settings:

  ```
  sudo ufw allow from any to any port 123 proto udp
  ```

- Install the **ntpdate** package:

```
sudo apt install ntpdate
```

- Restart the **ntp** service:

```
sudo systemctl restart ntp
```

Before installing the **ntp** utility, take into account that Ubuntu is using another tool instead of **ntpd** by default, named **timesyncd**. When installing **ntpd**, the default utility will be disabled.

To query the NTP synchronization status, we can run the following command:

```
ntpstat
```

This is the output:

```
packt@neptune:~$ ntpstat
synchronised to NTP server (31.209.85.242) at stratum 2
   time correct to within 29 ms
   polling server every 64 s
```

Figure 7.36 – Querying the NTP synchronization status with ntpstat

ntpstat provides the IP address of the NTP server the system is synchronized with (**31.209.85.242**), the synchronization margin (**29** milliseconds), and the time-update polling interval (**64** seconds). To find out more about the NTP server, we can **dig** its IP address with the following command:

```
dig -x 31.209.85.242
```

It looks like it's one of the **1wlcom** time servers (**ntp1.1wlcom.net**), as shown here:

```
packt@neptune:~$ dig -x 31.209.85.242

; <<>> DiG 9.18.1-1ubuntu1.3-Ubuntu <<>> -x 31.209.85.242
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 49498
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;242.85.209.31.in-addr.arpa.    IN      PTR

;; ANSWER SECTION:
242.85.209.31.in-addr.arpa. 2324 IN     PTR     ntp1.lwlcom.net.

;; Query time: 16 msec
;; SERVER: 127.0.0.53#53(127.0.0.53) (UDP)
;; WHEN: Sun Mar 19 15:04:57 UTC 2023
;; MSG SIZE  rcvd: 84
```

Figure 7.37 – Querying the NTP synchronization status with ntpstat

The NTP client-server communication uses UDP as the transport protocol on port `123`. *Chapter 9*, *Securing Linux*, has a section dedicated to installing and configuring an NTP server. For more information on NTP, you can refer to https://en.wikipedia.org/wiki/Network_Time_Protocol.

With that, our brief journey into networking servers and protocols has come to an end. Everyday Linux administration tasks often require some sort of remote access to a system. There are many ways to access and manage computers remotely. The next section describes some of the most common remote-access facilities and related network protocols.

Remote access

Most Linux network services provide a relatively limited **remote management** interface, with their management **command-line interface** (**CLI**) utilities predominantly operating locally on the same system where the service runs. Consequently, the related administrative tasks assume local Terminal access. Direct console access to the system is sometimes not possible. This is when remote-access servers come into play to enable a virtual Terminal login session with the remote machine.

Let's look at some of the most common remote-access services and applications.

SSH

SSH is perhaps the most popular secure login protocol for remote access. SSH uses strong encryption, combined with user authentication mechanisms, for secure communication between a client and a server machine. SSH servers are relatively easy to install and configure, and the *Setting up an SSH server* section in *Chapter 13*, *Configuring Linux Servers*, is dedicated to describing the related steps. The default network port for SSH is `22`.

SSH supports the following authentication types:

- Public-key authentication
- Password authentication
- Keyboard-interactive authentication

The following sections provide brief descriptions of these forms of SSH authentication.

Public-key authentication

Public-key authentication (or **SSH-key authentication**) is arguably the most common type of SSH authentication.

IMPORTANT NOTE

This section will use the terms public-key and SSH-key interchangeably, mostly to reflect the related SSH authentication nomenclature in the Linux community.

The SSH-key authentication mechanism uses a *certificate/key* pair – a **public** key (*certificate*) and a **private** key. An SSH certificate/key pair is usually created with the `ssh-keygen` tool, using standard encryption algorithms such as the **Rivest–Shamir–Adleman** algorithm (**RSA**) or the **Digital Signature Algorithm** (**DSA**).

The SSH public-key authentication supports either **user-based authentication** or **host-based authentication** models. The two models differ in the ownership of the certificate/key pairs involved. With client authentication, each user has a certificate/key pair for SSH access. On the other hand, host authentication involves a single certificate/key pair per system (host).

Both SSH-key authentication models are illustrated and explained in the following sections. The basic SSH handshake and authentication workflows are the same for both models:

- First, the SSH client generates a secure certificate/key pair and shares its public key with the SSH server. This is a one-time operation for enabling public-key authentication.
- When a client initiates the SSH handshake, the server asks for the client's public key and verifies it against its allowed public keys. If there's a match, the SSH handshake succeeds, the server shares its public key with the client, and the SSH session is established.
- Further client-server communication follows standard encryption/decryption workflows. The client encrypts the data with its private key, while the server decrypts the data with the client's public key. When responding to the client, the server encrypts the

data with its own private key, and the client decrypts the data with the server's public key.

SSH public-key authentication is also known as **passwordless authentication**, and it's frequently used in automation scripts where commands are executed over multiple remote SSH connections without prompting for a password.

Let's take a closer look at the user-based and host-based public-key authentication mechanisms:

- **User-based authentication**: This is the most common SSH public-key authentication mechanism. According to this model, every user connecting to a remote SSH server has its own SSH key. Multiple user accounts on the same host (or domain) would have different SSH keys, each with its own access to the remote SSH server, as suggested in the following figure:

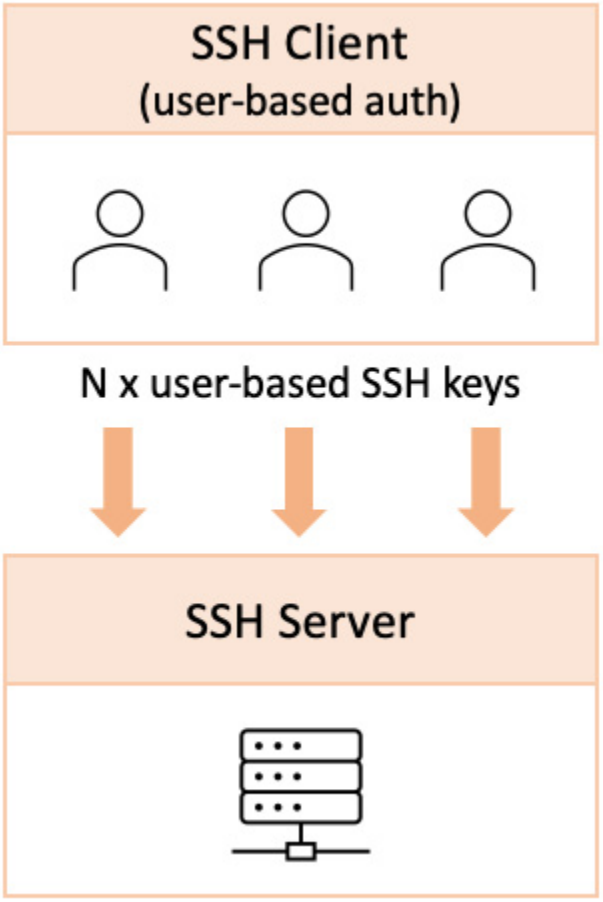

Figure 7.38 – User-based key authentication

- **Host-based authentication**: This is another form of SSH public-key authentication and involves a single SSH key per system (host) to connect to a remote SSH server, as illustrated in the following figure:

Figure 7.39 – Host-based key authentication

With host-based authentication, the underlying SSH key can only authenticate SSH sessions that originated from a single client host. Host-based authentication allows multiple users to connect from the same host to a remote SSH server. If a user attempts to use a host-based SSH key from a different machine than the one allowed by the SSH server, access will be denied.

Sometimes, a mix of the two public-key authentications is used – user- and host-based authentication –an approach that provides an increased level of security to SSH access.

When security is not critical, simpler SSH authentication mechanisms could be more suitable. Password authentication is one such mechanism.

Password authentication

Password authentication requires a simple set of credentials from the SSH client, such as a username and password. The SSH server validates the user credentials, either based on the local user accounts (in `/etc/passwd`) or select user accounts defined in the SSH server configuration (`/etc/ssh/sshd_config`). The SSH server configuration described in *Chapter 9*, *Securing Linux*, further elaborates on this subject.

Besides local authentication, SSH can also leverage remote authentication methods such as Kerberos, LDAP, RADIUS, and others. In such cases, the SSH server delegates the user authentication to a remote authentication server, as described in the *Authentication servers* section earlier in this chapter.

Password authentication requires either user interaction or some automated way to provide the required credentials. Another similar authentication mechanism is keyboard-interactive authentication, described next.

Keyboard-interactive authentication

Keyboard-interactive authentication is based on a dialogue of multiple challenge-response sequences between the SSH client (user) and the SSH server. This dialogue is a plaintext exchange of

questions and answers, where the server may prompt the user for any number of challenges. In some respect, password authentication is a **single-challenge interactive authentication** mechanism.

The *interactive* connotation of this authentication method could lead us to think that user interaction would be mandatory for the related implementation. Not really. Keyboard-interactive authentication could also serve implementations of authentication mechanisms based on custom protocols, where the underlying message exchange would be modeled as an authentication protocol.

Before moving on to other remote access protocols, we should call out the wide use of SSH due to its security, versatility, and performance. However, SSH connectivity may not always be possible or adequate in specific scenarios. In such cases, *TELNET* may come to the rescue. We'll take a look at it next.

TELNET

TELNET is an application-layer protocol for bidirectional network communication that uses a plaintext CLI with a remote host. Historically, TELNET was among the first remote-connection protocols, but it always lacked secure implementation. SSH eventually became the standard way to log in from one computer to another, yet TELNET has its advantages over SSH when it comes to troubleshooting various application-layer protocols, such as web- or email-server communication. You will learn more about how to use TELNET in *Chapter 9*, *Securing Linux*.

TELNET and SSH are command-line-driven remote-access interfaces. There are cases when a direct desktop connection is needed to a remote machine through a **graphical user interface** (**GUI**). We'll look at desktop sharing next.

VNC

Virtual Network Computing (**VNC**) is a desktop-sharing platform that allows users to access and control a remote computer's GUI. VNC is a cross-platform client-server application. A VNC server running on a Linux machine, for example, allows desktop access to multiple VNC clients running on Windows or macOS systems. The VNC network communication uses the **Remote Framebuffer** (**RFB**) protocol, defined by *RFC 6143*. Setting up a VNC server is relatively simple. VNC assumes the presence of a graphical desktop system. More details on this will be provided in *Chapter 13*, *Configuring Linux Servers*.

This concludes our section about network services and protocols. We tried to cover the most common concepts about general-purpose network servers and applications, mostly operating in a client-server or distributed fashion. With each network server, we described the related network protocols and some of the internal aspects involved. *Chapter 9*, *Securing Linux*, and *Chapter 13*, *Configuring Linux Servers*, will showcase practical implementations for some of these network servers.

In the next section, our focus will turn to network security internals.

Understanding network security

Network security represents the processes, actions, and policies to prevent, monitor, and protect unauthorized access to computer networks. Network security paradigms span a vast array of technologies, tools, and practices. Here are a few important ones:

- **Access control**: Selectively restricting access based on user authentication and authorization mechanisms. Examples of access control include users, groups, and permissions. Some of the related concepts were covered in *Chapter 4*, *Managing Users and Groups*.
- **Application security**: Securing and protecting server and end user applications (email, web, and mobile apps). Examples of application security include **Security-Enhanced Linux** (**SELinux**), strongly encrypted connections, antivirus, and anti-malware programs. We'll cover **SELinux** in *Chapter 10*, *Disaster Recovery, Diagnostics, and Troubleshooting*.
- **Endpoint security**: Securing and protecting servers and end user devices (smartphones, laptops, and desktop PCs) on the network. Examples of endpoint security include *firewalls* and various intrusion-detection mechanisms. We'll look at firewalls in *Chapter 10*, *Disaster Recovery, Diagnostics, and Troubleshooting*.
- **Network segmentation**: Partitioning computer networks into smaller segments or **virtual LANs** (**VLANs**). This is not to be confused with subnetting, which is a logical division of networks through addressing.
- **VPNs**: Accessing corporate networks using a secure encrypted tunnel from public networks or the internet. We'll look at VPNs in more detail in *Chapter 9*, *Securing Linux*, and *Chapter 13*, *Configuring Linux Servers*.

In everyday Linux administration, setting up a network security perimeter should always follow the paradigms enumerated previously, roughly in the order listed. Starting with access-control mechanisms and ending with VPNs, securing a network takes an *inside-out* approach, from local systems and networks to firewalls, VLANs, and VPNs.

Summary

This chapter provided a relatively condensed view of basic Linux networking principles. We learned about network communication layers and protocols, IP addressing schemes, TCP/IP configurations, well-known network application servers, and VPN. A good grasp of networking paradigms will give Linux administrators a more comprehensive view of the distributed systems and underlying communication between the application endpoints involved.

Some of the theoretical aspects covered in this chapter will be taken for a practical spin in *Chapter 13*, *Configuring Linux Servers*, where we'll focus on real-world implementations of network servers. *Chapter 10*, *Disaster Recovery, Diagnostics, and Troubleshooting*, will further explore network security internals and practical Linux firewalls. Everything we have learned so far will serve as a good foundation for the assimilation of these upcoming chapters.

The following chapter will introduce you to Linux shell scripting, where you will learn about the most common shell features and how to use decisions, loops, variables, arrays, and functions.

Questions

Here's a quick quiz to outline and test some of the essential concepts covered in this chapter:

1. How does the OSI model compare to the TCP/IP model?

   Hint: *Figure 7.2* could be of help.

2. Think of a couple of TCP/IP protocols and try to see where and how they operate in some of the network administration tasks or applications you are familiar with.
3. At what networking layer does the HTTP protocol operate? How about DNS?

   Hint: Both operate on the same layer.

4. What is the network class for IP address **`192.168.0.1`**?

   Hint: Refer to *Figure 7.5*.

5. What is the network prefix that corresponds to network mask **`255.255.0.0`**?

   Hint: Check out *Figure 7.5* again.

6. How do you configure a static IP address using the **`nmcli`** utility?

   Hint: Use **`connection modify`**.

7. How do you change the hostname of a Linux machine?

   Hint: Use the **`hostnamectl`** utility.

8. What is the difference between the POP3 and IMAP email protocols?
9. How does SSH host-based authentication differ from user-based SSH key authentication?
10. What is the difference between SSH and TELNET?

Further reading

For more information about what was covered in this chapter, please refer to the following Packt titles:

- *Linux Administration Best Practices*, by Scott Alan Miller
- *Linux for Networking Professionals*, by Rob VandenBrink

8

Linux Shell Scripting

Knowing how to use the basics of Linux shell programming and the **command-line interface** (**CLI**) is essential for a modern-day Linux professional.

In this chapter, you will learn how to use the Linux shell's programming capabilities to automate different tasks in Linux. You will learn about the structure of a basic Linux shell script and how it is organized and executed. We'll explore most of the commands already available to you from the previous chapters, especially the ones for working with files and directories and input and output redirection. Along the way, we'll introduce you to writing scripts, the structure and complexity of shell programming, and how to use specialized tools such as `sed` and `gawk`. We hope that by the end of this chapter, you'll be comfortable using scripts in your day-to-day workflow and be ready for future, more advanced explorations.

We're going to cover the following main topics:

- Introducing shell features
- The structure of a shell script
- Decisions, loops, variables, arrays, and functions
- Using `sed` and (`g`)`awk`

Technical requirements

This chapter requires a working installation of a standard Linux distribution, on either server, desktop, PC, or **virtual machine** (**VM**). Our examples and case studies will use mainly Ubuntu/Debian and RHEL/Fedora platforms, but the commands and examples that will be explored are equally suitable for other Linux distributions, such as openSUSE/SLE.

Exploring the Linux shell

Back in *Chapter 2*, *The Linux Shell and Filesystem*, we introduced you to the shell by exploring the available virtual consoles, command types, and the filesystem. This gave you a fair foundation for what we are about to explore in this chapter. By now, with everything we have been showing you in this book, you are already well versed in using the command line; you know some of the most common and useful commands available in Linux as we explored file operations, package, user, and

disk management, all the way up to network administration. All this knowledge will eventually be put to use in this chapter, where we will explore advanced shell features, shell variables, regular expressions, and how to take advantage of the powerful programming and automation features of the Bash shell.

In the next section, we will begin discovering the advanced features of the shell.

Bash shell features

The shell not only runs commands but also has many more features that make a system administrator's life more comfortable while at the command line. Some of these features include the use of **wildcards** and **metacharacters**, **brace expansion**, and **variables**. Apart from these, you will learn about the shell's `PATH` and aliases.

Before we proceed, let's dig into a little history about the standards the shell is based on. Back in the day, when UNIX emerged as an operating system, the need for a standard to oversee different variants appeared. Thus, the **Institute of Electrical and Electronics Engineers** (**IEEE**) created the **Portable Operating System Interface** (**POSIX**) as a family of different standards that were meant to assure compatibility between operating systems. Therefore, UNIX and Linux, as well as macOS (based on Darwin, the kernel of macOS derived from UNIX), AIX, HP-UX, and Oracle Solaris, are POSIX compliant. POSIX has different standards for the C language API, file format definitions, directory structures, environment variables definitions, locale specifications, character sets, and regular expressions.

With this short history lesson under our belts, let's proceed. In the next section, we will show you how to use shell wildcards and metacharacters.

Wildcards and metacharacters

In Linux, **wildcards** are used to match filenames. There are three main types of wildcards:

- **Asterisk** (`*`): This is used to match any string of none or more characters
- **Question mark** (`?`): This is used to match a single character
- **Brackets** (`[ ]`): This is used to match any of the characters inside brackets

Metacharacters are special characters that are used in Linux and any Unix-based system. These metacharacters are as follows:

	Metacharacter	Description		Metacharacter	Description
	>	Output redirection		<	Input redirection
	>>	Output redirection – append		<<	Input redirection - append
	*	File substitution wildcard		?	File substitution wildcard
	[ ]	File substitution wildcard		;	Command execution sequence
	|	Pipe for using multiple commands		( )	Group of commands in the execution sequence
	||	Conditional execution (OR)		&&	Conditional execution (AND)
	&	Run a command in the background		#	Use a command directly in the shell
	$	Variable value expansion		\	The escape character
	`cmd`	Command substitution		$(cmd)	Command substitution

Figure 8.1 – Metacharacter list

Let's look at two examples that use metacharacters for command substitution. We use the output of one command inside another command. This can be done in two ways, as shown in the following figure:

```
packt@neptune:~$ echo "Today is `date`"
Today is Mon Feb 27 02:09:18 PM UTC 2023
packt@neptune:~$ echo "Today is $(date)"
Today is Mon Feb 27 02:09:38 PM UTC 2023
packt@neptune:~$
```

Figure 8.2 – Example of command execution and substitution

The purpose of the preceding example is to show you how command substitution works in the shell, but perhaps we should explain further what the commands used do. We used two commands: **`echo`** and **`date`**. We used the output of the **`date`** command inside the output of the **`echo`** command. The **`echo`** command is one of the simplest commands in Linux as it prints the message between the quotes to the standard output. In our case, the message also consists of the **`date`** command, which outputs the current date of the system, in the format shown.

You can also combine two or more commands, and to do this in Linux, we use the pipe. The pipe sends the output of the first command that was used as the input for the second command, and so forth, depending on how many pipes you use.

In the following example, we're using the **`ls -l /etc`** command to do a long listing of the contents of the **`/etc`** directory; we will pipe this to the **`less`** command. Use it as shown in the following code:

```
ls -l /etc | less
```

The **`less`** command will show one display at a time, allowing you to see all the contents. You can use the arrow keys or the page up and page down keys to navigate through the output and see all the contents of the **`/etc`** directory.

The pipe and command substitution will be very useful, especially when you're working with complex commands or when scripting, as you will see later in this chapter when you learn how to

create and use scripts.

Now, let's execute some commands in a sequence. After that, we will use metacharacters to group commands and redirect the output to a file. All this is shown in the following screenshot:

```
packt@neptune:~$ who; pwd;
packt    pts/0        2023-02-27 13:49 (192.168.124.1)
/home/packt
packt@neptune:~$ (who; pwd) > users
packt@neptune:~$ cat users
packt    pts/0        2023-02-27 13:49 (192.168.124.1)
/home/packt
packt@neptune:~$
```

Figure 8.3 – Example of command sequence execution

As you can see in the preceding output, the two commands that were executed on the first line can easily be grouped using brackets, and their output can be redirected to a file.

We used three types of metacharacters – the *command execution sequence* (`;`), the brackets for *grouping commands* in the execution sequence, and the *output redirection* (`>`) to redirect the output to a file. The file did not exist initially as it was created only when the command was executed. The last command we used was the `cat` command, which *concatenates* the contents of the newly created file.

The first two commands that were used were `who`, which prints information about the currently logged-in users to the standard output, and `pwd`, which prints the present working directory as the location we are in inside the filesystem. In the following section, we will show you how to use brace expansion with the shell.

Brace expansion

Curly brackets can also be used to expand the arguments of a command. **Braces** are not just limited to filenames, unlike a wildcard. They work with any type of string. Inside these braces, you can use a single string, a sequence, or several strings separated by commas.

In this section, we will show you some examples of using this type of expansion. First, we will use **brace expansion** to delete two files from one directory. Second, we will show you how to create multiple new files using brace expansion. Let's say that we have two files named `report` and `new-report` inside our present working directory and we want to delete them both at once. We can use the following command:

```
rm {report,new-report}
```

To create multiple files (five of them, for example) that share parts of their name, as in `file1`, `file2`, … `filen`, we can use the following command:

```
touch file{1..5}
```

The following screenshot shows the output of both these commands:

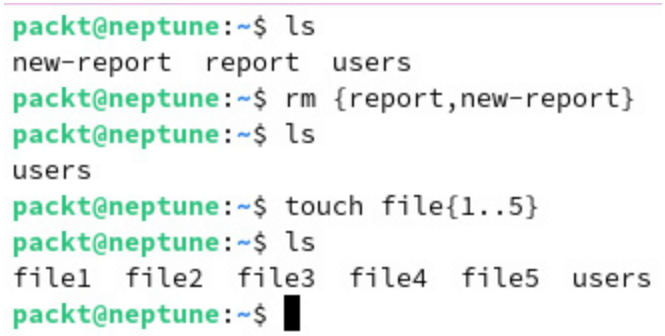

Figure 8.4 – Examples of using brace expansion

IMPORTANT NOTE

Brace expansion is a powerful tool that adds flexibility and power to any system administrator's workflow. They will prove very useful when learning how to script, for example.

Now that we've created those files, it should be really easy for you to figure out how to use brace expansion to delete multiple files at once. Type the following command into your console and see what happens:

```
rm file{1..5}
```

It will delete all five files we created previously. Use the `ls` command to see the contents of the present working directory.

In the next section, we will talk about shell command aliases, what they are, and how to use them.

The shell's aliases

The Linux shell supports **aliases**, which are a very convenient way to create short commands as aliases for longer ones. For example, in Ubuntu, there is a predefined alias called `ll` that is shorthand for `ls -alF`. You can define your own aliases too. You can make them temporary or permanent, similar to variables. In the following example, we changed the alias for the `ll` command:

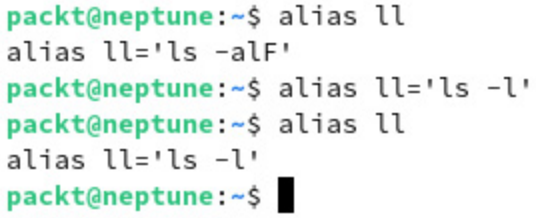

Figure 8.5 – Changing the alias of a command

This modification is only temporary, and it will revert to the default version after reboot or shell restart. If you want to make it permanent, you should edit the `~/.bashrc` file and add the aliases you created previously inside the file. To do this, open the file with your preferred text editor and add the lines you used in the Terminal to the file. Save the file and execute it. Also, a better practice would be to add those lines to a new file called `.bash_aliases`. You can view the default contents of `.bashrc` for more information on how to use aliases.

IMPORTANT NOTE

The `.bashrc` file is a hidden script file that consists of different Terminal session configurations. Also, the file can contain different functions that can help the user overcome repetitive tasks. It is automatically executed when the user logs in, but it can also be manually executed by using the `source .bashrc` command.

In the next section, we will show you what shell variables are and how to use them.

Bash shell variables

The Bash shell uses different types of variables, in the same way you would use them in any programming language. The Bash shell has some built-in variables and indirect variables and offers the possibility to define your own variables as well.

Linux has two major types of shell variables: **global** and **local variables**. They are generally identical for every Linux distribution out there, with some exceptions. You will need to consult your distribution's documentation for any specific modifications to the environment variables.

We will walk you through the most widely used variables in Linux, starting with the built-in ones.

Built-in shell variables

Here is a short list of some of the standard built-in variables:

- **`HOME`**: The user's home directory (for example, **`/home/packt`**)
- **`LOGNAME`**: The user's login name (for example, **`packt`**)
- **`PWD`**: The shell's current working directory
- **`OLDPWD`**: The shell's previous working directory
- **`PATH`**: The shell's search path (list of directories separated by colons)
- **`SHELL`**: The path to the shell
- **`USER`**: The user's login name
- **`TERM`**: The type of the Terminal

To call a variable while in the shell, all you have to do is place a dollar sign, `$`, in front of the variable's name. Here is a short example that shows how to use the variables that we just listed:

```
packt@neptune:~$ echo $HOME
/home/packt
packt@neptune:~$ echo $LOGNAME
packt
packt@neptune:~$ echo $PWD
/home/packt
packt@neptune:~$ echo $OLDPWD

packt@neptune:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/loc
al/games:/snap/bin
packt@neptune:~$ echo $SHELL
/bin/bash
packt@neptune:~$ echo $USER
packt
packt@neptune:~$ echo $TERM
xterm-256color
packt@neptune:~$
```

Figure 8.6 – Variable calling from the shell

You can also assign your own shell variables, as in the following example. Here, we're assigning the `sysadmin` string to a new variable called `MYVAR` and then printing it to standard output:

```
MYVAR=sysadmin; echo $MYVAR
```

The variables listed at the beginning of this section are just a part of all the variables available by default inside the shell. To see all the shell variables, use the `printenv` command. If the list is too long, you can redirect it to a file. In the following example, your variables list is inside the `shell_variables` file, and you can see it by concatenating or by editing inside a text editor such as Vim:

```
printenv > ~/shell_variables
```

Here, we use the tilde symbol (~) to specify the logged-in user's home directory. The shell's variables are only available inside the shell. If you want some variables to be known to other programs that are run by the shell, you must export them by using the `export` command. Once a variable is exported from the shell, it is known as an **environment variable**.

The shell's search path

The `PATH` variable is an essential one in Linux. It helps the shell know where all the programs are located. When you enter a command into your Bash shell, it first has to search for that command through the Linux filesystem. Some directories are already listed inside the `PATH` variable, but you can also add new ones. Your addition can be temporary or permanent, depending on how you do it. To

make a directory's path available temporarily, simply add it to the `PATH` variable. In the following example, we're adding the `/home/packt` directory to `PATH`:

```
packt@neptune:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/loc
al/games:/snap/bin
packt@neptune:~$ PATH=$PATH:/home/packt
packt@neptune:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/loc
al/games:/snap/bin:/home/packt
packt@neptune:~$
```

Figure 8.7 – Adding a new location to PATH

To make any changes permanent, we must modify the `PATH` variable inside a file called `~/.bash_profile` or `~/.bashrc`.

IMPORTANT NOTE

Some distributions, such as openSUSE, add an extra `bin` directory inside the user's home directory. This is a place where you can put files that you want the shell to execute – for example, script files.

The shell's `$PATH` variable is important, especially when using scripts, as you will preferably have to create scripts inside a directory that is known by the shell. In the next section, we will show you how to create your first Bash scripts.

Basics of shell scripting

We have already covered important aspects of the Linux command line, shell variables, wildcards, and metacharacters. Now, we will start exploring what scripts are, how to create them, and how to use them in a Linux CLI. We will not use the graphical user interface, only the CLI, which we primarily used in our previous chapters. Let's start with some basic, but important, concepts about shell scripting.

First, let's learn what a **script** is. If we were to check the meaning of the term in a dictionary, the answer would be that a script is a series of instructions that are executed by a computer, mainly to automate a specific task. Instructions can easily be assimilated as commands too. Thus, a series of commands executed together by the shell can be considered a script. This is a very basic script, but it is a script. Let's look at how we can create a script file.

Creating a shell script file