Network Programmability and Automation Fundamentals

Rough Cuts

KHALED ABUELENAIN
JEFF DOYLE
ANTON KARNELIUK
VINIT JAIN

ciscopress.com

Network Programmability and Automation Fundamentals

Khaled Abuelenain, CCIE No. 27401
Jeff Doyle, CCIE No. 1919
Anton Karneliuk, CCIE No. 49412
Vinit Jain, CCIE No. 22854

Network Programmability and Automation Fundamentals

Cisco Press logo is a trademark of Cisco Systems, Inc.

Published by:
Cisco Press

Library of Congress Control Number: 2020922839

ISBN-13: 978-1-58714-514-8

ISBN-10: 1-58714-514-6

Warning and Disclaimer

This book is designed to provide information about network programmability and automation. Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied.

The information is provided on an “as is” basis. The authors, Cisco Press, and Cisco Systems, Inc. shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the discs or programs that may accompany it.

The opinions expressed in this book belong to the author and are not necessarily those of Cisco Systems, Inc.

Feedback Information

At Cisco Press, our goal is to create in-depth technical books of the highest quality and value. Each book is crafted with care and precision, undergoing rigorous development that involves the unique expertise of members from the professional technical community.

Readers' feedback is a natural continuation of this process. If you have any comments regarding how we could improve the quality of this book or otherwise alter it to better suit your needs, you can contact us through email at feedback@ciscopress.com. Please make sure to include the book title and ISBN in your message.

We greatly appreciate your assistance.

Editor-in-Chief
Mark Taub

Director, ITP Product Management
Brett Bartow

Executive Editor
Brett Bartow

Managing Editor
Sandra Schroeder

Development Editor
Ellie C. Bru

Project Editor
Mandie Frank

Copy Editor
Kitty Wilson

Technical Editors
Jeff Tantsura, Viktor Osipchuk

Editorial Assistant
Cindy Teeters

Designer
Chuti Prasertsith

Composition
codeMantra

Indexer

Proofreader

Trademark Acknowledgments

About the Authors

Khaled Abuelenain, CCIE No. 27401 (R&S, SP), is currently the Consulting Director at Acuative, a Cisco Managed Services Master Partner. Khaled has spent the past 18 years designing, implementing, operating, and automating networks and clouds. He specializes in service provider technologies, SD-WAN, data center technologies, programmability, automation, and cloud architectures. Khaled is especially interested in Linux and OpenStack.

Khaled is a contributing author of the best-selling Cisco Press book *Routing TCP/IP, Volume II*, 2nd edition, by Jeff Doyle. He also blogs frequently on network programmability and automation on blogs.cisco.com. Khaled is also a member of the DevNet500 group, being one of the first 500 individuals in the world to become DevNet certified.

Khaled lives in Riyadh, Saudi Arabia, and when not working or writing, he likes to run marathons and skydive. He can be reached at kabuelenain@gmail.com, on Twitter at @kabuelenain or on LinkedIn at linkedin.com/in/kabuelenain.

Jeff Doyle, CCIE No. 1919, is a Member of Technical Staff at Apstra. Specializing in IP routing protocols, complex BGP policy, SDN/NFV, data center fabrics, IBN, EVPN, MPLS, and IPv6, Jeff has designed or assisted in the design of large-scale IP and IPv6 service provider networks in 26 countries over 6 continents.

Jeff is the author of *CCIE Professional Development: Routing TCP/IP, Volumes I and II* and *OSPF and IS-IS: Choosing an IGP for Large-Scale Networks*; a co-author of *Software-Defined Networking: Anatomy of OpenFlow*; and an editor and contributing author of *Juniper Networks Routers: The Complete Reference*. Jeff is currently writing *CCIE Professional Development: Switching TCP/IP*. He also writes for Forbes and blogs for both *Network World* and *Network Computing*. Jeff is one of the founders of the Rocky Mountain IPv6 Task Force, is an IPv6 Forum Fellow, and serves on the executive board of the Colorado chapter of the Internet Society (ISOC).

Anton Karneliuk, CCIE No. 49412 (R&S, SP), is a Network Engineer and Manager at THG Hosting, responsible for the development, operation, and automation of networks in numerous data centers across the globe and the international backbone. Prior to joining THG, Anton was a team lead in Vodafone Group Network Engineering and Delivery, focusing on introduction of SDN and NFV projects in Germany. Anton has 15 years of extensive experience in design, rollout, operation, and optimization of large-scale service provides and converged networks, focusing on IP/MPLS, BGP, network security, and data center Clos fabrics built using EVPN/VXLAN. He also has several years of full-stack software development experience for network management and automation.

Anton holds a B.S. in telecommunications and an M.S. in information security from Belarusian State University of Informatics and Radio Electronics. You can find him actively blogging about network automation and running online training at Karneliuk.com. Anton lives with his wife in London.

Vinit Jain, CCIE No. 22854 (R&S, SP, Security & DC), is a Network Development Engineer at Amazon, managing the Amazon network backbone operations team. Previously, he worked as a technical leader with the Cisco Technical Assistance Center (TAC), providing escalation support in routing and data center technologies. Vinit is a speaker at various networking forums, including Cisco Live! events. He has co-authored several Cisco Press titles, such as *Troubleshooting BGP*, and *Troubleshooting Cisco Nexus Switches and NX-OS*, *LISP Network Deployment and Troubleshooting,* and has authored and co-authored several video courses, including *BGP Troubleshooting*, the *CCNP DCCOR Complete* video course, and the *CCNP ENCOR Complete* video course. In addition to his CCIEs, Vinit holds multiple certifications related to programming and databases. Vinit graduated from Delhi University in mathematics and earned a master's in information technology from Kuvempu University in India. Vinit can be found on Twitter as @VinuGenie.

About the Technical Reviewers

Jeff Tantsura, CCIE No. 11416 (R&S), has been in the networking space for over 25 years and has authored and contributed to many RFCs and patents and worked in both service provider and vendor environments.

He is co-chair of IETF Routing Working Group, chartered to work on new network architectures and technologies, including protocol-independent YANG models and next-generation routing protocols. He is also the co-chair of the RIFT (Routing in Fat Trees) Working Group, chartered to work on a new routing protocol that specifically addresses fat tree topologies typically seen in the data center environment.

Jeff serves on the Internet Architecture Board (IAB). His focus has been on 5G transport and integration with RAN, IoT, MEC, low-latency networking, and data modeling. He's also a board member of San Francisco Bay Area ISOC chapter.

Jeff is Head of Networking Strategy at Apstra, a leader in intent networking, where he defines networking strategy and technologies.

Jeff also holds the certification Ericsson Certified Expert IP Networking.

Jeff lives in Palo Alto, California, with his wife and youngest child.

Viktor Osipchuk, CCIE No. 38256 (R&S, SP), is a Senior Network Engineer at Google, focusing on automation and improving one of the largest production networks in the world. Before joining Google, Viktor spent time at DigitalOcean and Equinix, helping to architect and run their worldwide infrastructures. Viktor spent many years at Cisco, supporting customers and focusing on automation, telemetry, data models, and APIs for large-scale web and service provider deployments. Viktor has around 15 years of diverse network experience, an M.S. in telecommunications, and associated industry certifications.

Dedications

Khaled Abuelenain: To my mother, the dearest person to my heart, who invested all the years of her life so I can be who I am today. I owe you more than any words can express. To my father, my role model, who always led by example and showed me the real meaning of work ethic. Nothing I do or say will ever be enough to thank you both.

And to the love of my life, my soulmate, and my better half, Mai, for letting me work and write while you take care of, literally, everything else. This book would not have happened if not for your phenomenal support, patience and love. I will forever be grateful for the blessing of having you in my life.

Jeff Doyle: I would like to dedicate this book to my large and growing herd of grandchildren: Claire, Samuel, Caroline, Elsie, and Amelia. While they are far too young to comprehend or care about the contents of this book, perhaps someday they will look at it and appreciate that Grampa is more than a nice old man and itinerant babysitter.

Anton Karneliuk: I dedicate this book to my family, which has tremendously supported me during the writing process. First of all, many thanks to my amazing wife, Julia, who took on the huge burden of sorting out many things for our lives, allowing me to concentrate on the book. You acted as a navigation star during this journey, and you are my beauty. I'd also like to thank my parents and brother for me helping me form the habit of working hard and completing the tasks I've committed to, no matter how badly I want to drop them.

Vinit Jain: I would like to dedicate this book to the woman who has been a great influence and inspiration in my life: Sonal Sethia (*Sonpari*). You are one of the most brilliant, talented, courageous, and humble people I have ever known. You have always inspired me to push myself beyond what I thought I was capable of. You have been there for me during difficult times and believed in me when even I did not. You are my rock. This is a small token of my appreciation, gratitude, and love for you. I am really glad to have found my best friend in you and know that I will always be there for you.

Acknowledgments

Khaled: First and foremost, I would like to thank Jeff Doyle, my co-author, mentor, and friend, for getting me started with writing, and for his continuous assistance and guidance. Jeff has played a fundamental role in my professional life as well as in the lives of many other network engineers; he probably doesn't realize the magnitude of this role! Despite all that he has done to this industry and the network engineering community, Jeff remains one of the most humble and amiable human beings I have ever come across. Thank you, Jeff, I owe you a lot!

I am grateful to Anton and Vinit for agreeing to work with me on this project. It has been challenging at times, but it has been seriously fun most of the time.

I would also like to thank Jeff Tantsura and Viktor Osipchuk for their thorough technical reviews and feedback. I bothered Viktor very frequently with discussions and questions over email, and never once did he fail to reply and add a ton of value !

I especially want to thank Brett Bartow and Eleanor Bru for their immense support and phenomenal patience. And I'm grateful to Mandie Frank, Kitty Wilson and everyone else at Cisco Press who worked hard to get this book out to the light. Such an amazing team.

Anton: Special thanks to Schalk Van Der Merwe, CTO, and Andrew Mutty, CIO, at The Hut Group for believing in me and giving me freedom and responsibility to implement my automation ideas in a high-scale data center environment. Thanks to all my brothers-in-arms from The Hut Group hosting networks for constantly sharing with me ideas about what use cases to focus on for automation. I want to thank my previous manager in Vodafone Group, Tamas Almasi, who supported me during my initial steps in network automation and helped me create an appropriate mindset during numerous testbeds and proofs of concept. Last but not least, I'm very grateful to Khaled Abuelenain for his invitation to co-author this book and the whole author and technical reviewer team; it was a pleasure to work with you.

Vinit: A special thanks to Khaled for asking me to co-author this book and for being amazingly patient and supportive of me as I faced challenges during this project. I would like to thank Jeff Doyle and Anton Karneliuk for their amazing collaboration on this project. I learned a lot from all of you guys and look forward to working with all of you in the future.

I would also like to thank our technical reviewers, Jeff Tantsura and Viktor Osipchuk, and our editor, Eleanor Bru, for your in-depth verification of the content and insightful input to make this project a successful one.

This project wouldn't have been possible without the support of Brett Bartow and other members of the editorial team.

Contents at a Glance

Introduction

Part I: Introduction

Chapter 1. The Network Programmability

Part II: Linux

Chapter 2. Linux Fundamentals

Chapter 3. Linux Storage, Security, and Networks

Chapter 4. Linux Scripting

Part III: Python

Chapter 5. Python Fundamentals

Chapter 6. Python Applications

Part IV: Transport

Chapter 7. HTTP and REST

Chapter 8. Advanced HTTP

Chapter 9. SSH

Part V: Encoding

Chapter 10. XML and XSD

Chapter 11. JSON and JSD

Chapter 12. YAML

Part VI: Modeling

Chapter 13. YANG

Part VII: Protocols

Chapter 14. NETCONF and RESTCONF

Chapter 15. gRPC, Protobuf, and gNMI

Chapter 16. Service Provider Programmability

Part VIII: Programmability Applications

Chapter 17. Programming Cisco Platforms

Chapter 18. Programming Non-Cisco Devices

Chapter 19. Ansible

Part IX: Looking Ahead

Chapter 20. Looking Ahead

Contents

Introduction

Goals and Methods of This Book

Who This Book Is For

How This book Is Organized

How This Book Is Structured

Part I: Introduction

Chapter 1. The Network Programmability

First, a Few Definitions

Your Network Programmability and Automation Toolbox

Software and Network Engineers: The New Era

Part II: Linux

Chapter 2. Linux Fundamentals

The Story of Linux

The Linux Boot Process

A Linux Command Shell Primer

Finding Help in Linux

Files and Directories in Linux

Input and Output Redirection

Archiving Utilities

Linux System Maintenance

Installing and Maintaining Software on Linux

Summary

Chapter 3. Linux Storage, Security, and Networks

Linux Storage

Linux Security

Linux Networking

Summary

Chapter 4. Linux Scripting

Regular Expressions and the grep Utility

The awk Programming Language

The sed Utility

General Structure of Shell Scripts

Output and Input

Variables

Conditional Statements

Loops

Functions

Expect

Summary

Part III: Python

Chapter 5. Python Fundamentals

Computer Science Concepts

Python Fundamentals

Summary

References

Chapter 6. Python Applications

Organizing the Development Environment

Python Modules

Python Applications

Summary

Part IV: Transport

Chapter 7. HTTP and REST

HTTP Overview

The REST Framework

The HTTP Connection

HTTP Transactions

HTTP Messages

Resource Identification

Postman

HTTP and Bash

HTTP and Python

Summary

Chapter 8. Advanced HTTP

HTTP/1.1 Authentication

Transport Layer Security (TLS) and HTTPS

HTTP/2

Summary

Chapter 9. SSH

SSH Overview

Setting Up SSH

Enabling SSH on Cisco Devices

Secure File Transfer

Summary

References

Part V: Encoding

Chapter 10. XML and XSD

XML Overview, History, and Usage

XML Syntax and Components

Making XML Valid

Navigating XML Documents

XML Stylesheet Language Transformations (XSLT)

Processing XML Files with Python

Summary

Chapter 11. JSON and JSD

JavaScript Object Notation (JSON)

JSON Schema Definition (JSD)

Summary

Chapter 12. YAML

YAML Structure

Handling YAML Data Using Python

Summary

Part VI: Modeling

Chapter 13. YANG

A Data Modeling Primer

YANG Data Models

Types of YANG Modules

YANG Tools

Summary

Part VII: Protocols

Chapter 14. NETCONF and RESTCONF

NETCONF

RESTCONF

Summary

Chapter 15. gRPC, Protobuf, and gNMI

Requirements for Efficient Transport

History and Principles of gRPC

gRPC as a Transport

The Protocol Buffers Data Format

Working with gRPC and Protobuf in Python

The gNMI Specification

The Anatomy of gNMI

Managing Network Elements with gNMI/gRPC

Summary

Chapter 16. Service Provider Programmability

The SDN Framework for Service Providers

Segment Routing (SR)

BGP Link State (BGP-LS)

Path Computation Element Protocol (PCEP)

Summary

Part VIII: Programmability Applications

Chapter 17. Programming Cisco Platforms

API Classification

Network Platforms

Meraki

DNA Center

Collaboration Platforms

Summary

Chapter 18. Programming Non-Cisco Devices

General Approaches to Programming Networks

Implementation Examples

Summary

Chapter 19. Ansible

Ansible Basics

Extending Ansible Capabilities

Jinja2 Templates

Using Ansible for Cisco IOS XE

Using Ansible for Cisco IOS XR

Using Ansible for Cisco NX-OS

Using Ansible in Conjunction with NETCONF

Summary

Part IX: Looking Ahead

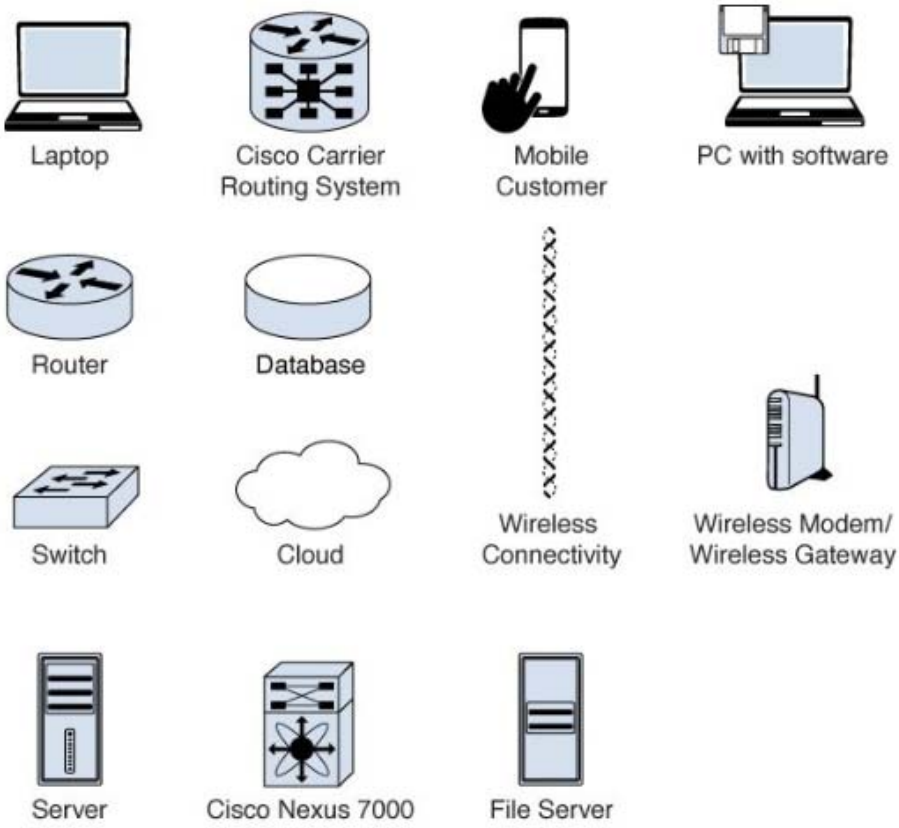
Chapter 20. Looking Ahead

Some Rules of Thumb

What Do You Study Next?

What Does All This Mean for Your Career?

Icons Used in This Book



Command Syntax Conventions

The conventions used to present command syntax in this book are the same conventions used in Cisco's Command Reference. The Command Reference describes these conventions as follows:

• **Boldface** indicates commands and keywords that are entered literally as shown. In actual configuration examples and output (not general command syntax), boldface indicates commands that are manually input by the user (such as a **show** command).

• *Italics* indicate arguments for which you supply actual values.

• Vertical bars (|) separate alternative, mutually exclusive elements.

• Square brackets [ ] indicate optional elements.

• Braces { } indicate a required choice.

• Braces within brackets [{ }] indicate a required choice within an optional element.

Note

This book covers multiple operating systems, and in each example, icons and router names indicate the OS that is being used. IOS and IOS XE use router names like R1 and R2 and are referenced by the IOS router icon. IOS XR routers use router names like XR1 and XR2 are referenced by the IOS XR router icon.

Introduction

For more than three decades, network management has been entirely based on the command-line interface (CLI) and legacy protocols such as SNMP. These protocols and methods are severely limited. The CLI, for example, is vendor specific, lacks a unified data hierarchy (sometimes even for platforms from the same vendor), and was designed primarily as a human interface. SNMP suffers major scaling problems, is not fit for writing configuration to devices, and overall, is very complex to implement and customize.

In essence, automation aims at offloading as much work from humans as possible and delegating that work to machines. But with the aforementioned legacy interfaces and protocols, machine-to-machine communication is neither effective nor efficient; and at times, it is impossible.

Moreover, device configuration and operational data have traditionally lacked a proper hierarchy and failed to follow a data model. In addition, network management workflows have always been far from mature, compared to software development workflows in terms of versioning, collaboration, testing, and automated deployments.

Enter network programmability. Programmability revolves around programmable interfaces, commonly referred to as application programming interfaces (APIs). APIs are interfaces that are designed primarily to be used for machine-to-machine communication. A Python program accessing a network router to retrieve or push configuration, without human intervention, is an example of a machine-to-machine interaction. Contrast this with the CLI, where a human needs to manually enter commands on a device and then visually inspect the output.

Network equipment vendors (for both physical and virtual equipment) are placing ever-increasing emphasis on the importance of managing their equipment using programmable interfaces, and Cisco is at the forefront of this new world. This new approach to managing a network provides several benefits over legacy methods, including the following:

• Normalizing the interface for interaction with network platforms by abstracting communication with these platforms and breaking the dependency of this communication on specific network OS scripting languages (for example, NX-OS, IOS XR and Junos OS)

• Providing new methods of interacting with network platforms and, in the process, enabling and aligning with new technologies and architectures, such as SDN, NFV, and cloud

• Harnessing the power of programming to automate manual tasks and perform repetitive tasks efficiently

• Enabling rapid infrastructure and service deployment by using workflows for service provisioning

• Increasing the reliability of the network configuration process by leveraging error checking, validation, and rollback and minimizing human involvement in the configuration process

• Using common software development tools and techniques for network configuration management, such as software development methodologies, versioning, staging, collaboration, testing, and continuous integration/continuous delivery

This book covers all the major programmable interfaces used in the market today for network management. The book discusses the protocols, tools, techniques, and technologies on which network programmability is based. Programming, operating systems, and APIs are not new technologies. However, programmable interfaces on network platforms, and using these programmable interfaces to fully operate and maintain a network, along with the culture accompanying these new methods and protocols, may be (relatively) new. This book explains, in detail, all the major components of this new ecosystem.

Goals and Methods of This Book

This is a “fundamentals” book aimed at transitioning network engineers from a legacy network-based mindset to a software-based (and associated technologies) mindset. A book covering fundamentals generally struggles to cover as many subjects as possible with just enough detail. The fine balance between breadth and depth is challenging, but this book handles this challenge very well.

This book introduces the emerging network programmability and automation ecosystem based on programmable interfaces. It covers each protocol individually, in some significant detail, using the relevant RFCs as guiding documents. Protocol workflows, messages, and other protocol nuances tend to be dry, and at times boring, so to keep things interesting, practical examples are given wherever possible and relevant. You, the reader, can follow and implement these examples on your machine, which can be as simple as a Linux VM with Python Version 3.x installed, and free tools to work with APIs, such as Postman and cURL. This book makes heavy use of the Cisco DevNet sandboxes, so in the majority of cases, you do not need a home lab to test and experiment with physical equipment.

A whole section of the book is dedicated to putting the knowledge and skills learned throughout the book to good use. One chapter covers programming Cisco platforms and another covers programming non-Cisco platforms. A third chapter in that same section is dedicated exclusively to Ansible. This book provides an abundance of hands-on practice.

The last chapter provides a way forward, discussing tools and technologies that you might want to explore after you are done with this book.

Who This Book Is For

This book is meant for the following individuals and roles, among others:

• Network architects and engineers who want to integrate programmability into their network designs

• NOC engineers monitoring and operating programmable networks or those who rely on network management systems that utilize programmability protocols

• Network engineers designing, implementing, and deploying new network services

• Software engineers or programmers developing applications for network management systems

• Network and software engineers working with networks or systems involving SDN, NFV, or cloud technologies

• Network engineers pursuing their Cisco DevNet certifications

Whether you are an expert network engineer with no prior programming experience or knowledge, or a software engineer looking to utilize your expertise in the network automation domain, after reading this book, you will fully understand the most commonly used protocols, tools, technologies, and techniques related to the subject, and you will be capable of effectively using the newly learned material to design, implement, and operate full-fledged programmable networks and the associated network automation systems.

How This book Is Organized

This book covers the information you need to transition from having a focus on networking technology to focusing on software and network programmability. This book covers six main focus areas:

• Operating systems: Linux

• Software development: Python

• Transport: HTTP, REST and SSH

• Encoding: XML, JSON, and YAML

• Modeling: YANG

• Protocols: NETCONF, RESTCONF, gRPC, and service provider programmability

• Practical programmability: Cisco platforms, non-Cisco platforms, and Ansible

Each chapter in this book either explicitly covers one of these focus areas or prepares you for one of them. Special consideration has been given to the ordering of topics to minimize forward referencing. Following an introduction to the programmability landscape, Linux is covered first because to get anything done in network programmability, you will almost always find yourself working with Linux. The book next covers Python because the vast majority of the rest of the book includes coverage of Python in the context of working with various protocols. The following chapters present an organic flow of topics: transport, encoding, modeling, and the protocols that build on all the previous sections. For example, understanding NETCONF requires you to understand SSH, XML, and YANG, and understanding RESTCONF requires that you understand HTTP, XML/JSON, and YANG. Both NETCONF and RESTCONF require knowledge of Python, most likely running on a Linux machine.

How This Book Is Structured

The book is organized into nine parts, described in the following sections.

PART I, "Introduction"

Chapter 1, "The Network Programmability and Automation Ecosystem": This chapter introduces the concepts and defines the terms that are necessary to understand the protocols and technologies covered in the following chapters. It also introduces the network programmability stack and explores the different components of the stack that constitute a typical network programmability and automation toolbox.

PART II, "Linux"

Chapter 2, "Linux Fundamentals": Linux is the predominant operating system used for running software for network programmability and automation. Linux is also the underlying operating system for the vast majority of network device software, such as IOS XR, NX-OS, and Cumulus Linux. Therefore, to be able to effectively work with programmable devices, it is of paramount importance to master the fundamentals of Linux. This chapter introduces Linux, including its architecture and boot process, and covers the basics of working with Linux through the Bash shell, such as working with files and directories, redirecting input and output, performing system maintenance, and installing software.

Chapter 3, "Linux Storage, Security and Networks": This chapter builds on Chapter 2 and covers more advanced Linux topics. It starts with storage on Linux systems and the Linux Logical Volume Manager. It then covers Linux user, group, file, and system security. Finally, it explains three different methods to manage networking in Linux; the **ip** utility, the NetworkManager service, and network configuration files.

Chapter 4, "Linux Scripting": This chapter builds on Chapters 2 and 3 and covers Linux scripting using the Bash shell. The chapter introduces the **grep**, **awk**, and **sed** utilities and covers the syntax and semantics of Bash scripting. The chapter covers comments, input and output, variables and arrays, expansion, operations and comparisons, how to execute system commands from a Bash script, conditional statements, loops, and functions. It also touches on the Expect programming language.

PART III, "Python"

Chapter 5, "Python Fundamentals": This chapter assumes no prior knowledge of programming and starts with an introduction to programming, covering some very important software and computer science concepts, including algorithms and object-oriented programming. It also discusses why programming is a foundational skill for learning network programmability and covers the fundamentals of the Python programming language, including installing Python Version 3.x, executing Python programs, input and output, data types, data structures, operators, conditional statements, loops, and functions.

Chapter 6, "Python Applications": This chapter builds on Chapter 5 and covers the application of Python to different domains. The chapter illustrates the use of Python for creating web applications using Django and Flask, for network programmability using NAPALM and Nornir, and for orchestration and machine learning. The chapter also covers some very important tools and protocols used in software development in general, such as Git, containers, Docker and virtual environments.

PART IV, "Transport"

Chapter 7, "HTTP and REST": This is one of the most important chapters in this book. It introduces the HTTP protocol and the REST architectural framework, as well as the relationship between them. This chapter covers HTTP connections based on TCP. It also covers the anatomy of HTTP messages and dives into the details of HTTP request methods and response status codes. It also provides a comprehensive explanation of the most common header fields. The chapter discusses the syntax rules that govern the use of URIs and then walks through working with HTTP, using tools such as Postman, cURL and Python libraries, such as the requests library.

Chapter 8, "Advanced HTTP": Building on Chapter 7, this chapter moves to more advanced HTTP topics, including HTTP authentication and how state can be maintained over HTTP connections by using cookies. This chapter provides a primer on cryptography for engineers who know nothing on the subject and builds on that to cover TLS, and HTTP over TLS (aka HTTPS). It also provides a glimpse into HTTP/2 and HTTP/3, and the enhancements introduced by these newer versions of HTTP.

Chapter 9, "SSH": Despite being a rather traditional protocol, SSH is still an integral component of the programmability stack. SSH is still one of the most widely used protocols, and having a firm understanding of the protocol is crucial. This chapter discusses the three sub-protocols that constitute SSH and cover the lifecycle of an SSH connection: the SSH Transport Layer Protocol, User Authentication Protocol, and Connection Protocol. It also discusses how to set up SSH on Linux systems as well as how to work with SSH on the three major network operating system: IOS XR, IOS XE, and NX-OS. Finally, it covers SFTP, which is a version of FTP based on SSH.

PART V, "Encoding"

Chapter 10, "XML": This chapter covers XML, the first of three encoding protocols covered in this book. XML is the oldest of the three protocols and is probably the most sophisticated. This chapter describes the general structure of an XML document as well as XML elements, attributes, comments, and namespaces. It also covers advanced XML topics such as creating document templates using DTD and XML-based schemas using XSD, and it compares the two. This chapter also covers XPath, XSLT, and working with XML using Python.

Chapter 11, "JSON": JSON is less sophisticated, newer, and more human-readable than XML, and it is therefore a little more popular that XML. This chapter covers JSON data formats and data types, as well as the general format of a JSON-encoded document. The chapter also covers JSON Schema Definition (JSD) for data validation and how JSD coexists with YANG.

Chapter 12, "YAML": YAML is frequently described as a superset of JSON. YAML is slightly more human-readable than JSON, but data encoded in YAML tends to be significantly lengthier than its JSON-encoded counterpart. YAML is a very popular encoding format and is required for effective use of tools such as Ansible. This chapter covers the differences between XML, JSON, and YAML and discusses the structure of a YAML document. It also explains collections, scalers, tags, and anchors. Finally, the chapter discusses working with YAML in Python.

PART VI, "Modeling"

Chapter 13, "YANG": At the heart of the new paradigm of network programmability is data modeling. This is a very important chapter that covers both generic modeling and the YANG modeling language. This chapter starts with a data modeling primer, explaining what a data model is and why it is important to have data models. Then it explains the structure of a data model. This chapter describes the different node types in YANG and their place in a data model hierarchy. It also delves into more advanced topics, such as augmentations and deviations in YANG. It describes the difference between open-standard and vendor-specific YANG models and where to get each type. Finally, the chapter covers a number of tools for working with YANG modules, including **pyang** and **pyangbind**.

PART VII, "Protocols"

Chapter 14, "NETCONF and RESTCONF": NETCONF was the first protocol developed to replace SNMP. RESTCONF was developed later and is commonly referred to as the RESTful version of NETCONF. Building on earlier chapters, this chapter takes a deep dive into both NETCONF and RESTCONF. The chapter covers the protocol architecture as well as the transport, message, operations, and content layers of each of the two protocols. It also covers working with these protocols using Python.

Chapter 15, "gRPC, Protobuf, and gNMI": The gRPC protocol was initially developed by Google for network programmability that borrows its operational concepts from the communications models of distributed applications. This chapter provides an overview of the motivation that drove the development of gRPC. It covers the communication flow of gRPC and protocol buffers (Protobuf) used to serialize data for gRPC communications. The chapter also shows how to work with gRPC using Python. The chapter then takes a deep dive into gNMI, a gRPC-based specification. Finally, the chapter shows how gRPC and gNMI are used to manage a Cisco IOS XE device.

Chapter 16, "Service Provider Programmability": Service providers face unique challenges due to the typical scale of their operations and the stringent KPIs that must be imposed on their networks, especially given the heated race to adopt 5G and associated technologies. This chapter discusses how such challenges influence the programmability and automation in service provider networks and provides in-depth coverage of Segment Routing, BGP-LS, and PCEP.

PART VIII, "Programmability Applications"

Chapter 17, "Programming Cisco Platforms": This chapter explores the programmability capabilities of several Cisco platforms, covering a wide range of technology domains. In addition, this chapter provides several practical examples and makes heavy use of Cisco's DevNet sandboxes. This chapter covers the programmability of IOX XE, IOS XR, NX-OS, Meraki, DNA Center, and Cisco's collaboration platforms, with a use-case covering Webex Teams.

Chapter 18, "Programming Non-Cisco Platforms": This chapter covers the programmability of a number of non-Cisco platforms, such as the Cumulus Linux and Arista EOS platforms. This chapter shows that the knowledge and skills gained in the previous chapters are truly vendor neutral and global. In addition, this chapter shows that programmability using APIs does in fact abstract network configuration and management and breaks the dependency on vendor-specific CLIs.

Chapter 19, "Ansible": This chapter covers a very popular tool that has become synonymous with network automation: Ansible. As a matter of fact, Ansible is used in the application and compute automation domains as well. Ansible is a very simple, yet extremely powerful, automation

tool that provides a not-so-steep learning curve, and hence a quick and effective entry point into network automation. This is quite a lengthy chapter that takes you from zero to hero in Ansible.

PART IX, "Looking Ahead"

Chapter 20, "Looking Ahead": This chapter builds on the foundation covered in the preceding chapters and discusses more advanced technologies and tools that you might want to explore to further your knowledge and skills related to network programmability and automation.

Part I: Introduction

Chapter 1. The Network Programmability

We all have that one story we tell on ourselves about some stupid mistake that brought down a network segment or even an entire network. Here's mine.

Thirty years ago, I was sitting in an office in Albuquerque, logged in to a router in Santa Fe, making some minor, supposedly nondisruptive modifications to the WAN interface. I wanted to see the changes I had made to the config, and I got as far as typing **sh** of the IOS **show** command before realizing I was still in interface config mode and needed to back out of it before entering the show command. But instead of backspacing or taking some other moderately intelligent action, I reflexively hit Enter.

The router, of course, interpreted **sh** as **shutdown**, did exactly what it was told to do, and shut down the WAN interface—the only interface by which the router was remotely accessible. There was no warning message. No "You don't want to do that, you idiot." The WAN interface just went down, leaving me no choice but to drive the 60 miles to Santa Fe to get physical access to the router, endure the sour looks of the workers in the office I had isolated, and turn the interface back up.

There are other stories. Like the time not too many years after The Santa Fe Incident when I mistyped a router ID, causing the OSPF network to have duplicate RIDs and consequently misbehave in some interesting ways. I think that one later became a troubleshooting exercise in one of my books.

My point is that configuration mistakes cause everything from annoying little link failures to catastrophic outages that take hours or days to correct and put your company on the front pages of the news. Depending on the study you read, human error accounts for 60% to 75% of network outages.

Every network outage has a price, whether it's the cost of a little branch office being offline for an hour or a multinational corporation suffering millions of dollars in lost revenue and damaged reputation.

Even when we're not making configuration mistakes, we *Homo sapiens* tend to be a troublesome and expensive feature of any network.

The cost of building a network (CAPEX) has always been outweighed by the cost of running that network (OPEX). And that operational cost is more than just paying people to configure, change, monitor, and troubleshoot the network. There are costs associated with direct human operations, such as the following:

• Configuration mistakes, large and small, which are exacerbated by working under pressure during network outages

• Failure to comply with configuration standards

• Failure to even *have* configuration standards

• Failure to see and correctly interpret network telemetry that indicates impending trouble

• Failure to maintain accurate network documentation

• Having network experts constantly in "firefighting mode" rather than performing steady-state network analysis and advanced planning

It's important to emphasize that network automation and programmability do not necessarily mean reducing the workforce, although workers are going to require some retraining. At its best, automation makes network staff more valuable by removing their daily "firefighting drills," allowing them to spend their time thinking about the 3- and 5-year network plan; evaluating new technologies, vendor solutions, and industry trends; analyzing whether the network can better serve company objectives; and just keeping a better eye on the big picture.

Pilots of Boeing 777s report that on an average flight, they spend just 7 minutes manually flying the plane. They are quick to emphasize, however, that while the autopilot is doing the flying, the pilots are still very much in control. They input instructions and expectations, and then supervise the automated processes. The autopilot performs the mundane physical tasks necessary to fly the plane, and it probably performs those tasks more quickly and accurately than most pilots do. The pilots, freed from the distractions of manual flying, apply their expertise to monitoring approaching weather and flight patterns, keep an eye on the overall health of the plane, and even look over the shoulder of the autopilot to be sure it is correctly executing the instructions they gave it. The pilot's role is expanded, not diminished.

The pilot tells the airplane what he wants (that is, programming), and the plane does what it is told (that is, automation). We don't have this level of artificial intelligence and machine learning in our networks yet, but that's where we're headed.

First, a Few Definitions

There's a fair amount of confusion around the concepts discussed in this book. Is automation just a part of network management? Are automation and programmability the same thing? How does orchestration fit in? And does SDN really stand for "Still Does Nothing"?

Network Management

The terms *automation*, *programmability*, *orchestration*, *virtualization*, *SDN*, and *intent*—all of which are defined in this section—apply, in one way or another, to network management. So let's start by defining that:

Network management is how you make a network meet whatever expectations you have of it.

This is about as simple a definition that you can get, but behind this one sentence is arrayed an extensive repository of systems, processes, methodologies, rules, and standards pertaining to the management of all aspects of the network.

One framework for sorting out all the aspects of a network to be managed is FCAPS, which represents the following areas:

• Fault management

• Configuration management

• Accounting management

• Performance management

• Security management

It's doubtful that you would be reading this book if you didn't already know what these five areas represent. You probably also know that there are deep aspects of each. Configuration management, for example, covers not just provisioning but configuration standards and procedures, change management, configuration change tracking, reference designs, configuration file archiving, and the specialized systems to support all that stuff.

You probably also hear ITIL discussed regularly in the context of network management. ITIL, which stands for Information Technology Infrastructure Library, is a library of principles, processes, and procedures that support FCAPS but that also goes beyond that framework to apply to personnel and organizations, IT products, partners, suppliers, practices, and services that go into managing the network. Whereas FCAPS is system oriented, ITIL is services and governance oriented.

The outline of the ITIL 4 management practices provide an example of the complexity of ITIL:

• General management practices

• Architecture management

• Continual service improvement

• Information security management

• Knowledge management

• Measurement and reporting

• Organizational change management

• Portfolio management

• Project management

• Relationship management

• Risk management

• Service financial management

• Strategy management

• Supplier management

• Workforce and talent management

• Service management practices

• Availability management

• Business analysis

• Capacity and performance management

• Change control

• Incident management

• IT asset management

• Monitoring and event management

• Problem management

• Release management

• Service catalog management

• Service configuration management

• Service continuity management

• Service design

• Service desk

• Service level management

• Service request management

• Service validation and testing

• Technical management practices

• Deployment management

• Infrastructure and platform management

• Software development and management

This is quite a list, and it covers only the top-level topics. Fortunately for our definitions, we don't have to go into all of them. I just wanted to show you how extensive and formalized network management can be. For the purposes of this book we don't need to go into the highly structured, highly detailed ITIL specifications. Most of the topics in this book support the simpler FCAPS framework; in fact, most topics in the book support configuration management.

Managing a network system means interacting with the system in some way. It usually involves the following:

• Accessing the CLI via SSH (don't use Telnet!) or directly via a console port for configuration, monitoring, and troubleshooting

• Monitoring (and sometimes changing) the system through Simple Network Management Protocol (SNMP) agents and Management Information Bases (MIBs)

• Collecting system logs via syslog

• Collecting traffic flow statistics with NetFlow or IP Flow Information Export (IPFIX)

• Sending information to and extracting information from network devices through Application Programming Interfaces (APIs), whether the APIs are RESTful (such as RESTCONF) or not (such as NETCONF or gRPC)

Automation

Automation, very simply, means using software to perform a task you would otherwise do manually. And automation is nothing new or unfamiliar to you. Routing protocols, for example, are automation programs that save you the work of manually entering routes at every network node. DNS is an automation program that saves you from having to look up the IP address of any destination you want to talk to. You get the point.

Automation software might be built into a network node, might be a purchased software platform, or might be a program or script you create yourself.

That last bit -- creating your own automation routines -- is what this book is all about: It gives you the fundamentals to be able to understand and operate the underlying protocols used by products, as well as utilize those protocols in your scripts and programs.

Besides the obvious benefit of making life easier, automation provides the following advantages:

• Fast rollout of network changes

• Relief from performing routine repetitive tasks

• Consistent, reliable, tested, standards-compliant system changes

• Reduced human errors and network misconfigurations

• Better integration with change control policies

• Better network documentation and change analysis

Out of all of these advantages, you might be inclined to choose speed of deployment as the most important. Being able to deploy a network change "with a push of a button" definitely is less expensive than visiting each network node and manually reconfiguring. The time savings increase dramatically as the number of affected nodes increases.

However, consistent and reliable network changes, along with reduced human error (that is, accuracy) are of even greater benefit than speed of deployment. The significance of accuracy becomes more obvious as the number of times a change has to be implemented increases. Implementing a network change on five devices can be done fairly accurately using primitive tools and elevated vigilance. This may not be possible when implementing the same change on 1000 devices. Speed saves operational expense during deployment, but accuracy provides cumulative benefits over the life of the network.

Orchestration

Orchestration, in the traditional musical sense, is the composition of musical parts for a diversity of instruments. When the instrumentalists play their individual parts together—usually under the direction of a conductor—you get Beethoven's Fifth or the theme to *Lion of the Desert* or *Lawrence of Arabia*.

Orchestration in the IT sense is very much the same: Individual elements work together, following their own instructions, to create a useful service or set of services. For example, the deployment of a certain application in a data center is likely to require compute, storage, security, and network resources. Orchestration enables you to coordinate and deploy all of those resources to accomplish one goal.

Does this sound like automation? Well, yes and no. It's true that the differences between automation and orchestration can sometimes get fuzzy, but here's the difference:

• *Automation* is the performance of a single task, such as a configuration change across a set of switches or routers or the deployment of a virtual machine on a server, without manual (human) intervention.

• *Orchestration* is the coordination of many automated tasks, in a specific sequence, across disparate systems to accomplish a single objective. Another term for this is *workflow*.

So, automation performs individual tasks, and orchestration automates workflows. Both automation and orchestration save time and reduce human error.

A wealth of orchestration tools are available on the market, including the following:

• VMware vRealize Orchestrator, for VMWare environments

• OpenStack Heat, for OpenStack

• Google Cloud Composer, for (you guessed it) orchestrating Google Cloud

• Cisco Network Services Orchestrator (NSO), which, as the name implies, focuses on network services

• RedHat Ansible, which is usually used as a simple automation tool but can also perform some workflow automation

• Kubernetes, a specialized platform for orchestrating containerized workloads and services

Programmability

It's an understandable misconception that programmability is a part of automation. After all, most automation does not work unless you give it operating parameters. A routing protocol, for instance, doesn't do anything unless you tell it what interfaces to run on, perhaps what neighbors to negotiate adjacencies with, and what authentication factors to use.

Is network programmability, then, just providing instructions to automation software? No, that's configuration.

Programmability is the ability to customize your network to your own standards, policies, and practices. Programmability enables you to operate your network and the services it supports as a complete entity, built to support the specifics of your business. In this age in which most businesses depend on their applications and are built around their networks, that's huge.

Isn't that the way it should always have been? Your network should comply to your requirements; you should not have to adjust your requirements to comply to your network. Once you can customize your network to your own standards, you have the power to innovate, to quickly adapt to competitive challenges, and to create advantages over your competitors. These are all far more important advantages than just operational savings, reduced downtime, and faster problem remediation. (Although you get all that, too.)

But programmability, as a technical marketing term today, has a slightly different meaning. Programmability, used in this context, is the ability to monitor devices, retrieve data, and configure devices through a *programmable interface*, which is a software interface to your device through which other software can speak with your device. This interface is formally known as an A*pplication Programming Interface* (*API*).

What is the difference between a legacy interface such as the CLI and an API? For one, a CLI was created with a human operator in mind. A human types commands into the terminal and receives output to the screen. On the other hand, an API is used by other software, such as automation products or custom Python scripts to speak with a device without any human interaction, apart from writing the Python script itself or configuring the device parameters on that automation product.

APIs are covered in a lot of detail in this book because they are a foundational building block for any software-to-software interaction. Instead of reading an exhaustive comparison between legacy interfaces and APIs, you will see for yourself the major advantages of interacting with your network through programmable interfaces as you progress through the chapters of this book.

Virtualization and Abstraction

Virtualization is one of those words you've known and understood throughout your career. First there are those *V* acronyms: VPN, VPLS, VLAN, VXLAN, VC, VM, VRF, VTEP, OTV, NVE, VSAN, and more. There is often virtualization even when the word itself isn't used. TCP, for example, provides a virtual point-to-point connection over connectionless IP by using handshaking, sequence numbers, and acknowledgements.

Virtualization is the creation of a service that behaves like a physical service but is not. We use virtualization to share resources, such as consolidating multiple data networks over a shared MPLS cloud, communicating routing tables (VRFs) for multiple isolated networks and security zones over a shared MP-BGP core, implementing multiple VLANs on one physical LAN, or creating virtualized servers on a single physical server. The motivation might be to create a bunch of different services when you have only one physical resource to work with, or it might be to more efficiently use that resource by divvying it up among multiple users, each of whom gets the impression that they are the only one using the resource.

Boiling all this down to a simple definition, virtualization is a software-only or software-defined service built on top of one or more hardware devices. In the case of a virtualized network, the network might look quite different from the underlying physical network. For example, in Figure 1-1, from the individual perspectives of VLANs 100, 200, and 300, each is connected to a single switch, and none is aware of the other two VLANs. In Figure 1-2, the Layer 3 VPNs Red, Green, and Blue are built on top of a single MPLS infrastructure but are aware only of their own VPN peers.

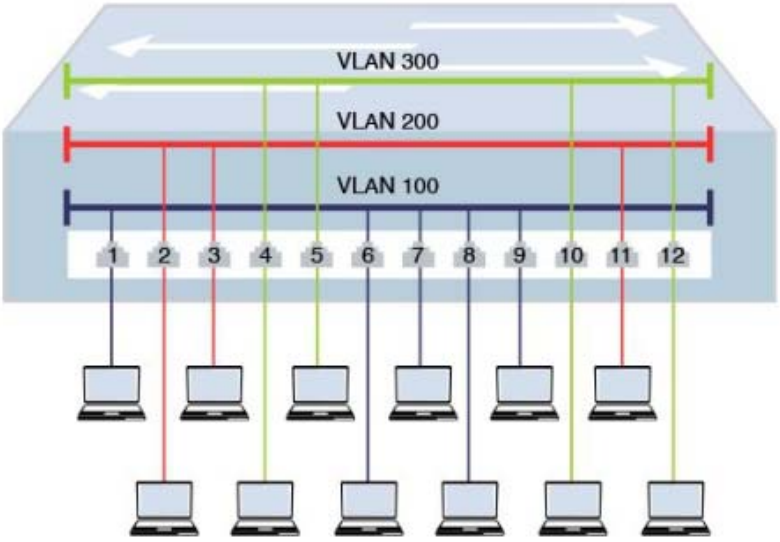


Figure 1-1 *VLANs Connected to a Switch Are Not Aware of Each Other*

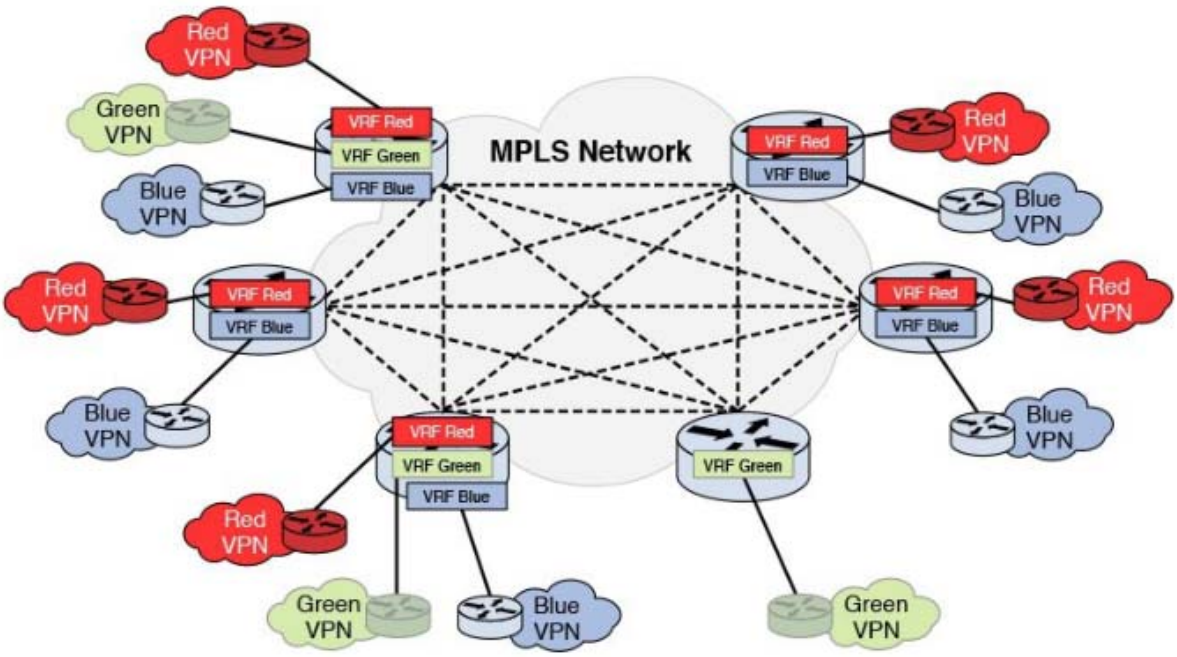


Figure 1-2 *VPNs Built on a Single MPLS Infrastructure That Are Aware Only of Their Own VPN Peers*

And here are a couple more definitions related to the networks pictured here: An *overlay* network is a software-defined network running over a physical *underlay* network. You'll encounter overlays and underlays particularly in data center networking.

Abstraction is a term you may not understand clearly, although you have certainly heard it used in the context of *network abstractions*. You also likely use the concept often when you're whiteboarding some network, and you draw a cloud to represent the Internet, an MPLS core, or some other part of a network, where you just mean that packets go in at one edge and come out at some other edge.

Abstraction goes hand-in-hand with virtualization because we build virtualized services on top of abstractions. The "whiteboard cloud" example illustrates this: Our whiteboard discussion is focused on the details of ingress and egress packet flows, not on the magic that happens in the cloud to get the packets to the right place.

Virtual machines are, for instance, built on an abstraction of the underlying physical server (see Figure 1-3). The server abstraction is the CPU, storage, memory, and I/O allotted to the VM rather than the server itself.

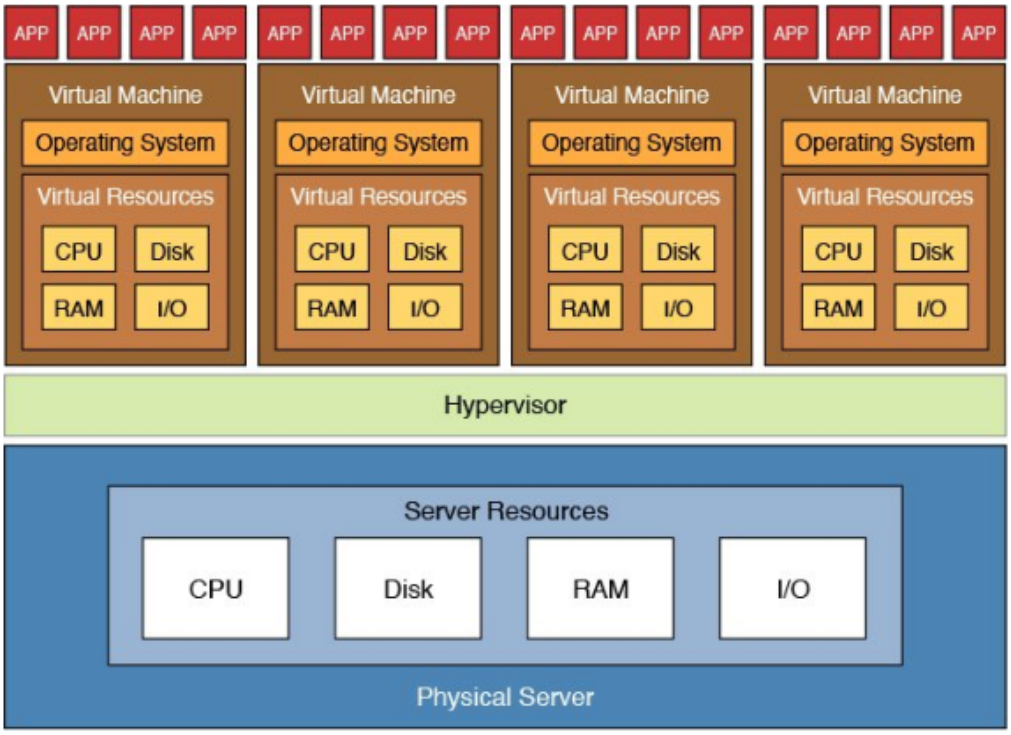


Figure 1-3 *A Server Abstracted into the Components Allotted to Each VM*

Another example, sticking with servers, is a container platform such as Docker (see Figure 1-4) packages up application code and its dependencies into containers that are isolated from the underlying server hardware. The advantage of both VMs and containerized applications is that they can be deployed, changed, and moved independently of the physical infrastructure.

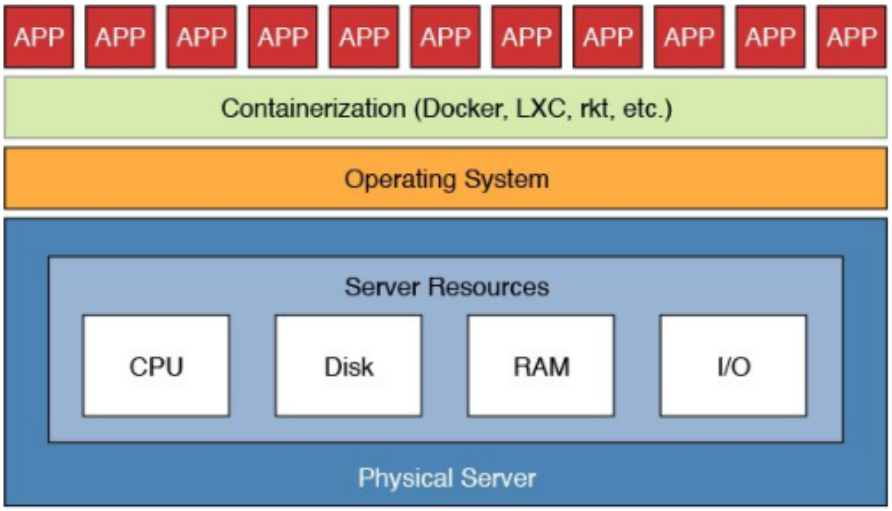


Figure 1-4 *Using Containers to Abstract Away the Underlying Server and Operating System for Individual Applications*

Network abstraction is the same idea but with more elements. By adding an "abstraction layer"—or abstracting away the network—you focus only on the virtualized network: adding, changing, and removing services independently of the network infrastructure (see Figure 1-5). Just as a VM uses some portion of the actual server resources, virtual network services use some portion of the physical network resources. Network abstraction can also allow you to change infrastructure elements without changing the virtual network.

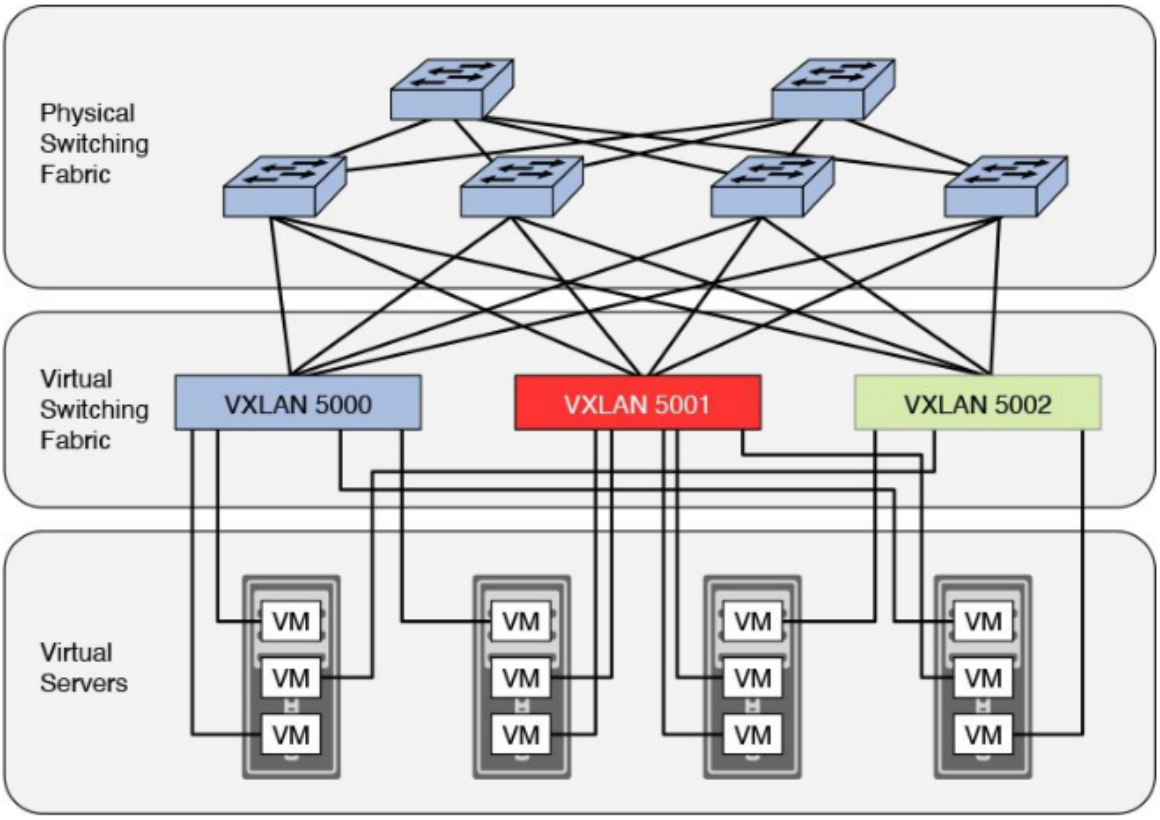


Figure 1-5 *Data Center Infrastructure Abstracted Away by VXLAN*

A network abstraction layer is essential for efficient automation and programmability because you want to be able to control your network independently of the specifics of vendors and operating systems. One of the things you will learn in Chapter 6, "Python Applications," for example, is how to use a Python library called NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor Support). In Chapter 13, "YANG," you'll learn about YANG, a network modeling language (that is, a language for specifying a model, or an abstraction, of the network).

The abstraction, or model, of the network serves as a *single source of truth* for your automation and orchestration. Do you have enough resources for a service that is about to be deployed? What's the available bandwidth? How will RIBs and FIBs change, and is there enough memory capacity to support the changes? What effect will the added service have on existing Layer 2 or Layer 3 forwarding behavior?

Without a single source of truth, the "intelligent" part of your automation or orchestration must reach out and touch every element in the network to gather the information it needs for pre-deployment verification. Each element is its own source of truth and might or might not express its truth consistently with other elements—especially in multivendor networks. A single source of truth, if built properly, continuously collects network state and telemetry to provide a real-time, accurate, and relevant model of the network. Every service you want to deploy can then be verified against this abstraction before it is deployed, increasing your confidence and decreasing failures.

But don't confuse this perspective of a single source of truth with a Configuration Management Database (CMDB), which is a repository of what the network state *should* be and, therefore, is not updated from the live network. Instead, the live network state is compared to the CMDB to verify its compliance.

Network abstraction gives rise to N*etwork as Code* (NaC) or the broader *Infrastructure as Code*, which encompasses network, storage, and compute. NaC is the code that ties together network abstraction, virtualization, programming, and automation to create an intelligent interface to your network.

NaC also brings networking into the DevOps realm and enables the application of proven software practices such as Continuous Integration/Continuous Delivery/Continuous Deployment (CI/CD), illustrated in Figure 1-6. Among the many tools you can use for developing NaC is RedHat Ansible, which is covered in Chapter 19, "Ansible."

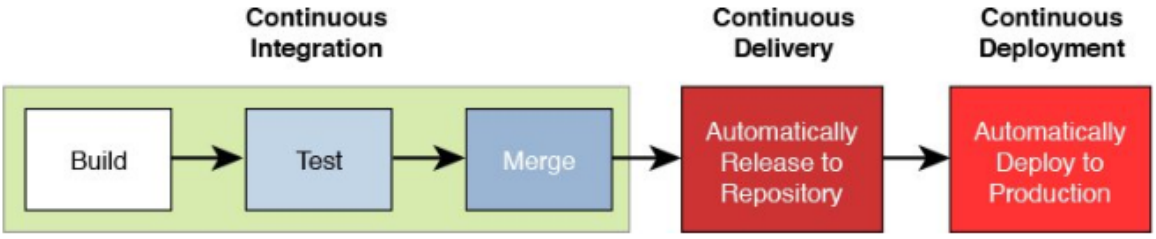


Figure 1-6 *CI/CD*

With your new network code developed, tested, and merged with existing code and passed to the virtualization layer, the last bit of the workflow is to translate the generalized code into verified, vendor- and operating system–specific configurations and push them to physical network elements. Interactive scripts using languages such as Expect, TCL, and Perl were—and sometimes still are— used to log in to the network devices and configure them; these scripts just automate the actions an operator would take to manually configure the device via the CLI.

These days, automation tools interact with networking devices through APIs, which are themselves abstractions of the underlying physical device. The difference is that the APIs reside on the individual devices and are specific to their own device. Automation software usually

communicates with the APIs via eXtensible Markup Language (XML) or JavaScript Object Notation (JSON), covered in Chapters 10, "Extensible Markup Language (XML) and XML Schema Definition (XSD)," and 11, "JavaScript Object Notation (JSON) and JSON Schema Definition (JSD)." You'll find that even the CLIs of modern routers and switches are actually applications running on top of the local APIs rather than direct interfaces to the operating systems.

Software-Defined Networking

Software-Defined Networking (SDN) isn't covered in this book, but all this discussion of automation, programmability, network abstraction, and APIs merits at least a mention of SDN.

The "SDN 101" concept of the technology is that SDN is a centralized control plane on top of a distributed forwarding plane. Instead of a network of switches and routers that each have their own control planes, SDN "pops the control planes off" and centralizes them in one controller or a controller cluster. The control plane is greatly simplified because individual control planes no longer have to synchronize with each other to maintain consistent forwarding behavior.

This concept embodies much of what we've been discussing in the previous sections: separation of physical infrastructure from service workflows, a network abstraction layer, and a single source of truth. Incorporating everything we've previously discussed provides a more refined definition of SDN:

SDN is a conceptual framework in which networks are treated as abstractions and are controlled programmatically, with minimal direct touch of individual network components.

This definition still adheres to the idea of centralized control, but it encompasses a wider set of SDN solutions, such as SD-WAN, that virtualizes the Wide-Area Network and places it under centralized control and subject to vendor SDN solutions such as Cisco's Application Centric Infrastructure (ACI) and VMware's NSX. The definition also takes in products such as Cisco's Network Services Orchestrator (NSO) that don't really fit in the more traditional definition of SDN.

Note

ACI and NSX are often lumped together when giving examples of SDN solutions. While they do many of the same things, there are also some significant differences in how they work and what they do. For instance, ACI has a different approach to network abstraction than the approach described here.

Intent-Based Networking

Intent-Based Networking (IBN) is the next evolutionary step beyond SDN. Like SDN, it isn't covered in this book, but it is certainly based on the concepts described so far. Chapter 20, "Looking Ahead," has more to say about IBN; you'll also get some exposure to it in the discussion of Cisco DNA Intent APIs in Chapter 17, "Programming Cisco Devices."

SDN gives you a centralized control point for your network, but you still have to provide most of the intelligence to deploy or change the underlay and overlay. In other words, you still have to tell the control plane how to do what you want it to do.

IBN adds an interpretive layer on top of the control plane that enables you to just express *what* you want—that is, your intent—and IBN translates your expressed intent into *how* to do it. Depending on the implementation, the IBN system either then pushes the developed configurations to a controller for deployment to the infrastructure or (more often) acts as the control plane itself and pushes configurations directly to the infrastructure.

Once your intent is configured (*intent fulfillment*), an IBN system uses closed-loop telemetry and state tables to monitor the network and ensure that it does not drift from your expressed intent (*intent assurance*).

IBN is still in its infancy as this book is being written, but it holds enormous potential for transforming the way we operate our networks. You'll learn more about this in Chapter 20.

Your Network Programmability and Automation Toolbox

All of the definitions so far in this chapter bring us to an important question: What tools does an adept network engineer and architect need to carry? And with that question we arrive at the entire purpose of this book.

One of the reasons for spending so much time on definitions is to be able to classify various tools and to understand the relationships among those classifications. Figure 1-7 offers one perspective on how you might classify tools within the functions discussed in the previous section and a number of functions not discussed in this chapter.

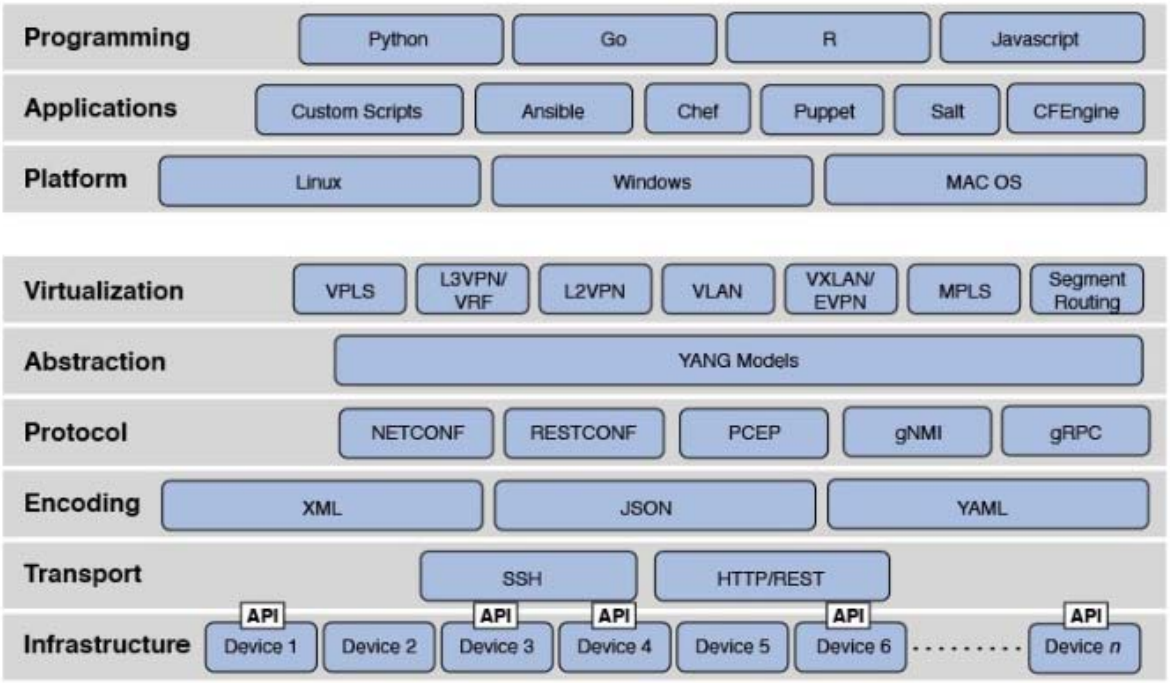


Figure 1-7 *The Network Programmability and Automation Ecosystem*

Before going further, it's important to note that Figure 1-7 is just *one* perspective. The order in which the Application, Automation, Platform, Virtualization, and Abstraction layers appear and how they interact can vary according to the network environment. What's more important are the tools available to you within the various layers.

It's also important to note that I've provided examples of more tools than are covered in this book. And that gets us to why the authors of this book have chosen the tools we have for you to learn.

Python

At the top of the programmability and automation ecosystem are programming languages. Python, Go, R, and JavaScript are given as examples in Figure 1-7. There are, of course, other programming languages that could be added here, C and C++ being the most prominent of them, although they are used more by people making their living at software development than by people making their living at other things—like networking—who need to be able to write programs and scripts to make their job easier. There are also a number of languages that we could add to the list, such as Perl, Expect, and Tcl, that are still around to one degree or another but that have been overshadowed by newer, more powerful languages. Like Python.

Which brings us to why this book exclusively covers Python: It's by far the most widely used programming language for network automation, supporting a terrific number of libraries, modules, and packages specific to networking. Python is easy to learn, easy to use, and easy to debug, which fits the bill for networkers who just need to get their job done without having to become professional programmers. That said, Python is far from a "beginner" language. It's used extensively by companies such as Facebook, Netflix, Instagram, Reddit, Dropbox, and Spotify. Google software developers even have a saying: "Use Python where we can, C++ where we must."

Python is versatile, working equally well for scripting and as a glue language (for tying together modules written in other languages). It's also highly portable to different platforms. Once you know a little Python, you might even find yourself using it for quick little tasks such as running math calculations.

Finally, the more you use packaged automation products such as Ansible or Cisco ACI or interact with network devices through their APIs rather than directly with their CLI, the more you'll find Python to be an essential tool in your toolbox. Chapter 5, "Python Fundamentals," covers the basics of Python, and Chapter 6, "Python Applications," covers some useful libraries and tools that you will want to use when automating your network using Python.

Ansible

The next category of tools in your programmability and automation toolbox is applications. And first on that list are custom scripts. If you are already wielding a programming language such as Python to perform your job, you almost certainly have a collection of scripts that you use to automate everyday repetitive tasks. The more proficient you are, the more useful your scripts become. You'll learn how to script some of the boring parts of your job in this book.

Also in the applications category are a number of prebuilt automation platforms that you can either download for free or purchase: Ansible, Salt, Chef, Puppet, and CFEngine are examples, but there are many others. What they have in common is that they all began life as platforms for automating server management. If you're in a DevOps shop or any environment that orchestrates large numbers of end systems, your organization probably already has a favorite automation platform from this list.

We've chosen Ansible as the automation engine to familiarize you with in this book. Not only is Ansible open source and available for the very reasonable price of free, it is the most popular automation framework among networkers. It's easy to learn and integrates well as a Python module; in fact, Ansible is written in Python. Even if you end up using some other framework within your organization, having a grounding in Ansible is valuable and will give you a head start in understanding the concepts of any of this class of automation platforms. Ansible is covered in Chapter 19.

Linux

The next tool in the lineup is the platform on which you're doing your programming and running your automation. Not the hardware itself but the operating system on the hardware. Figure 1-7 lists the three most well-known operating systems: Linux, Windows, and macOS. For each of these, there are specific versions and distributions. For example, Linux includes Fedora, CentOS, SuSE, Ubuntu, and many others. Under Windows are the many incarnations of Windows Server, Windows 7, 8, 10, and so on. There are also platform-specific operating systems on which your automation applications can run (for example, Cisco IOS XE and NX-OS).

Recall the earlier comment about Figure 1-7 being just one perspective on how the programmability and automation ecosystem is organized. The Programming, Applications, and Platform tools might be running on a management server. They might be running on your laptop. One or more of the layers might be running directly on top of an infrastructure device or themselves might be part of the infrastructure. So, don't take Figure 1-7 as the only way the various elements of the ecosystem might interact with each other.

For network programmability and automation, you need to have a strong working knowledge of Linux. Three chapters in this book are dedicated to Linux: Chapter 2, "Linux Fundamentals," Chapter 3, "Linux Networking and Security," and Chapter 4, "Linux Scripting." Here's just a few of the reasons Linux needs to be part of your toolbox:

• Linux is the most widely used operating system in IT environments, running more than two-thirds of the servers on the Internet. Linux is used as a server OS and also for the following:

• Automation

• Virtualization and containers

• Programming and scripting

• Software-Defined Networking

• Big Data systems

• Cloud computing

• Linux supports a huge number built-in networking features.

• Linux supports a huge number of development tools, such as Git.

• Linux supports a number of automation tools and supporting capabilities, including almost everything shown in Figure 1-7.

• Python interpreters (along with many other languages) run natively on Linux, and many Linux distributions come with Python already built in.

• The vast majority of network operating systems today (such as Cisco NX-OS, IOS XE, and IOS XR) run as applications on top of some Linux distribution. Some entire cloud platforms, such as OpenStack, are supported in Linux. Even macOS is very Linux-like under the hood.

• Although there are paid versions of Linux, such as Red Hat Enterprise Linux (RHEL), what you're primarily paying for is support. Linux distributions for the most part are free to download and use.

• Because Linux is open source, with enormous development support worldwide, the source code is tremendously reliable, stable, and secure.

Virtualization

"Wait a minute," you might say, "the services you show for the virtualization layer run on individual infrastructure devices. What are they doing separated from the infrastructure?"

You're right, the services themselves run on network devices. But what all of them represent are different forms of virtualized overlays to the physical underlay network. Think of the overlay and the underlay as the top and bottom of your network data plane. Between them are sandwiched all the layers that implement the virtualized overlay onto the physical underlay.

YANG

You've already read about abstraction in this chapter: Abstraction means a generic model of your network. Hence, it is closely associated with the virtualization layer. In Figure 1-7, the only modeling language shown is YANG (Yet Another Next Generation). There are other data modeling languages, such as Unified Modeling Language (UML) and NEMO (NEtwork MOdeling), but YANG is used so extensively for network modeling that it is the only language shown Figure 1-7. You'll learn all about using YANG in Chapter 13.

Protocols

The protocols layer dictates a programmatic interface for accessing or building the abstraction of a network. Protocols may be RESTful, such as RESTCONF, or not, such as NETCONF or gRPC. A protocol uses a particular encoding for its messages. NETCONF, for example, uses XML only, whereas RESTCONF supports both XML and JSON. A protocol uses a particular transport to reach a device. RESTCONF uses HTTP, while NETCONF uses SSH. A protocol uses Remote Procedure Calls (RPCs) to install, manipulate, and delete configurations based on your model or retrieve configuration or operational data based on your model. Models are described in YANG. The protocols shown in Figure 1-7 are all covered in this book in Chapters 14, "NETCONF and RESTCONF," 15, "gRPC, Protobuf, and gNMI," and 16, "Service Provider Programmability."

Encoding the Protocols

The protocols themselves need a common language to communicate with the infrastructure, and this is the purpose of the encoding layer. eXtensible Markup Language (XML), JavaScript Object Notation (JSON), and Yet Another Markup Language (YAML) are the most common

Chapter 2. Linux Fundamentals

Chapter 1, "The Network Automation and Programmability Ecosystem," discusses where operating systems (such as Windows, UNIX, and Linux) fit in the big picture of programmability and automation. As indicated in Chapter 1, today Linux is the predominant operating system used by developers and network engineers alike—and for good reasons. This chapter is dedicated to Linux fundamentals. It starts with an assumption that you know nothing about Linux. By the end of the chapter, you will have gained enough knowledge and hands-on experience to successfully install, operate, and maintain a Linux-based system. This system will be the first building block in the development environment you will use to apply most of the material covered in subsequent chapters of this book.

The Story of Linux

This section introduces the Linux operating system: how it started, where it stands today, and where it is headed in the future. It also touches on the architecture of the operating system and introduces the concept of Linux distributions.

History

The Linux operating system was first developed in 1991 by a Finnish computer science student at the University of Helsinki called Linus Torvalds. His motivation was to provide a free alternative to the UNIX-like operating system MINIX that would run on Intel's 80386 chipset. The majority of the Linux kernel was written in the C programming language.

The first release of Linux consisted of only a kernel. A *kernel* is the lowest-level software component of an operating system and is the layer that acts as an interface between the hardware and the rest of the operating system. A kernel on its own is not very useful. Therefore, the Linux kernel was bundled with a set of free software utilities developed under a project called *GNU* (which is a recursive acronym for GNU's Not Unix). In 1992, Linux was relicensed using the *General Public License Version 2* (*GPLv2*), which is also a part of the GNU project. Together, the kernel and GNU utilities made up the Linux operating system. A group of developers worked on developing the Linux kernel as well as integrating the kernel with GNU software components in order to release the first stable version of Linux, Linux 1.0, in March 1994. In the following few years, most of the big names in the industry, such as IBM, Oracle, and Dell, announced their support for Linux.

Even though Linux is licensed under the GPL and is, therefore, free, companies have built businesses around Linux and made a lot of money out of it. Companies like Red Hat have made make money by packaging the free Linux kernel along with other software components, bundled with subscription-based support services. This product is then sold to customers who do not want to have to depend on the goodwill of the open source community to receive support for their Linux servers that are running mission-critical applications.

Linux Today

Today, Linux is supported on virtually any hardware platform, and most commercial application developers provide versions of their software that run on Linux. Linux powers more than half of the servers on the Internet. More than 85% of smartphones shipped in 2017 ran on Android, a Linux-based operating system. More smart TVs, home appliances, and even cars are running some version of Linux every day. All supercomputers today run on Linux. Most network devices today either run on Linux or on a Linux-like network operating system (NOS), and many vendors expose a Linux shell so that network engineers can interact directly with it. The Linux shell is covered in detail later in this chapter.

Linux Development

Linux is an open-source operating system that is developed collaboratively by a vast number of software developers all over the world and sponsored by a nonprofit organization called the Linux Foundation.

Developers interested in introducing changes to the Linux kernel submit their changes to the relevant mailing list in units called *patches*. The developers on the mailing list respond with feedback on a patch, and the patch goes through a cycle of enhancements and feedback. Once a patch is ready to be integrated into the kernel, a Linux maintainer who is responsible for one section of the kernel signs off on the patch and forwards it to Linus Torvalds, who is also a Linux Foundation fellow, for final approval. If approved, the patch is integrated into the next release of the Linux kernel. A new major release of the kernel is made available approximately every three months.

When Linux was released in March 1994, the kernel consisted of just 176,250 lines of code. At the time of writing this book, version 5.0 of the Linux kernel consists of more than 25 million lines of code.

Linux Architecture

A detailed discussion of the Linux OS architecture is beyond the scope of this book. However, this section describes a few of the important characteristics of the different Linux OS components.

Figure 2-1 provides an architectural block diagram of Linux. It shows that applications are the top layer, presenting the software interface through which the user interacts with the device, hardware is at the bottom, and the kernel is in between.

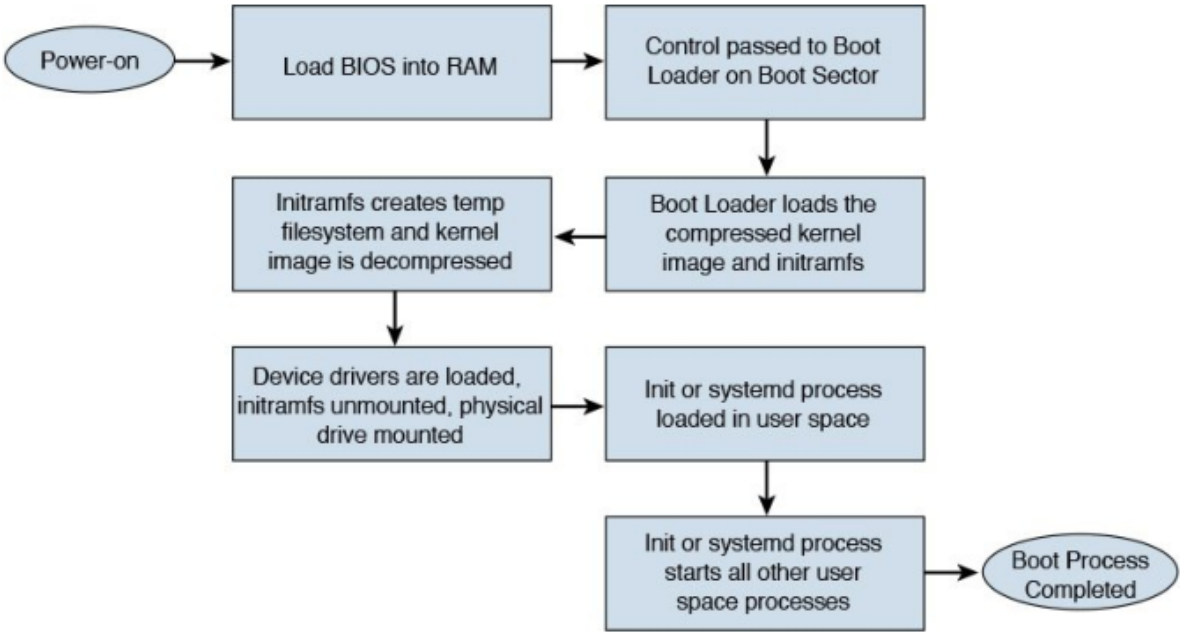


Figure 2-3 *The Linux OS Boot Process*

When you press the power-on button of your computer, system software, or *firmware*, saved on non-volatile flash memory on the computer's motherboard is run in order to initialize the computer's hardware and do *power-on self-tests* (*POST*) to confirm that the hardware is functioning properly. This firmware is called the *BIOS*, which stands for *basic input/output system*. After the hardware is initialized and the POST completed, based on the boot order that is set in the BIOS configuration, the BIOS starts searching for a *boot sector* on each of the drives listed in the configuration, in the order configured. The boot sector comes in several types, based on the drive type you are booting from. However, the BIOS has no understanding of the kind of boot sector it is accessing or the partitioning of the drive on which the boot sector resides. All it knows is that the boot sector is a *bootable* sector (because of the *boot sector signature* in its last 2 bytes), and it passes control to whatever software resides there (in this case, the *boot loader*). A *master boot record* (*MBR*) is a special type of boot sector that resides before the first partition and not on any one partition.

The boot loader then assumes control. The boot loader's primary function is to load the kernel image into memory and pass control to it in order to proceed with the rest of the boot process. A boot loader can also be configured to present the user with options in multi-boot environments, where the loader prompts the user to choose which of several different operating systems to boot. There are several boot loaders available, such as LILO, GRUB, and SYSLINUX, and the choice of which one to use depends on what needs to be achieved. Boot loaders can work in one or more stages, where the first stage is usually OS independent and the later stages are OS specific. Different boot loaders can also be *chain-loaded* (by configuration), one after the other, depending on what you (or the software implementation) need to do.

The boot loader searches for a kernel image to load based on the boot loader's configuration and, possibly, user input. Once the correct kernel image is identified, it is loaded into memory in compressed state. The boot loader also loads an *initial RAM disk* function called **initrd** or **initramfs**, which is a software image that provides a temporary file system in memory and allows the kernel to decompress and create a root file system without mounting any physical storage devices. (This is discussed further in the next section.) The kernel then decompresses in RAM and loads hardware device drivers as loadable kernel modules. Then **initrd** or **initramfs** is unmounted, and the physical drive is mounted instead.

Recall from earlier in this chapter that the Linux software components are classified as kernel space programs or user space programs. Up to this this point, no user space programs have run. The first user space program to run is the *parent process*, which is the **/usr/sbin/init** process or the **/lib/systemd/systemd** process in some systems. All other user space processes or programs are invoked by the **init** (or **systemd**) process.

Based on which components you chose to install, the **init** process starts a command shell and, optionally, a graphical user interface. At this point, you are prompted to enter your username and password in order to log in to the system.

To switch from the GUI to the command shell and back on CentOS, you need to log in to the GUI that boots up by default and then press Alt+Ctrl+F2 (or any function key from F3 to F6). The GUI then switches to full command-line mode. To switch back to the GUI, press Alt+Ctrl+F1. CentOS starts five command-line terminals and one graphical user interface.

A Linux Command Shell Primer

An *interpreter* is a program that accepts commands written in a high-level language, such as Python, and coverts them into lower-level code, either to be executed directly by the hardware or to be passed on to another program (such as the Python virtual machine) for further processing. Similarly, a *command shell* is a program that accepts commands from the user, parses and validates those commands, one by one, and then interprets the commands into a lower-level language to be passed to the Linux kernel for execution. Of course, the Linux shell communications model is a little more involved than this. This section focuses on the user interface of the command shell.

But why use the command-line interface (CLI) when you can use the graphical user interface (GUI)? There is nothing wrong with the GUI, but whether you want to use the CLI or the GUI depends primarily on what you are trying to accomplish. This book is about network automation and programmability. You will never tap into the true power of automation that Linux provides without relying heavily on the CLI (aka the command shell), whose use is described throughout this chapter. The significant value that automation provides applies to repeatable tasks; the key word here is *repeatable*. Automation in essence involves breaking up a task into smaller, repeatable tasks and then applying automation to those tasks, and this is where the CLI comes into play. Chapter 4, "Linux Scripting," builds on the CLI commands covered in this chapter and shows how to use Linux scripting to automate repeatable tasks, among other things.

There are numerous shells available today, some of which are platform independent and others that are available for particular operating systems only. Some shells are GPL licensed, and others are not. The shell covered here is the *Bash shell*, where Bash stands for *Bourne-*

again shell. Bash is a UNIX shell and command language written by Brian Fox for the GNU Project as a free software replacement for the Bourne shell, and it is the default shell on the vast majority of Linux distros in active development today.

To get started with Bash, log in to a CentOS machine and start the Terminal program, which is the interface to the Bash shell. You can start Terminal in several ways; the most straightforward method on CentOS 8 is to press on **Activities** at the top left corner of the screen. A search window will appear. Type terminal and press on the icon for Terminal that appears right under the search text box.

If you created the user NetProg during the installation and have logged in as that user, you should see a prompt similar to the one in Example 2-1.

Example 2-1 *The Terminal Program Prompt*

```
[NetProg@localhost ~]$
```

Throughout this chapter, the Terminal program window will be referred to as Terminal, the terminal, the Bash shell, the command-line shell or just the shell, interchangeably. The command prompt in Terminal is a great source of information. The username of the current user is shown first. In this case, it is user NetProg. Then, after the @ comes the computer (host) name, which in this case is the default localhost. Next comes the ~ (tilde), which represents the home directory of the current user, NetProg. Each user in Linux has a home directory that is named after the user and is always located under the /home/ directory. In this case, this directory is /home/NetProg/. If you use the **pwd** command, which stands for *print working directory*, the shell prints out the current working directory, which in this case is /home/NetProg/, as you can see in Example 2-2. This is referred to as the *working directory*.

Example 2-2 *Using the* ***pwd*** *Command*

```
[NetProg@localhost ~]$ pwd

/home/NetProg
```

The last piece of information that you can extract from the prompt is the fact that this is not user root, signified by the $ sign at the end of the prompt line. Example 2-3 introduces the command **su**, which stands for *switch user*. When you type **su** and press Enter, you are prompted for the root password that you set during the CentOS installation. Notice that the prompt changes to a # when you switch to user root.

Example 2-3 *Using the* ***su*** *Command*

```
[NetProg@localhost ~]$ su

Password:

[root@localhost NetProg]# pwd

/home/NetProg
```

The basic syntax for the **su** command is **su** {*username*}. When no username is specified in the command, it defaults to user root. Notice also in Example 2-3 that while the current user changed to root, the current directory is not the home directory of user root. In fact, it is the home directory of user NetProg, as shown in the **pwd** command output in Example 2-3. To switch to user root as well as the home directory for root, you use the **su** command with the **-** option, as shown in Example 2-4.

Example 2-4 *Using the* ***su -*** *Command*

```
[NetProg@localhost ~]$ su -

Password:

[root@localhost ~]# pwd

/root
```

If a user wants to run a command that requires root privileges, the user has two options. The first is to use the **su -** command to switch to the root user, and then execute the command as root. The second option is to use the **sudo** utility using the syntax **sudo** {*command*}. The sudo utility is used to execute a command as a superuser, granted that the user invoking the **sudo** command is authorized to do so. In other words, the user invoking the **sudo** command should be a member of the superusers group on the system, more formally known as the *sudoers* group. When the sudo utility is invoked, the invoking user is checked against the sudoers group, and if she is a member, the user is prompted to enter her password. If the authorization is successful, the command that requires root privileges is executed. More on users and groups in Chapter 3.

Whenever you need to clear the terminal screen, you use the command **clear**. This command clears the current terminal screen and all of the scroll back buffer except for one screen length of buffer history.

When you press the up arrow once at the terminal prompt, the last command you entered is recalled. Pressing the up arrow once more recalls the command before that. Each time you press the arrow key, one older command is recalled. To see a list of your previously entered commands, type the command **history**, which lists, by default, the last 1000 commands you entered. The number of previously entered commands that can be retained is configurable. If you are using the Bash shell, the history is maintained in the ~/.bash_history file.

Finding Help in Linux

Before proceeding any further, let's look at how to find help in Linux. Covering every option and argument of every command in Linux in this single chapter would simply not be possible. However, Linux provides an easy way to get help that enables you to further investigate and experiment with the commands covered in the subsequent sections and chapters so you can expand your knowledge beyond what is covered here. Linux has built-in documentation for virtually every Linux command and feature. It makes comprehensive information readily available to Linux users.

The simplest way to get help for a command is by using the **--help** option, or the shorter version **-h**, right after the command. Example 2-5 shows the help provided for the command **ls**, which stands for *list*. As stated in the help output, this command is used to "list information about the FILEs (the current directory by default)." As you can see, the output from the command **help** is quite detailed. The output in Example 2-5 has

been truncated for brevity. Don't worry if some or most of this output does not make much sense to you at this point.

Example 2-5 *Help for the* ***ls*** *Command*

```
[NetProg@localhost ~]$ ls --help

Usage: ls [OPTION]... [FILE]...

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

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


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

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

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

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

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

      --block-size=SIZE      scale sizes by SIZE before printing them; e.g.,

                               '--block-size=M' prints sizes in units of

                               1,048,576 bytes; see SIZE format below

  -B, --ignore-backups       do not list implied entries ending with ~


--------- OUTPUT TRUNCATED FOR BREVITY ------
```

To illustrate the output of the command **ls** and how the **help** output from Example 2-5 can be put to good use, Example 2-6 shows the output of the command when entered while in the home directory of user NetProg. Three different variations of arguments are used. The first is plain vanilla **ls**. The second is **ls -l**, which (as you can see in the help output) forces **ls** to use a long listing format. The final variation is **ls -la**, which (as you can see in the help output) tells **ls** to not ignore entries starting with a period, which are hidden files and directories; this argument basically tells **ls** to list all files and directories, including hidden ones.

Example 2-6 *Using Three Different Variations of the* ***ls*** *Command*

```
[NetProg@localhost ~]$ ls

Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos


[NetProg@localhost ~]$ ls -l

total 32

drwxr-xr-x. 2 NetProg NetProg 4096 Feb 13 04:48 Desktop

drwxr-xr-x. 2 NetProg NetProg 4096 Feb 13 04:48 Documents

drwxr-xr-x. 2 NetProg NetProg 4096 Feb 13 04:48 Downloads

drwxr-xr-x. 2 NetProg NetProg 4096 Feb 13 04:48 Music

drwxr-xr-x. 2 NetProg NetProg 4096 Feb 13 04:48 Pictures

drwxr-xr-x. 2 NetProg NetProg 4096 Feb 13 04:48 Public

drwxr-xr-x. 2 NetProg NetProg 4096 Feb 13 04:48 Templates

drwxr-xr-x. 2 NetProg NetProg 4096 Feb 13 04:48 Videos


[NetProg@localhost ~]$ ls -la

total 80

drwx------. 14 NetProg NetProg 4096 Feb 13 04:48 .

drwxr-xr-x.  5 root    root    4096 Feb 13 04:07 ..

-rw-------.  1 NetProg NetProg    4 Feb 13 04:08 .bash_history

-rw-r--r--.  1 NetProg NetProg   18 Jan  4 12:45 .bash_logout

-rw-r--r--.  1 NetProg NetProg  193 Jan  4 12:45 .bash_profile

-rw-r--r--.  1 NetProg NetProg  231 Jan  4 12:45 .bashrc

drwx------.  9 NetProg NetProg 4096 Feb 13 04:48 .cache

drwxr-xr-x. 11 NetProg NetProg 4096 Feb 13 04:48 .config

drwxr-xr-x.  2 NetProg NetProg 4096 Feb 13 04:48 Desktop
```

```
drwxr-xr-x.  2 NetProg NetProg 4096 Feb 13 04:48 Documents
```

Again, don't worry if some of this output does not make sense to you. The **ls** command is covered in detail later in this chapter, in the section "File and Directory Operations."

The second way Linux provides help to users is through the *manual pages*, also known as the *man pages*. Man pages are documentation pages for Linux built-in commands and programs. Applications that are not built in also have the option to add their own man pages during installation. To access the man pages for a command, you enter **man** {*command*}. Example 2-7 shows the first man page for the **ls** command.

Example 2-7 *The First Man Page for the* ***ls*** *Command*

```
LS(1)                            User Commands                           LS(1)


NAME
       ls - list directory contents


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


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

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

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

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

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

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

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

       -B, --ignore-backups
              do not list implied entries ending with ~

 Manual page ls(1) line 1 (press h for help or q to quit)
```

You should use the down arrow or Enter key to scroll down through the man page one line at a time. You should press the spacebar or Page Down key to scroll down one page at a time. To scroll up, you should either use the up arrow key to scroll one line at a time or the Page Up key to scroll up one page at a time. To exit the man page, press q. To search through the man pages, type / followed by the search phrase you are looking for. What you type, including the /, appears at the bottom of the page. If you press Enter, the phrase you are looking for is highlighted throughout the man pages. The line containing the first search result appears at the top of the page. You should press the letter n to move forward through the search results or N (capital n) to move backward to previous search results. To get to the top of the man page, you should press the g key, and to go to the end of the man page, you should press G (capital g). Being able to jump to the start or end of a man page with a single keypress is handy when you're dealing with a man page that is thousands of lines long.

All available man pages on a Linux distro are classified into sections, and the number of sections depends on the distro you are using. CentOS has nine sections. Each section consists of the man pages for a different category of components of the Linux OS. In Example 2-7, notice the LS(1) on the first line of the output, on both the left and right sides. This indicates that this man page is for the command **ls**, and this is Section 1 of the man pages.

From the output of the **man man** command, which brings up the manual pages for the **man** command itself, you can see that the man pages are classified into the following sections:

- **Section 1**: Executable programs or shell commands
- **Section 2**: System calls (that is, functions provided by the kernel)
- **Section 3**: Library calls (that is, functions within program libraries)
- **Section 4**: Special files (usually found in /dev)
- **Section 5**: File formats and conventions, such as /etc/passwd
- **Section 6**: Games
- **Section 7**: Miscellaneous (including macro packages and conventions), such as man(7) and groff(7)
- **Section 8**: System administration commands (usually only for root)
- **Section 9**: Kernel routines (nonstandard)

Why are the man pages categorized into different sections? Consider this scenario: **tar** is both a command that executes the utility to archive files and also a file format for archived files. The man pages for the archiving utility are provided in Section 1 (executable programs or shell commands), while the man pages for the file format are provided in Section 5 (file formats and conventions). When you type **man tar**, you invoke the man pages for the **tar** utility under Section 1, by default. In order to invoke the man pages for the **tar** file format, you need to type **man 5 tar**. And to see all possible man pages for a specific phrase, you use the form **man -k** {*phrase*}, as shown in Example 2-8 for the phrase **tar**. Note that the phrase **tar** was enclosed in quotes with an **^** before and a **$** after **tar**. This is an example of putting regular expressions to good use to avoid getting results that you do not need. In this case, you are only looking for the phrase **tar** and not for words that start or end with **tar** or any other variation of **tar** such as words that contain **tar** in them. Regular expressions are discussed in detail in Chapter 4. For now, you just need to know that regular expressions make it possible to match on certain strings using special symbols, such as the **^** symbol, which represents the beginning of a line, and the **$** symbol, which represents the end of a line.

Example 2-8 *Man Pages in Different Sections for* ***tar***

```
[NetProg@localhost ~]$ man -k "^tar$"

tar (1)              - an archiving utility

tar (5)              - format of tape archive files
```

An interesting—and maybe more intuitive—alternative to the man pages is the GNU *info* documentation. The info pages are help pages similar to the man pages, but the info pages are more detailed, documentation-style (rather than command-reference-style) hypertext documents, named *nodes*. The hyperlinks in the info pages enhances the experience of a user looking for information or help. The GNU info files can be accessed using either the **info** or **pinfo** commands. You can pass a phrase to one of these commands as an argument, where the phrase is what you are looking for. Or you can just type the command with no argument and then search the output for what you are looking for by typing **/** and then the search phrase. You can navigate through the info files by using the up and down arrow keys. You can go to the next node by pressing the n key or to the previous node by pressing the p key. Experiment with the GNU info pages by trying to locate the help for the commands covered so far and comparing the info pages with the man pages.

Files and Directories in Linux

By now, you should be familiar with the Linux Bash shell prompt and know where to go to find help. This section takes a closer look at the Linux file system, files, and directories.

The Linux File System

A disk (or any other storage medium, for that matter) is organized into one or more partitions. A *partition* is simply a logical section or slice of a disk. Each partition is logically separated from the other. In order to start using a particular partition, you need to create a file system on that partition. A file system defines how data is stored and retrieved from a disk. It defines a block of related data that has a beginning and an end and, most importantly, a name by which the block of data is identified. This block of data is called a *file*. Files are further grouped into directories, and directories are grouped into other directories, creating a tree-like hierarchy. Among other things, a file system does the following:

- Defines the size of the *allocation unit*, which is the minimum amount of physical storage space that can be allocated to a file
- Manages the space allocation to files, which may be composed of discontiguous allocations, as files are created, deleted, or changed in size
- Defines how to map between files and directories
- Defines the naming schemes of files and how to map the names to the actual locations of the files on the physical medium
- Maintains the metadata for files and directories—that is, the information about those files or directories (for example, file size and time of last modification)

Linux supports several file systems, including ext2, ext3, ext4, XFS, Btrfs, JFS, and NTFS.

Linux organizes its files and directories following the *Filesystem Hierarchy Standard* (*FHS*), which was developed specifically for Linux. This

standard ensures that the different Linux distros follow the same guidelines when implementing their file system hierarchy so that application developers can develop software that is portable between distros and meet other needs for standardization. The detailed standard can be found at http://refspecs.linuxbase.org/fhs.shtml.

Figure 2-4 illustrates the very basic directory tree structure in Linux. This hierarchy starts at the root, represented by a **/**, and all other files and directories branch from this root directory.

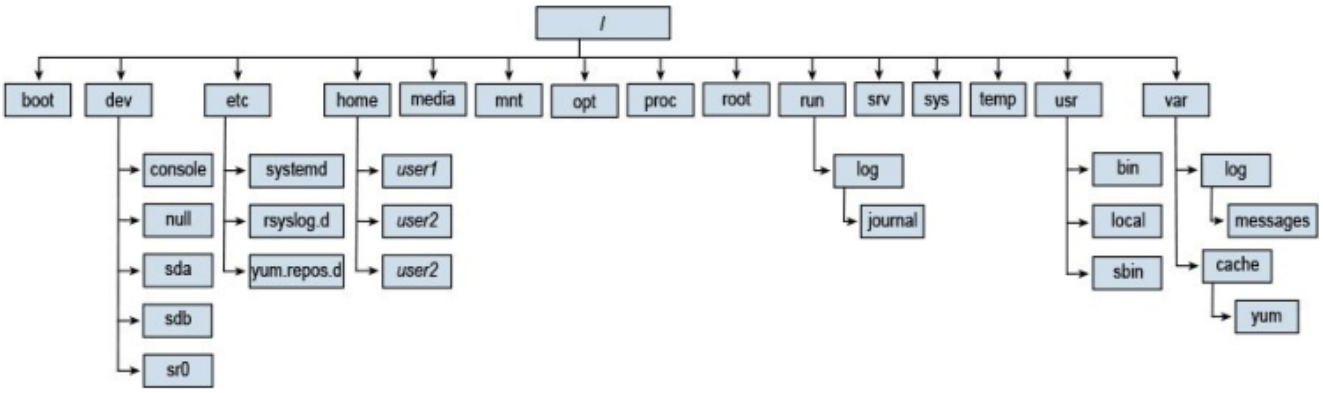


Figure 2-4 *The Linux Directory Tree Structure, Starting at the Root (/) Directory and Branching Out*

Everything in Linux is represented by a file somewhere in the file hierarchy. It is important that you become familiar with the Linux file system hierarchy and know which files to view or edit in order to get a particular task accomplished. Your knowledge will gradually increase as you progress through this chapter and Chapter 3. For now, the following is a high-level description of the main default directories on a CentOS Linux system:

• **/**: This is the root directory that is at the top of the file hierarchy and from which all other directories branch. This is not to be confused with the **/root** directory, which is the home directory of user root.

• **boot**: This directory contains the boot loader, kernel image, and initial RAM disk files.

• **dev**: This directory contains all the files required to access the devices. For example, when a hard disk is mounted on the system, the path to this disk is something similar to /dev/sda.

• **etc**: This directory contains the system configuration files and the configuration files of any application installed using the package management system of the distro being used (yum or dnf in the case of CentOS).

• **home**: Each user in Linux has a home directory that is named after the user's username. All home directories for all users reside under this home directory. A user's home directory contains all the subdirectories and files owned by this user. User NetProg's home directory, for example, is /home/NetProg.

• **media**: This directory contains subdirectories that are used as mount points for (temporary) *removable media* such as floppy disks and CD-ROMs.

• **mnt**: This directory is provided so that the system administrator can temporarily mount a file system, as needed.

• **opt**: This directory is reserved for the installation of add-on application software packages.

• **proc**: This directory is used for the storage and retrieval of process information as well as other kernel and memory information.

• **root**: This is the home directory of the user root that has superuser privileges. Note that this is not the root directory, which is the **/** directory.

• **run**: This directory contains files that are re-created each time the system reboots. The information in these files is about the running system and is as old as the last time the system was rebooted.

• **srv**: This directory contains site-specific data that is served by this system.

• **sys**: This directory contains information about devices, drivers, and some kernel features. Its underlying structure is determined by the particular Linux kernel being used.

• **tmp**: This directory contains temporary files that are used by users and applications. All the contents of this directory are flushed every configurable period of time (which is, by default, 10 days for CentOS).

• **usr**: This directory contains the files for installed applications. Application-shared libraries are also placed here. This directory contains the following subdirectories:

• **usr/bin**: This subdirectory contains the binary files for the commands that are used by any user on the system, such as **pwd**, **ls**, **cp**, and **mv**.

• **usr/sbin**: This subdirectory contains the command binary files for commands that may be executed by users of the system with administrator privileges.

• **usr/local**: This directory is used for the installed application files, similar to the Program Files directory in Windows.

• **var**: This directory contains files that are constantly changing in size, such as system log files.

• **.**: The dot is a representation of the current working directory. The value of **.** is equivalent to the output of the **pwd** command.

• **..**: The double-dot notation is a representation of the parent directory of the working directory. That is, the directory that is one level one level higher in the file system hierarchy than the working directory.

File and Directory Operations

This section introduces a number of commands for navigating, creating, deleting, copying, and viewing files and directories using the Linux command-line shell.

Navigating Directories

The command **cd** stands for *change directory* and is used to change the working directory from one directory to another. The syntax for **cd** is **cd** {*path*}, where *path* is the destination that you want to become your working directory. The *path* argument can be provided in one of two forms: either as a *relative path* or as an *absolute path*. Example 2-9 illustrates the use of both forms.

The relative path can be used when the destination directory is a subdirectory under the current working directory. In this case, the first part of the path (which is the absolute path to the current working directory) is implied. In Example 2-9, because the current working directory is /home/NetProg and you want to navigate to /home/NetProg/LinuxStudies, you can use the command **cd LinuxStudies**, where the first part of the path, /home/NetProg/, is implied because this is the current working directory. Obviously, the relative path does not work if you need to navigate to a directory that is not under your current working directory. In Example 2-9, for example, you could not navigate to /home/NetProg/Documents from /home/NetProg/LinuxStudies by simply entering **cd Documents**. In this case, the absolute path *must* be used.

Example 2-9 *Relative and Absolute Paths*

```
[NetProg@localhost ~]$ ls
Desktop    Downloads     Music     Public     Videos
Documents  LinuxStudies  Pictures  Templates


! Using the relative path to navigate to LinuxStudies
[NetProg@localhost ~]$ cd LinuxStudies
[NetProg@localhost LinuxStudies]$ pwd
/home/NetProg/LinuxStudies


! Now the relative path does not work when attempting to navigate to
/home/NetProg/Documents
[NetProg@localhost LinuxStudies]$ cd Documents
-bash: cd: Documents: No such file or directory


! Using the absolute path to navigate to Documents
[NetProg@localhost LinuxStudies]$ cd /home/NetProg/Documents
[NetProg@localhost Documents]$ pwd
/home/NetProg/Documents
```

At any point in your navigation, entering **cd** without any arguments takes you back to your home directory, characterized by the tilde (**~**) in the command prompt.

When you have navigated to the desired directory, you typically need to display its contents. The **ls** command stands for *list* and, as the name implies, **ls** is used to list the directory contents of the current working directory. When used without any options, it lists the files, side by side, without displaying any information apart from the file or subdirectory name. The **-a** option causes **ls** to display all files, including hidden files. The name of a hidden file starts with a dot (**.**). The **-l** option displays the files in a list format, along with the attributes of each file. The **-i** option adds the *inode number* to the displayed information. Example 2-10 displays the output of the **ls** command with all three options added, inside the home directory of user NetProg.

Example 2-10 ***ls*** *Command Output*

```
[NetProg@localhost ~]$ ls -lai
total 84
31719425 drwx------. 14 NetProg NetProg 4096 Feb 13 17:41 .
       2 drwxr-xr-x.  5 root    root    4096 Feb 13 04:07 ..
31719432 -rw-------.  1 NetProg NetProg  293 Feb 14 09:55 .bash_history
31719426 -rw-r--r--.  1 NetProg NetProg   18 Jan  4 12:45 .bash_logout
31719427 -rw-r--r--.  1 NetProg NetProg  193 Jan  4 12:45 .bash_profile
31719428 -rw-r--r--.  1 NetProg NetProg  231 Jan  4 12:45 .bashrc
31719433 drwx------.  9 NetProg NetProg 4096 Feb 13 04:48 .cache
31719434 drwxr-xr-x. 11 NetProg NetProg 4096 Feb 13 04:48 .config
```

```
31719485 drwxr-xr-x.  2 NetProg NetProg 4096 Feb 13 04:48 Desktop

31719489 drwxr-xr-x.  2 NetProg NetProg 4096 Feb 13 04:48 Documents

31719486 drwxr-xr-x.  2 NetProg NetProg 4096 Feb 13 04:48 Downloads


--------- OUTPUT TRUNCATED FOR BREVITY ------
```

A lot of information can be extracted from the output in Example 2-10. The phrase **total 84** indicates the total number of disk blocks allocated to store all the files in that directory. The second two lines of the output are for the current directory (**.**) and the directory one level above the current directory (**..**).

To elaborate on the use of the **.** and **..**, assume that the current working directory is /home/NetProg/LinuxStudies, and you want to navigate to /home/NetProg/Documents. You have two options: Either enter **cd /home/NetProg/Documents**, which is the absolute path, or use the shorthand notation **cd ../Documents**, where the **..** substitutes for /home/NetProg. You use the dot (**.**) notation similarly but for the current working directory. The value of using shorthand notation for the current working directory may not be immediately obvious, considering the availability of relative paths. However, by the end of Chapter 4 you will see how useful this notation is.

Next, all the files and subdirectories are listed; by default, they appear in alphabetical order. As noted earlier, the name of a hidden file or directory starts with a dot (**.**). The information from the beginning of each line all the way up to the file or directory name is collectively known as the *file* or *directory attributes*. Here is a description of each attribute:

• **Inode number**: The inode number is also called the file *serial number* or file *index number*. As per the **info** description for **ls**, the inode number is a number that "uniquely identifies each file within a particular file system."

• **File type**: This first character right after the inode number defines the file type. Three characters are commonly used in this field:

• **-** stands for a regular file.

• **d** stands for a directory.

• **l** stands for a soft link.

There are several other file types that are not discussed here.

• **File permissions**: Also called the file mode bits, the file permissions define who is allowed to read (**r**), write (**w**), or execute (**x**) the file. Users are classified into three categories: the owner of the file (**u**), the group of the file (**g**), and everyone else, or other (**o**). File permissions are covered in detail later in this chapter. For now, you need to know that the first three letters belong to the file owner, the second three belong to the group of the file, and the last three belong to everyone else. So, **rwxr-xr--** means that the file owner with permissions **rwx** can read, write, and execute the file. Users who are members of the same group as the file group, with permissions **r-x**, can read and execute the file but not write to it. Everyone else, with permissions **r--**, can only read the file but can neither write to it nor execute it. While the meaning of write, read, and execute are self-explanatory for files, they may not be so obvious for directories. Reading from a directory means listing its contents using the **ls** command, and writing to a directory means creating files or subdirectories under that directory. Executing directory X means changing the current working directory to directory X by using the **cd** command.

• **Alternate access method**: Notice the dot (**.**) right after the file permissions. This dot means that you have alternate means to set permissions for this file, such as using *access control lists* (*ACL*). ACLs are covered in detail in Chapter 3.

• **Number of hard links**: The number to the right of the file permissions is the total number of hard links to a file or to all files inside a directory. This is discussed in detail in section "Hard and Soft Links," later in this chapter.

• **File/directory owner**: This is the name of the file owner. In Example 2-10, it is NetProg.

• **File/directory group**: This is the file's group name. In Example 2-10, it is also NetProg. The file owner may or may not be part of this group. For example, the file owner could be NetProg, the group of the file could be Sales, and user NetProg may not be a member of the group Sales. Chapter 3 discusses how file access works in each case.

• **Size**: This is the file size, in bytes.

• **Time stamp**: This is the time when the file was last modified.

Viewing Files

In this section you will see how to display the contents of files on the terminal by using the commands **cat**, **more**, **less**, **head**, and **tail**.

The **cat** command, which stands for *concatenate*, writes out a file to *standard output* (that is, the screen). Example 2-11 shows how to use the **cat** command to display the output of the PIM.txt file.

Example 2-11 ***cat*** *Command Output*

```
[NetProg@localhost LinuxStudies]$ cat PIM.txt

!

Router-1#show ip pim neighbor

PIM Neighbor Table

Mode: B - Bidir Capable, DR - Designated Router, N - Default DR Priority,

      P - Proxy Capable, S - State Refresh Capable, G - GenID Capable
```

```
Neighbor          Interface               Uptime/Expires    Ver   DR
Address                                                           Prio/Mode
192.168.10.10     TenGigabitEthernet1/2   7w0d/00:01:26     v2    1 / G
192.168.20.20     TenGigabitEthernet2/1   2w2d/00:01:32     v2    1 / P G




PE-L3Agg-Mut-303-3#show ip pim interface


Address          Interface               Ver/   Nbr    Query  DR     DR
                                         Mode   Count  Intvl  Prior
192.168.10.11    TenGigabitEthernet1/2   v2/S   1      30     1      192.168.10.10
192.168.20.21    TenGigabitEthernet2/1   v2/S   1      30     1      192.168.20.20
!
[NetProg@localhost LinuxStudies]$
```

Several useful options can be used with **cat**. For example, **cat -n** inserts a line number at the beginning of each line. **cat -b**, on the other hand, inserts a line number for non-empty lines only. **cat -s** is the squeeze option, which squeezes more than one consecutive empty lines into a single empty line. Example 2-12 shows the output of the **cat** command on the file PIM.txt, using the **-sn** option to squeeze any consecutive empty lines in the file into one empty line and then number all lines, including the empty lines. For a more comprehensive list of options, you can visit the **cat info** page by using the command **info coreutils cat**.

Example 2-12 ***cat -sn*** *Command Output*

```
[NetProg@localhost LinuxStudies]$ cat -sn PIM.txt
     1    !
     2    Router-1#show ip pim neighbor
     3    PIM Neighbor Table
     4    Mode: B - Bidir Capable, DR - Designated Router, N - Default DR Priority,
     5          P - Proxy Capable, S - State Refresh Capable, G - GenID Capable
     6    Neighbor          Interface               Uptime/Expires    Ver   DR
     7    Address                                                           Prio/Mode
     8    192.168.10.10     TenGigabitEthernet1/2   7w0d/00:01:26     v2    1 / G
     9    192.168.20.20     TenGigabitEthernet2/1   2w2d/00:01:32     v2    1 / P G
    10
    11    PE-L3Agg-Mut-303-3#show ip pim interface
    12
    13    Address          Interface               Ver/   Nbr    Query  DR     DR
    14                                             Mode   Count  Intvl  Prior
    15    192.168.10.11    TenGigabitEthernet1/2   v2/S   1      30     1      192.168.10.10
    16    192.168.20.21    TenGigabitEthernet2/1   v2/S   1      30     1      192.168.20.20
    17    !
[NetProg@localhost LinuxStudies]$
```

One of the major drawbacks of **cat** is that the file being displayed is output to the screen all at once, without a pause. The next two commands, **more** and **less**, provide a more readable form of output, where just one section of the file is displayed on the screen, and then the user is prompted for input in order to proceed with the following section of the file, and so forth. Therefore, both of these commands are handy tools for displaying files that are longer than the current screen length. **more** is the original utility and is very compact, so it is ideal for systems with limited resources. However, the major drawback of **more** is that it does not allow you to move backward in a file; you can only move forward. Therefore, the **less** utility was eventually developed to allow users to move forward and backward over the content of a file. Over time, several developers contributed to the **less** program, adding more features in the process. One other distinctive feature of **less** is that it does not have to read the whole file before it starts displaying output; it is therefore much faster than many other programs, including the prominent **vi** text editor.

Example 2-13 shows the **more** command being used to display the contents of the file InternetRoutes.txt.

Example 2-13 ***more*** *Command Output*

```
[NetProg@localhost LinuxStudies]$ more InternetRoutes.txt

Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP

       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area

       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2

       E1 - OSPF external type 1, E2 - OSPF external type 2

       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2

       ia - IS-IS inter area, * - candidate default, U - per-user static route

       o - ODR, P - periodic downloaded static route, H - NHRP

       + - replicated route, % - next hop override


Gateway of last resort is 67.16.148.37 to network 0.0.0.0


S*    0.0.0.0/0 [1/0] via 67.16.148.37

      1.0.0.0/8 is variably subnetted, 2511 subnets, 16 masks

B        1.0.4.0/22 [200/100] via 67.16.148.37, 6d02h

B        1.0.4.0/24 [200/100] via 67.16.148.37, 6d02h

B        1.0.5.0/24 [200/100] via 67.16.148.37, 6d02h

B        1.0.6.0/24 [200/100] via 67.16.148.37, 6d02h

B        1.0.7.0/24 [200/100] via 67.16.148.37, 6d02h

B        1.0.16.0/24 [200/150] via 67.16.148.37, 7w0d

--More--(0%)
```

Notice the text --More--(0%) at the end of the output, which indicates how much of the file has been displayed so far. You can perform the following operations while viewing a file by using **more**:

• In order to keep scrolling down the file contents, press the Enter key to scroll one line at a time or press the Spacebar to scroll one screenful at a time.

• Type a number and press s to skip that number of lines forward in the file.

• Similarly, type a number and then press f to skip forward that number of screens.

• If you press =, the line number where you are currently located is displayed in place of the percentage at the bottom of the screen. This is the line number of the last line of the output at the bottom of the screen.

• To search for a specific pattern using regular expressions, type */{pattern}* and press Enter. The output jumps to the first occurrence of the pattern you are searching for.

• To quit the output and return to the terminal prompt, press q.

Now let's look at an example of using the **less** command. Example 2-14 shows the contents of the InternetRoutes.txt file displayed using **less**. Notice the filename at the end of the output.

Example 2-14 ***less*** *Command Output*

```
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP

       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area

       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2

       E1 - OSPF external type 1, E2 - OSPF external type 2

       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2

       ia - IS-IS inter area, * - candidate default, U - per-user static route

       o - ODR, P - periodic downloaded static route, H - NHRP

       + - replicated route, % - next hop override


Gateway of last resort is 67.16.148.37 to network 0.0.0.0


S*    0.0.0.0/0 [1/0] via 67.16.148.37
```

```
      1.0.0.0/8 is variably subnetted, 2511 subnets, 16 masks
B        1.0.4.0/22 [200/100] via 67.16.148.37, 6d02h
B        1.0.4.0/24 [200/100] via 67.16.148.37, 6d02h
B        1.0.5.0/24 [200/100] via 67.16.148.37, 6d02h
B        1.0.6.0/24 [200/100] via 67.16.148.37, 6d02h
B        1.0.7.0/24 [200/100] via 67.16.148.37, 6d02h
B        1.0.16.0/24 [200/150] via 67.16.148.37, 7w0d
InternetRoutes.txt
```

The following are some operations you can perform while viewing the file by using **less**:

• Use Enter, e, or j to scroll forward through the file one line at a time or use the Spacebar or z to scroll forward one screenful at a time.

• Press y to scroll backward one line at a time or b to scroll backward one screen at a time. Type a number before the y or b to scroll that many lines or screens, respectively.

• Press g to go to the beginning of the file or G to go to the end of the file.

• Press = to see the filename and the range of line numbers currently displayed on the screen, out of the total number of lines in the file, partial and full data size information, as well as your location in the file as a percentage.

• To search for a specific pattern using regular expressions, type */{pattern}* and press Enter. The output jumps to the first occurrence of the pattern you are searching for.

• To quit the output and return to the terminal prompt, press q.

For a complete list of operations you can perform while viewing the file by using **less**, visit the man or info pages for the **less** command.

It is generally recommended to use **less** instead of **more** because the latter is not under current development right now. Keep in mind, however, that you might run into systems with limited resources that support only **more**.

The final two commands covered in this section are **head** and **tail**. As their names may imply, these simple commands or utilities print a set number of lines from the start of the file or from the end of the file. Example 2-15 shows both commands being used to display selected output from the start or end of the InternetRoutes.txt file.

Example 2-15 ***head*** *and* ***tail*** *Command Output*

```
! displaying the first 15 lines of the file
[NetProg@localhost LinuxStudies]$ head -n 15 InternetRoutes.txt
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static route
       o - ODR, P - periodic downloaded static route, H - NHRP
       + - replicated route, % - next hop override

Gateway of last resort is 67.16.148.37 to network 0.0.0.0

S*    0.0.0.0/0 [1/0] via 67.16.148.37
      1.0.0.0/8 is variably subnetted, 2511 subnets, 16 masks
B        1.0.4.0/22 [200/100] via 67.16.148.37, 6d02h

! displaying the last 10 lines of the file
[NetProg@localhost LinuxStudies]$ tail -n 10 InternetRoutes.txt
B        110.204.0.0/17 [200/100] via 67.16.148.37, 6d02h
B        110.204.128.0/17 [200/100] via 67.16.148.37, 6d02h
B        110.205.0.0/16 [200/100] via 67.16.148.37, 6d02h
B        110.205.0.0/17 [200/100] via 67.16.148.37, 6d02h
B        110.205.128.0/17 [200/100] via 67.16.148.37, 6d02h
```

```
B        110.206.0.0/17 [200/100] via 67.16.148.37, 6d02h

B        110.206.128.0/17 [200/100] via 67.16.148.37, 6d02h


Connection closed by foreign host.

[NetProg@localhost LinuxStudies]$
```

The first section of the output in Example 2-15 shows how to extract the first 15 lines of the file by using **head**, and the second section of the output shows how to display the last 10 lines of the same file by using **tail**. A very useful variation is to use the **head** command with a negative number, such as **-20**, after the **-n** option. When this form is used, it means that all of the file is to be displayed *except for* the last 20 lines. Instead of using number of lines, you can specify the first or last number of bytes of the file to be displayed (using **head** or **tail**, respectively) by replacing the option **-n** with **-c**. Finally, to keep a *live* view of a file, you can use the **tail** command with the **-f** option. With this option, if a new line is added to the file, it appears on the screen. This comes in handy when viewing log files that are expected to change, and these changes need to be monitored as they happen; this is a very common scenario when troubleshooting system incidents. To quit live mode, press Ctrl+c.

File Operations

This section covers the most common file operations: creating, copying, deleting, and moving files.

In Example 2-16, the directory LinuxStudies has three empty subdirectories under it. The example shows how to use the **touch** command to create a file and call it PolicyMap.txt. When you pass a filename to the **touch** command as an argument, that file is created if it does not already exist. If the file already exists, the access and modification time stamps of the file are changed to the time when the **touch** command was issued. You can see this file in the output of the **ls** command.

Example 2-16 *Creating a File by Using* ***touch***

```
[NetProg@localhost LinuxStudies]$ ls -l

total 12

drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:14 configurations

drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:19 operational

drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:06 temp


[NetProg@localhost LinuxStudies]$ touch PolicyMap.txt


[NetProg@localhost LinuxStudies]$ ls -l

total 12

drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:14 configurations

drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:19 operational

-rw-rw-r--. 1 NetProg NetProg    0 Feb 17 12:22 PolicyMap.txt

drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:06 temp

[NetProg@localhost LinuxStudies]$
```

Note three things in Example 2-16:

• The file created is empty and has a size of zero bytes.

• The file's time stamp is the time at which the **touch** command was issued.

• Linux is case sensitive. The files PolicyMap.txt and policymap.txt are two entirely different files. The same case sensitivity applies to commands.

Example 2-17 shows how to copy the file PolicyMap.txt to the operational directory by using the **cp** command. Because this is a copy operation, now both the LinuxStudies directory and the subdirectory operational contain copies of the file, as shown by using the **ls -l** command in each of the directories. Remember that the dot (.) and double dot (..) notations, combined with relative paths, are often used to refer to the current working directory and the parent directory, respectively. The file is then deleted from the LinuxStudies directory by using the **rm** command. Issuing the **ls** command again shows that the file was indeed deleted. Following that, the file is moved (not copied) with the **mv** command from the operational subdirectory to the configurations subdirectory. Issuing the **ls** command in both directories shows that the file was moved to the latter, and the former is empty now. Finally, the file is renamed PolicyMapConfig.txt: The **mv** command renames and moves the old file to the new one, in the same location. The **ls** command, issued one final time, confirms that the file renaming was successful.

Example 2-17 *File Operations: Copy, Delete, and Move*

```
! File copy operation

[NetProg@localhost LinuxStudies]$ cp PolicyMap.txt ./operational/

[NetProg@localhost LinuxStudies]$ ls -l

total 16
```

```
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:14 configurations
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:19 operational
-rw-rw-r--. 1 NetProg NetProg  0 Feb 17 12:24 PolicyMap.txt
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:06 temp
[NetProg@localhost LinuxStudies]$ ls -l ./operational/
total 4
-rw-rw-r--. 1 NetProg NetProg 361 Feb 17 12:24 PolicyMap.txt


! File delete operation
[NetProg@localhost LinuxStudies]$ rm PolicyMap.txt
[NetProg@localhost LinuxStudies]$ ls -l
total 12
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:14 configurations
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:19 operational
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:06 temp
[NetProg@localhost LinuxStudies]$ cd operational/
[NetProg@localhost operational]$ ls -l
total 4
-rw-rw-r--. 1 NetProg NetProg 361 Feb 17 12:24 PolicyMap.txt


! File move operation
[NetProg@localhost operational]$ mv PolicyMap.txt ../configurations/
[NetProg@localhost operational]$ ls -l
total 0
[NetProg@localhost operational]$ cd ../configurations/
[NetProg@localhost configurations]$ ls -l
total 4
-rw-rw-r--. 1 NetProg NetProg 361 Feb 17 12:24 PolicyMap.txt


! Renaming a file by moving it to the same location but with a different name
[NetProg@localhost configurations]$ mv PolicyMap.txt PolicyMapConfig.txt
[NetProg@localhost configurations]$ ls -l
total 4
-rw-rw-r--. 1 NetProg NetProg 361 Feb 17 12:24 PolicyMapConfig.txt
[NetProg@localhost configurations]$
```

A more secure alternative to the command **rm** is the command **shred**, which overwrites the file a configurable number of time (three by default) in order to eliminate the possibility of recovering the deleted file even via direct hardware access.

The following is a summary of the commands used for file operations:

• **touch** {*file_name*}: Use this syntax to create a file.

• **cp** {*source*} {*destination*}: Use this syntax to copy a file to another location.

• **mv** {*source*} {*destination*}: Use this syntax to move a file from one location to another:

• **mv** {*old_file_name*} {*new_file_name*}: Use this syntax to rename a file. The old and new files could be collocated in the same directory or located in different directories.

• **rm** {*file_name*}: Use this syntax to delete a file.

• **shred** {*file_name*}: Use this syntax to securely delete a file.

When operating on files, it is important to be careful about what the current directory is, what the destination directory is, and where the file currently resides. When using the commands listed here, you need to use absolute paths, relative paths, or no path at all—whichever is

applicable at the time. Also, notice the use of the shorthand dot and double dot notations in the previous examples and how they make a command line both shorter and easier.

Directory Operations

This section discusses directory operations. Some of the commands in this section are the same as those used with files, but they have added options for directories. Some of the commands in this section are exclusive to directories.

The next few examples demonstrate how to create a directory, copy it to a new location, move it to a new location, and rename it. The examples also show what you need to do to delete a directory in two different cases: either the directory is empty or it is not.

Example 2-18 shows two directories being created under the LinuxStudies directory with the **mkdir** command: EmptyDir and NonEmptyDir. By using the **cp** command, the file PolictMapConfig.txt is then copied to the directory NonEmptyDir. One directory is now empty, and the other directory contains one file.

Example 2-18 *Creating Directories*

```
[NetProg@localhost LinuxStudies]$ ls

configurations  operational  temp

[NetProg@localhost LinuxStudies]$ mkdir EmptyDir

[NetProg@localhost LinuxStudies]$ mkdir NonEmptyDir

[NetProg@localhost LinuxStudies]$ ls

configurations  EmptyDir  NonEmptyDir  operational  temp

[NetProg@localhost LinuxStudies]$
```

The next example shows a different hierarchy: A new directory is created, and the two directories created in the previous example are moved into it. In Example 2-19, a new directory called MasterDir is created using the **mkdir** command and then the **mv** command is used to move both directories under the newly created MasterDir directory. The output of the **ls** command shows that both directories were successfully moved to the new location.

Example 2-19 *Moving Directories*

```
[NetProg@localhost LinuxStudies]$ mkdir MasterDir

[NetProg@localhost LinuxStudies]$ ls

configurations  EmptyDir  MasterDir  NonEmptyDir  operational  temp

[NetProg@localhost LinuxStudies]$ mv EmptyDir MasterDir

[NetProg@localhost LinuxStudies]$ mv NonEmptyDir MasterDir

[NetProg@localhost LinuxStudies]$ ls MasterDir

EmptyDir  NonEmptyDir

[NetProg@localhost LinuxStudies]$
```

Example 2-20 shows a new directory called MasterDirReplica being created. The **cp** command is then used in an attempt to copy both EmptyDir and NonEmptyDir to the new directory. As shown in the example, the operation fails; the error message indicates that when copying directories, you need to issue the **cp** command with the **-r** option, which stands for *recursive*. When the **cp** command is issued with the correct option, the copy operation is successful, as indicated by the output of the **ls** command. Notice that the **-r** option needs to be added to the **cp** command, whether the directory is empty or not.

Example 2-20 *Copying Directories*

```
[NetProg@localhost LinuxStudies]$ mkdir MasterDirReplica

[NetProg@localhost LinuxStudies]$ ls

configurations  MasterDir  MasterDirReplica  operational  temp

[NetProg@localhost LinuxStudies]$ cp MasterDir/EmptyDir MasterDirReplica

cp: -r not specified; omitting directory 'MasterDir/EmptyDir'

[NetProg@localhost LinuxStudies]$ cp MasterDir/NonEmptyDir MasterDirReplica

cp: -r not specified; omitting directory 'MasterDir/NonEmptyDir'

[NetProg@localhost LinuxStudies]$ cp -r MasterDir/EmptyDir MasterDirReplica

[NetProg@localhost LinuxStudies]$ cp -r MasterDir/NonEmptyDir MasterDirReplica

[NetProg@localhost LinuxStudies]$ ls MasterDirReplica/

EmptyDir  NonEmptyDir

[NetProg@localhost LinuxStudies]$
```

The command to delete an *empty* directory in Linux is **rmdir**. For historical reasons, **rmdir** works only for empty directories, and the command **rm -r** is required to delete *non-empty* directories. In Example 2-21, an attempt is made to delete both the EmptyDir and NonEmptyDir

directories by using the rmdir command, but as expected, it does not work on the directory NonEmptyDir. Using **rm -r** works fine, and the final **ls** command shows that.

Example 2-21 *Deleting Directories*

```
[NetProg@localhost LinuxStudies]$ ls MasterDir

EmptyDir  NonEmptyDir

[NetProg@localhost LinuxStudies]$ rmdir MasterDir/EmptyDir

[NetProg@localhost LinuxStudies]$ ls MasterDir

NonEmptyDir

[NetProg@localhost LinuxStudies]$ rmdir MasterDir/NonEmptyDir

rmdir: failed to remove 'MasterDir/NonEmptyDir': Directory not empty

[NetProg@localhost LinuxStudies]$ ls MasterDir

NonEmptyDir

[NetProg@localhost LinuxStudies]$ rm -r MasterDir/NonEmptyDir

[NetProg@localhost LinuxStudies]$ ls MasterDir

[NetProg@localhost LinuxStudies]$
```

Finally, Example 2-22 shows the use of the **mv** command to rename the directory NonEmptyDir to NonEmptyDirRenamed.

Example 2-22 *Renaming Directories*

```
[NetProg@localhost MasterDirReplica]$ ls

EmptyDir  NonEmptyDir

[NetProg@localhost MasterDirReplica]$ mv NonEmptyDir NonEmptyDirRenamed

[NetProg@localhost MasterDirReplica]$ ls

EmptyDir  NonEmptyDirRenamed
```

The following is a summary of the commands used for directory operations:

- **mkdir** {*directory_name*}: Use this syntax to create directories.
- **cp -r** {*source*} {*destination*}: Use this syntax to copy directories to another location.
- **mv** {*source*} {*destination*}: Use this syntax to move directories from one location to another.
- **mv** {*old_dir_name*} {*new_dir_name*}: Use this syntax to rename directories. A renamed directory could be collocated (in the same path) with the original directory, or it could be in a different path (moved and renamed in the same operation).
- **rmdir** {*directory_name*}: Use this syntax to delete empty directories.
- **rm -r** {*directory_name*}: Use this syntax to delete non-empty directories.

Hard and Soft Links

Linux provides the facility to create a link from one file to another file. A *link* is basically a relationship between two files. This relationship means that changes to one file affect the linked file in one way or another. There are two types of links in Linux: hard links and soft, or symbolic, links. You create links in Linux by using the **ln** command. A link is created between the original file, referred to as the *target*, and a newly created file, referred to as the *link*.

Hard Links

You create a hard link between a target and a link by using the syntax (in its simplest form) **ln** {*target-file*} {*link-file*}. A hard link is characterized by the following:

- Any changes to the contents of the target file are reflected in the link file and vice versa.
- Any changes to the target file attributes, such as the file permissions, are reflected in the link file and vice versa.
- Deleting the target file does *not* delete the link file.
- A target file can have one or more link files linked to it. The target and all its hard links have the same inode number.
- Hard links are allowed for files only, not for directories.

Example 2-23 shows a hard link named HL-1-to-Access-List.txt created to the file Access-List.txt. The command **ls -li** is used to list the files in the directory LinuxStudies, including the file attributes. Notice that apart from the different name, the hard link file is basically a replica of the target: Both have the same file size, permissions, and inode number. Then a second hard link, named HL-2-to-Access-List.txt, is created. Notice the number 1 to the right of the file permissions of the original target, Access-List.txt, before any hard links are created. This number increments by 1 every time a hard link is created.

Then the first hard link is deleted. Notice that the target file and the second hard link stay intact. The target file is deleted, and the second hard link stays intact. As mentioned previously, deleting the target or one of the hard links does not affect the other hard links.

Example 2-23 *Creating Hard Links*

```
[NetProg@localhost ~]$ cd LinuxStudies

[NetProg@localhost LinuxStudies]$ ls -li

total 2304

57934070 -rw-r--r--. 1 NetProg NetProg     470 Feb 14 10:08 Access-List.txt

57934069 -rw-r--r--. 2 NetProg NetProg 2353097 Feb 14 10:09 showrun.txt


! Create the first hardlink

[NetProg@localhost LinuxStudies]$ ln Access-List.txt HL-1-to-Access-List.txt

[NetProg@localhost LinuxStudies]$ ls -li

total 2308

57934070 -rw-r--r--. 2 NetProg NetProg     470 Feb 14 10:08 Access-List.txt

57934070 -rw-r--r--. 2 NetProg NetProg     470 Feb 14 10:08 HL-1-to-Access-List.txt

57934069 -rw-r--r--. 2 NetProg NetProg 2353097 Feb 14 10:09 showrun.txt


! Create the second hard link

[NetProg@localhost LinuxStudies]$ ln Access-List.txt HL-2-to-Access-List.txt

[NetProg@localhost LinuxStudies]$ ls -li

total 2312

57934070 -rw-r--r--. 3 NetProg NetProg     470 Feb 14 10:08 Access-List.txt

57934070 -rw-r--r--. 3 NetProg NetProg     470 Feb 14 10:08 HL-1-to-Access-List.txt

57934070 -rw-r--r--. 3 NetProg NetProg     470 Feb 14 10:08 HL-2-to-Access-List.txt

57934069 -rw-r--r--. 2 NetProg NetProg 2353097 Feb 14 10:09 showrun.txt


! Remove the first hard link - target and second hard link stay intact

[NetProg@localhost LinuxStudies]$ rm HL-1-to-Access-List.txt

[NetProg@localhost LinuxStudies]$ ls -li

total 2308

57934070 -rw-r--r--. 2 NetProg NetProg     470 Feb 14 10:08 Access-List.txt

57934070 -rw-r--r--. 2 NetProg NetProg     470 Feb 14 10:08 HL-2-to-Access-List.txt

57934069 -rw-r--r--. 2 NetProg NetProg 2353097 Feb 14 10:09 showrun.txt


! Delete the target - second hard link stays intact

[NetProg@localhost LinuxStudies]$ rm Access-List.txt

[NetProg@localhost LinuxStudies]$ ls -li

total 2304

57934070 -rw-r--r--. 1 NetProg NetProg     470 Feb 14 10:08 HL-2-to-Access-List.txt

57934069 -rw-r--r--. 2 NetProg NetProg 2353097 Feb 14 10:09 showrun.txt

[NetProg@localhost LinuxStudies]$
```

Example 2-24 shows how hard-linked files change together. Reverting to the original state where we have the file Access-List.txt and two hard links to it, using the command **chmod**, the permissions of the target are changed from **-rw-r--r--** to **-rw-rw-r--**. This command is covered in detail in Chapter 3. For now, notice the new permissions that change for the target as well as all the hard links. To take this a step further, the permissions are changed for the second hard link to **-rw-rw-rw-**. Notice now how the permissions change for the target as well as the other hard link.

Example 2-24 *HowAttributes Are Reflected Across Hard Links*

```
[NetProg@localhost LinuxStudies]$ ls -li

total 2312
```

```
57934070 -rw-r--r--. 3 NetProg NetProg     470 Feb 14 10:45 Access-List.txt
57934070 -rw-r--r--. 3 NetProg NetProg     470 Feb 14 10:45 HL-1-to-Access-List.txt
57934070 -rw-r--r--. 3 NetProg NetProg     470 Feb 14 10:45 HL-2-to-Access-List.txt
57934069 -rw-r--r--. 2 NetProg NetProg 2353097 Feb 14 10:09 showrun.txt


! Changing the file permissions for the target
[NetProg@localhost LinuxStudies]$ chmod g+w Access-List.txt
[NetProg@localhost LinuxStudies]$ ls -li
total 2312
57934070 -rw-rw-r--. 3 NetProg NetProg     470 Feb 14 10:45 Access-List.txt
57934070 -rw-rw-r--. 3 NetProg NetProg     470 Feb 14 10:45 HL-1-to-Access-List.txt
57934070 -rw-rw-r--. 3 NetProg NetProg     470 Feb 14 10:45 HL-2-to-Access-List.txt
57934069 -rw-r--r--. 2 NetProg NetProg 2353097 Feb 14 10:09 showrun.txt


! Changing the file permissions for the second hard link
[NetProg@localhost LinuxStudies]$ chmod o+w HL-2-to-Access-List.txt
[NetProg@localhost LinuxStudies]$ ls -li
total 2312
57934070 -rw-rw-rw-. 3 NetProg NetProg     470 Feb 14 10:45 Access-List.txt
57934070 -rw-rw-rw-. 3 NetProg NetProg     470 Feb 14 10:45 HL-1-to-Access-List.txt
57934070 -rw-rw-rw-. 3 NetProg NetProg     470 Feb 14 10:45 HL-2-to-Access-List.txt
57934069 -rw-r--r--. 2 NetProg NetProg 2353097 Feb 14 10:09 showrun.txt
[NetProg@localhost LinuxStudies]$
```

Similarly, content changes in one file are automatically reflected in the target and all other hard links, as shown in Example 2-25. The command **cat** displays the contents of file Access-List.txt on the terminal, and then **cat** is used again to display the contents of HL-1-to-Access-List.txt. The contents of the two files are, as expected, the same. Now the text editor **vi** is used to add a new line at the top of access list Test-Access-List with sequence number 5 in the file Access-List.txt. The file contents are then viewed using **cat** to confirm that the changes were successfully saved. Viewing the contents of both hard-linked files shows that the new line was also added to the ACL Test-Access-List in both files.

Example 2-25 *How Content Changes Are Reflected Across Hard Links*

```
[NetProg@localhost LinuxStudies]$ ls -li
total 2312
57934145 -rw-rw-r--. 3 NetProg NetProg     512 Feb 14 14:45 Access-List.txt
57934145 -rw-rw-r--. 3 NetProg NetProg     512 Feb 14 14:45 HL-1-to-Access-List.txt
57934145 -rw-rw-r--. 3 NetProg NetProg     512 Feb 14 14:45 HL-2-to-Access-List.txt
57934069 -rw-r--r--. 2 NetProg NetProg 2353097 Feb 14 10:09 showrun.txt


! Identical file content before editing
[NetProg@localhost LinuxStudies]$ cat Access-List.txt
!
ipv4 access-list Test-Access-List
 10 permit ipv4 192.168.10.0 0.0.0.255 any
 20 permit ipv4 192.168.20.0 0.0.3.255 any
 30 permit ipv4 192.168.30.0 0.0.0.255 any
!
[NetProg@localhost LinuxStudies]$ cat HL-1-to-Access-List.txt
!
ipv4 access-list Test-Access-List
 10 permit ipv4 192.168.10.0 0.0.0.255 any
 20 permit ipv4 192.168.20.0 0.0.3.255 any
```

```
 30 permit ipv4 192.168.30.0 0.0.0.255 any
!


! Content changes in target are automatically reflected in both hard links
[NetProg@localhost LinuxStudies]$ vi Access-List.txt
[NetProg@localhost LinuxStudies]$ cat Access-List.txt
!
ipv4 access-list Test-Access-List
 5 permit ipv4 192.168.10.0 0.0.0.255 any
 10 permit ipv4 192.168.10.0 0.0.0.255 any
 20 permit ipv4 192.168.20.0 0.0.3.255 any
 30 permit ipv4 192.168.30.0 0.0.0.255 any
!
[NetProg@localhost LinuxStudies]$ cat HL-1-to-Access-List.txt
!
ipv4 access-list Test-Access-List
 5 permit ipv4 192.168.10.0 0.0.0.255 any
 10 permit ipv4 192.168.10.0 0.0.0.255 any
 20 permit ipv4 192.168.20.0 0.0.3.255 any
 30 permit ipv4 192.168.30.0 0.0.0.255 any
!
[NetProg@localhost LinuxStudies]$ cat HL-2-to-Access-List.txt
!
ipv4 access-list Test-Access-List
 5 permit ipv4 192.168.10.0 0.0.0.255 any
 10 permit ipv4 192.168.10.0 0.0.0.255 any
 20 permit ipv4 192.168.20.0 0.0.3.255 any
 30 permit ipv4 192.168.30.0 0.0.0.255 any
!
[NetProg@localhost LinuxStudies]$
```

In very simple terms, a hard link creates a new *live* copy of a file that changes as the target or any of the other hard links change. More accurately, a hard link creates a new pointer to the same inode that has a different name (filename). The inode number is the same across all hard-linked files, and hard links cannot span different file systems because inode numbers may not be unique across different file systems on separate partitions.

One last thing to note in the output of the **ls -l** command is that each file by default has one hard link that must be present before you create any hard links to the file manually. This hard link is the original target file. This reinforces the concept that a hard link is just a pointer to the same inode number, and the first pointer to a particular inode number is the target itself.

One use case for hard links is the utility to distribute data. Consider a configuration file or a device inventory with one or more hard links, each being used by a different device or application. Every time the file or one of the hard links is updated by one of the applications or devices, the updates are automatically reflected to all the other files. Think of all the possibilities that this functionality provides in the real world of automation!

Soft Links

A soft link, commonly referred to as a symbolic link, or symlink for short, does not create a live copy of a target file as a hard link does. Instead, a symbolic link, as the name implies, is just a pointer, or a shortcut, to the original file, not the inode, as is the case with hard links. Symlinks are created using the command **ln -s** {*target_file*} {*link_file*}. Symlinks are characterized by the following:

• The target file and the link file have different inode numbers.

• Symlink file permissions are always rwxrwxrwx, regardless of the permissions of the target file.

• Symlinks have the letter **l** to the left of the file permissions in the output of the **ls -l** command and an arrow pointing to the target file at the end of the line of the same output.

• A symlink does not disappear when the target file is deleted. Instead, the output of the command **ls -l** shows the target file highlighted in red and flashing to indicate that the symlink is broken.

• The symlink references the target file by name. If the target file is replaced by any other file that has the same name, the symlink points to that new file.

• Unlike hard links, symlinks can be created for directories as well as files.

Example 2-26 shows symlinks at work. First, a symlink named SL-1-to-Access-List.txt is created for the file Access-List.txt. Notice the different inode numbers and file permissions between the target and link files. Notice also the **l** that is prepended to the file permissions of the soft link and the arrow pointing to the target at the end of the line; both the **l** and the arrow indicate that this is a soft link. The target is then deleted using the **rm** command. However, the soft link file still appears in the output of the **ls** command. On a computer screen, the target would also be highlighted in red to indicate a broken link. Next in the example, a new empty file is created using the **touch** command, but it has the same name as the file that was deleted, Access-List.txt. When the **ls** command is issued, it shows that the symlink is operational again, and it points to the newly created text file. To further confirm that the symlink is working, the **cat** command is issued, and it shows both files being empty. The **vi** editor is then used to add an ACL, Access-List-Test, to the symlink file, and after the **cat** command is issued for both files, it turns out that the changes made to the symlink have been reflected to the target, Access-List.txt.

Example 2-26 *Symlinks at Work*

```
[NetProg@localhost LinuxStudies]$ ln -s Access-List.txt SL-1-to-Access-List.txt

[NetProg@localhost LinuxStudies]$ ls -li

total 2312

57934145 -rw-rw-r--. 3 NetProg NetProg     512 Feb 14 14:45 Access-List.txt

57934143 lrwxrwxrwx. 1 NetProg NetProg      15 Feb 14 14:55 SL-1-to-Access-List.txt -> Access-List.txt


! Deleting the target does not delete the symlink

[NetProg@localhost LinuxStudies]$ rm Access-List.txt

[NetProg@localhost LinuxStudies]$ ls -li

total 2308

57934143 lrwxrwxrwx. 1 NetProg NetProg      15 Feb 14 14:55 SL-1-to-Access-List.txt -> Access-List.txt


! Creating a new file with the same name as the deleted target reinstates the symlink

[NetProg@localhost LinuxStudies]$ touch Access-List.txt

[NetProg@localhost LinuxStudies]$ ls -li

total 2308

57934146 -rw-rw-r--. 1 NetProg NetProg       0 Feb 14 14:58 Access-List.txt

57934143 lrwxrwxrwx. 1 NetProg NetProg      15 Feb 14 14:55 SL-1-to-Access-List.txt -> Access-List.txt


[NetProg@localhost LinuxStudies]$ cat Access-List.txt


[NetProg@localhost LinuxStudies]$ cat SL-1-to-Access-List.txt


[NetProg@localhost LinuxStudies]$ vi SL-1-to-Access-List.txt

[NetProg@localhost LinuxStudies]$ cat SL-1-to-Access-List.txt

!

ipv4 access-list Test-Access-List

 10 permit ipv4 192.168.10.0 0.0.0.255 any

 20 permit ipv4 192.168.20.0 0.0.0.255 any

 30 permit ipv4 192.168.30.0 0.0.0.255 any

!

[NetProg@localhost LinuxStudies]$ cat Access-List.txt

!

ipv4 access-list Test-Access-List

 10 permit ipv4 192.168.10.0 0.0.0.255 any

 20 permit ipv4 192.168.20.0 0.0.0.255 any

 30 permit ipv4 192.168.30.0 0.0.0.255 any
```

```
!

[NetProg@localhost LinuxStudies]$
```

Soft links provide similar functionality to Windows shortcuts. One use case for symlinks is to consolidate all your work in one directory. The directory contains symlinks to all files from other directories. Changes made to any symlink are reflected to the original file, and you do not have to move the original file from its place in the file system.

Input and Output Redirection

Earlier in this chapter, you briefly learned about the GNU utilities that are bundled with the Linux kernel to form the Linux operating system. Utilities are a collection of software tools that enable a user to perform common system tasks without having to write their own tool set.

All the commands introduced so far in this chapter (as well as in the remainder of this chapter) are actually utilities, and each is invoked via the respective command. For example, the **ls**, **cat**, **more**, **less**, **head**, **tail**, **cp**, **mv**, **rm**, **mkdir**, **rmdir**, and **ln** commands covered so far are actually utilities, and you run each utility by typing the corresponding command in the shell. Most utilities are grouped together in packages. When a package is installed, all constituent utilities are installed. Two popular packages are **coreutils** and **util-linux**.

The true power of Linux lies not only in its architecture but in the vast number of utilities that come prepackaged with it, new utilities that can be easily installed and immediately add to the power and usability of the system, and, finally, the option of programming your own custom utilities. Utilities are introduced in this section because input and output redirection are arguably two of the most powerful features of Linux that act on utilities. Redirection stretches the flexibility and usability of utilities and combines the workings of two or more utilities in ways unique to Linux, as you will see in this section.

The Linux and UNIX philosophy has been inspired by the experience of the software development leaders who developed programming languages. Ken Thomson and Dennis Ritchie developed the C language as well as UNIX. Ken and Dennis, among others, realized early on that the operating system should present an interface to the user that facilitates a productive and interactive experience. Mimicking programming languages, they wanted the user to be able to filter input/output of programs and apply control to the flow of standard input, output, and errors between these utilities.

The UNIX forefathers applied software engineering methods traditionally used in programming languages to their operating system user experience. These engineering methods are reflected in the powerful command-line utilities of both UNIX and Linux, along with pipes and redirection, to smoothly integrate tools.

The power of the UNIX and Linux command line is achieved with the following design philosophies:

• Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.

• Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information.

Thanks to these design philosophies for the command line, an administrator is immediately equipped with a powerful and infinitely flexible tool chain for all sorts of operations.

The community of Linux developers around the world is continuously contributing to the long list of available utilities, creating small blocks that can work together to produce powerful results, making it easy to automate mundane administrative tasks. A good way to demonstrate this power is by showing an advanced example that illustrates the full potential of utilities and pipes. Example 2-27 is a relatively complex example that pings the gateways in the Linux routing table and inserts the unreachable ones in a file. This file is then sent via email. In this example, the output of one command is piped to another using the | (pipe) symbol.

Note

You do not need to worry about the particular semantics of this example as its goal is to illustrate the sheer power of piping the output of one command to be used as input to another command.

Example 2-27 *A Relatively Complex Example of Piping*

```
[NetProg@localhost]$ netstat -nr | awk '{print $2}' | grep -o '[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}' |
xargs -n1 ping -c1  | grep  -b1 100 |  mail -s "Unreachable gateways" netprog@thenetdev.com
```

This section covers input and output redirection in detail. For now, here is a brief explanation of each command in Example 2-27:

• **netstat -nr**: Displays the routing table and pipes it to the next command, **awk**.

• **awk '{print $2}'**: Filters the second column only (gateways) from the output of the previous command (**netstat**), and pipes the result to the next command (**grep**).

• **grep -o '[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}'**: Only shows IP addresses from the output of the previous command (**awk**) and pipes the result to the next command (**xargs**).

• **xargs -n1 ping -c1**: Pings IP addresses that were provided by the previous command (**grep**) and pipes the ping results to the next command (**grep**).

• **grep -b1 100**: Filters the unreachable IP addresses from the ping performed in the previous command (**xargs**) and pipes the result to the next command (**mail**).

• **mail -s "Unreachable IPs" -t netprog@thenetdev.com**: Sends an email with the output of the previous command (**grep**) with the subject "Unreachable IPs" to user NetProg's email address.

In Linux, for each command that you execute or utility that you run, there are three files, each of which contains a different *stream* of data:

• **stdin (standard input)**: This is the file that the command reads to get its input. stdin has the file handle 0.

• **stdout (standard output)**: This is the file to which the command sends its output. stdout has the file handle 1.

• **stderr (standard error)**: This is the file to which the command sends any errors, also known as exceptions. stderr has the file handle 2.

A file handle is a number assigned to a file by the OS when that file is opened.

stdin is, by default, what you type on your keyboard. Similarly, stdout and stderr are, by default, displayed onscreen.

Input and output redirection are powerful capabilities in Linux that are very important pieces of the automation puzzle. Output that is normally seen onscreen can be redirected to a file. Output from a command can also be split into regular output and error, which can then be redirected separately. The contents of a file or the output of a command can be redirected to another command as input to that command.

The **sort** utility accepts input through stdin (via the keyboard), sorts the input in alphabetical order, and then sends the output to stdout (to the screen). In Example 2-28, after the user types the command **sort** and presses Enter, the shell waits for input from the user through the keyboard. The user types the letters q, w, e, r, t, and y on the keyboard, one by one, pressing Enter after each letter, to start a new line. The user then executes the **sort** command by pressing the Ctrl+d key combination. As shown in the example, the lines are sorted in alphabetical order, as expected.

Example 2-28 *Using the sort utility and providing the input through the default stdin stream, the keyboard*

```
 [NetProg@localhost LinuxStudies]$ sort
q
w
e
r
t
y ! Press ctrl+d here
e
q
r
t
w
y
[NetProg@localhost LinuxStudies]$
```

Input redirection can be used to change a command's stdin to a file instead of the keyboard. One way to do this is to specify the file as an *argument* to the command. Another way is to use the syntax {*command*} **<** {*file*}, where the contents of *file* are used as input to *command*. Example 2-29 shows how stdin to the **sort** command is changed to the file qwerty.txt using both methods.

Example 2-29 *Changing stdin for the* ***sort*** *Command from the Keyboard to a File by Providing the File as an Argument, and by Using Input Redirection*

```
[NetProg@localhost LinuxStudies]$ cat qwerty.txt
q
w
e
r
t
y
[NetProg@localhost LinuxStudies]$ sort qwerty.txt
e
q
r
t
w
y
[NetProg@localhost LinuxStudies]$ sort < qwerty.txt
e
q
```

```
r
t
w
y
[NetProg@localhost LinuxStudies]$
```

How to change stdout and stderr may be a bit more obvious than how to change stdin because the output from commands is usually expected to appear on the screen. Output redirection can be used to redirect the output to a file instead. In Example 2-30, the output from the **sort** command is output to the file qwerty-sorted.txt, and then the **cat** command is used to display the contents of the sorted file.

Example 2-30 *Redirecting Stdout to the File qwerty-sorted.txt with the* **>**

```
[NetProg@localhost LinuxStudies]$ ls
configurations  operational  QoS.txt  qwerty.txt  temp
[NetProg@localhost LinuxStudies]$ sort qwerty.txt > qwerty-sorted.txt
[NetProg@localhost LinuxStudies]$ ls
configurations  operational  QoS.txt  qwerty-sorted.txt  qwerty.txt  temp
[NetProg@localhost LinuxStudies]$ cat qwerty-sorted.txt
e
q
r
t
w
y
[NetProg@localhost LinuxStudies]$
```

Notice that file qwerty-sorted.txt did not exist before the **sort** command was executed. The file was created before it was used to store the redirected output. Similarly, if the file had existed before the command was executed, it would have been overwritten.

What if you need to append the output to the file instead of overwriting it ? Example 2-31 shows how to append output to an existing file by using the **>>** notation. As you saw earlier, the **cat** command outputs the contents of a file to the screen. In Example 2-31, instead of displaying the contents of QoS.txt on the screen, the **cat** command with the **>>** notation is used to redirect the file's contents to the qwerty-sorted.txt file, but this time the output is appended to the exiting content of qwerty-sorted.txt instead of overwriting it.

Example 2-31 *Appending Command Output by Using* **>>**

```
[NetProg@localhost LinuxStudies]$ cat QoS.txt >> qwerty-sorted.txt
[NetProg@localhost LinuxStudies]$ cat qwerty-sorted.txt
!
policy-map MOBILE_RAN_QOS_OUT
 !
 class MOBILE_VOICE_CLASS
  priority level 1
  police rate percent 50
   conform-action transmit
   exceed-action drop
  !
  set cos 5
 !
 class MOBILE_BROADBAND
  bandwidth percent 35
  set cos 3
  random-detect default
 !
 class class-default
  bandwidth percent 15
```

```
  set cos 0
  random-detect default
 !
 end-policy-map
!


q
w
e
r
t
y
[NetProg@localhost LinuxStudies]$
```

stderr is also, by default, displayed on the screen. If you want to redirect stderr to a file instead, you use the syntax {*command*} **2>** {*file*}, where the regular output goes to the screen, while the errors or exception messages are redirected to the file specified in the command. Example 2-32 shows how the error message from issuing the **stat** command on a nonexistent file is redirected to file error.txt. The **stat** command gives you important information about the file, such as the file size, inode number, permissions, user ID of the file owner, group ID of the file, and what time the file was last accessed, modified (content changed), and changed (metadata such as permissions changed).

Example 2-32 *Redirecting stderr to a File by Using* **2>**

```
[NetProg@localhost LinuxStudies]$ stat QoS.txt
  File: QoS.txt
  Size: 361           Blocks: 8          IO Block: 4096   regular file
Device: fd03h/64771d    Inode: 31719574    Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1001/ NetProg)   Gid: ( 1001/ NetProg)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2018-02-23 18:04:59.919457898 +0300
Modify: 2018-02-23 18:00:21.881647657 +0300
Change: 2018-02-23 18:00:21.881647657 +0300
 Birth: -
[NetProg@localhost LinuxStudies]$ stat WrongFile.txt
stat: cannot stat 'WrongFile.txt': No such file or directory
[NetProg@localhost LinuxStudies]$ stat WrongFile.txt 2> error.txt
[NetProg@localhost LinuxStudies]$ cat error.txt
stat: cannot stat 'WrongFile.txt': No such file or directory
[NetProg@localhost LinuxStudies]$
```

So far, you have seen how to redirect stdout to a file and how to do the same for stderr—but not both together. To redirect both stdout and stderr to a file, you use the syntax {*command*} **&>** {*file*}. Example 2-33 shows the **cat** command being used to concatenate three files: QoS.txt, WrongFile.txt, and qwerty.txt. However, one of these files, WrongFile.txt, does not exist, and so an error message is generated. As a result, the contents of QoS.txt and qwerty.txt are concatenated, and then both stdout and stderr are redirected to the same file, OutandErr.txt.

Example 2-33 *Redirecting Both stdout and stderr to OutandErr.txt*

```
[NetProg@localhost LinuxStudies]$ cat QoS.txt WrongFile.txt qwerty.txt &> OutandErr.txt
[NetProg@localhost LinuxStudies]$ cat OutandErr.txt
!
policy-map MOBILE_RAN_QOS_OUT
 !
 class MOBILE_VOICE_CLASS
  priority level 1
  police rate percent 50
   conform-action transmit
```

```
   exceed-action drop
  !
  set cos 5
 !
 class MOBILE_BROADBAND
  bandwidth percent 35
  set cos 3
  random-detect default
 !
 class class-default
  bandwidth percent 15
  set cos 0
  random-detect default
 !
 end-policy-map
!
cat: WrongFile.txt: No such file or directory
q
w
e
r
t
y
[NetProg@localhost LinuxStudies]$
```

To split stdout and stderr into their own separate files, you can use the syntax {*command*} **>** {*output_file*} **2>** {*error_file*}, as shown in Example 2-34.

Example 2-34 *Redirecting stdout and stderr Each to Its Own File*

```
[NetProg@localhost LinuxStudies]$ cat QoS.txt WrongFile.txt qwerty.txt > output.txt 2> error.txt
[NetProg@localhost LinuxStudies]$ cat output.txt
!
policy-map MOBILE_RAN_QOS_OUT
 !
 class MOBILE_VOICE_CLASS
  priority level 1
  police rate percent 50
   conform-action transmit
   exceed-action drop
  !
  set cos 5
 !
 class MOBILE_BROADBAND
  bandwidth percent 35
  set cos 3
  random-detect default
 !
 class class-default
  bandwidth percent 15
  set cos 0
```

```
  random-detect default
 !
 end-policy-map
!
q
w
e
r
t
y
[NetProg@localhost LinuxStudies]$ cat error.txt
cat: WrongFile.txt: No such file or directory
[NetProg@localhost LinuxStudies]$
```

To ignore or discard an error altogether and not save it to a file, you can simply redirect it to /dev/null. The file /dev/null is a special device file that discards any data redirected to it.

You can also append both stdout and stderr to an existing file by using the syntax {*command*} **>>** {*file*} **2>&1**.

As mentioned earlier in the chapter, Linux provides a facility to redirect the output of one command to be used as input for another command. This is done using the **|** (pipe) operator. Example 2-35 shows how the output of the **stat** command for the QoS.txt file is piped to the **sort** command, which sorts the output in alphabetical order. The result is then piped again to the **head** command to extract the first line of the output.

Example 2-35 *Piping Command Output to Another Command*

```
[NetProg@localhost LinuxStudies]$ stat QoS.txt
  File: QoS.txt
  Size: 361           Blocks: 8          IO Block: 4096   regular file
Device: fd03h/64771d    Inode: 31719574    Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1001/ NetProg)   Gid: ( 1001/ NetProg)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2018-02-23 18:04:59.919457898 +0300
Modify: 2018-02-23 18:00:21.881647657 +0300
Change: 2018-02-23 18:00:21.881647657 +0300
 Birth: -

! Stat output piped to sort
[NetProg@localhost LinuxStudies]$ stat QoS.txt | sort
Access: (0664/-rw-rw-r--)  Uid: ( 1001/ NetProg)   Gid: ( 1001/ NetProg)
Access: 2018-02-23 18:04:59.919457898 +0300
 Birth: -
Change: 2018-02-23 18:00:21.881647657 +0300
Context: unconfined_u:object_r:user_home_t:s0
Device: fd03h/64771d     Inode: 31719574    Links: 1
  File: QoS.txt
Modify: 2018-02-23 18:00:21.881647657 +0300
  Size: 361       .   Blocks: 8          IO Block: 4096   regular file

! Double piping to sort and then head
[NetProg@localhost LinuxStudies]$ stat QoS.txt | sort | head -n 1
Access: (0664/-rw-rw-r--)  Uid: ( 1001/ NetProg)   Gid: ( 1001/ NetProg)
[NetProg@localhost LinuxStudies]$
```

You have seen how stdout is by default displayed on the screen and how to redirect it to a file. But can you display it on the screen and

simultaneously redirect it to a file? Yes. You can use the pipe operator coupled with the **tee** command to do just that. In Example 2-36, the output of the command **ls -l** is piped to the **tee** command, and as a result, the output is both displayed on the screen and saved to the file lsoutput.txt.

Example 2-36 *Piping Command Output To the **tee** Command to Display It on the Screen As Well As Save It To File lsoutput.txt*

```
[NetProg@localhost LinuxStudies]$ ls -l | tee lsoutput.txt
total 40
-rw-rw-r--. 1 NetProg NetProg   46 Feb 23 20:28 colors.txt
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 23 16:59 configurations
-rw-rw-r--. 1 NetProg NetProg   61 Feb 23 18:15 error.txt
-rw-rw-r--. 1 NetProg NetProg  475 Feb 23 19:43 Existing.txt
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:34 operational
-rw-rw-r--. 1 NetProg NetProg  419 Feb 23 19:25 OutandErr.txt
-rw-rw-r--. 1 NetProg NetProg  361 Feb 23 18:00 QoS.txt
-rw-rw-r--. 1 NetProg NetProg  373 Feb 23 18:08 qwerty-sorted.txt
-rw-rw-r--. 1 NetProg NetProg   12 Feb 23 17:28 qwerty.txt
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 15:03 temp


[NetProg@localhost LinuxStudies]$ cat lsoutput.txt
total 40
-rw-rw-r--. 1 NetProg NetProg   46 Feb 23 20:28 colors.txt
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 23 16:59 configurations
-rw-rw-r--. 1 NetProg NetProg   61 Feb 23 18:15 error.txt
-rw-rw-r--. 1 NetProg NetProg  475 Feb 23 19:43 Existing.txt
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 12:34 operational
-rw-rw-r--. 1 NetProg NetProg  419 Feb 23 19:25 OutandErr.txt
-rw-rw-r--. 1 NetProg NetProg  361 Feb 23 18:00 QoS.txt
-rw-rw-r--. 1 NetProg NetProg  373 Feb 23 18:08 qwerty-sorted.txt
-rw-rw-r--. 1 NetProg NetProg   12 Feb 23 17:28 qwerty.txt
drwxrwxr-x. 2 NetProg NetProg 4096 Feb 17 15:03 temp
[NetProg@localhost LinuxStudies]$
```

The **tee** command overwrites the output file (in this case, the lsoutput.txt file). You can use the **tee** command with the **-a** option to append to the file instead of overwriting it.

Archiving Utilities

An archiving utility takes a file or a group of files as input and encodes the file or files into one single file, commonly known as an archive. The archiving utility also makes it possible to add files to the archive, remove files from the archive, update the files in the archive, or de-archive the archive file into its constituent files. Archiving utilities have historically been used for backup purposes and to package several files into one file that can be easily distributed, downloaded, and so on. The most commonly used archiving utility in Linux is the **tar** utility, which stands for *tape archive*. Archive files produced by the **tar** utility have a .tar extension and are commonly referred to as tarballs.

In contrast to an archiving utility, a compression utility takes a file or a group of files as input and compresses the file or files into another format that is smaller than the original file. This compression is *lossless*, meaning that no information is lost in the process. The compressed file can be decompressed and returned to its original state without any data or metadata being lost. In Linux, the most popular compression utilities are **gzip**, **bzip2**, and **xz**. The performance of compression utilities is measured based on several criteria, two of which are how quickly the compression happens and the compression ratio (which is how small the new compressed file is in comparison with the original uncompressed file). The **xz** utility is the best when it comes to compression ratio, but it is the slowest. The **gzip** utility is the fastest but has the lowest (worst) compression ratio. As you have already concluded, **bzip2** lies in the middle with respect to speed and compression ratio.

We cover archiving and compression utilities together in this section because a very common use case involves compressing files using one of the compression utilities listed here and then archiving the compressed files by using the **tar** utility. In addition to covering these utilities, this section also illustrates how compression and archiving can be performed using a single command.

Example 2-37 shows how to use the **gzip** utility to compress the InternetRoutes.txt file. You simply issue the command **gzip InternetRoutes.txt**, and the utility creates another file, InternetRoutes.txt.gz, which is the compressed file, and removes the original uncompressed file. To decompress the file back to its original form, you use the command **gunzip InternetRoutes.txt.gz**. What if you want to keep the original file as well as the compressed file after compression? You use **gzip** with the **-k** option, which stands for *keep*, as shown in the example. Similarly, you can use the **-k** option with **gunzip** to decompress the file and keep the compressed file intact.

Example 2-37 *Using the **gzip** Utility to Compress a Text File*

```
[NetProg@localhost LinuxStudies]$ ls -l
total 17212
-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 InternetRoutes.txt

[NetProg@localhost LinuxStudies]$ gzip InternetRoutes.txt
[NetProg@localhost LinuxStudies]$ ls -l
total 1268
-rw-rw-r--. 1 NetProg NetProg 1296408 Feb 24 12:27 InternetRoutes.txt.gz

[NetProg@localhost LinuxStudies]$ gunzip InternetRoutes.txt.gz
[NetProg@localhost LinuxStudies]$ ls -l
total 17212
-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 InternetRoutes.txt

[NetProg@localhost LinuxStudies]$ gzip -k InternetRoutes.txt
[NetProg@localhost LinuxStudies]$ ls -l
total 18480
-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 InternetRoutes.txt
-rw-rw-r--. 1 NetProg NetProg  1296408 Feb 24 12:27 InternetRoutes.txt.gz

[NetProg@localhost LinuxStudies]$ rm InternetRoutes.txt
[NetProg@localhost LinuxStudies]$ ls -l
total 1268
-rw-rw-r--. 1 NetProg NetProg 1296408 Feb 24 12:27 InternetRoutes.txt.gz

[NetProg@localhost LinuxStudies]$ gunzip -k InternetRoutes.txt.gz
[NetProg@localhost LinuxStudies]$ ls -l
total 18480
-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 InternetRoutes.txt
-rw-rw-r--. 1 NetProg NetProg  1296408 Feb 24 12:27 InternetRoutes.txt.gz
[NetProg@localhost LinuxStudies]$
```

Notice in the example that the size of the original uncompressed file is approximately 17 MB, and the size of the compressed file is approximately 1.2 MB; this represents a compression ratio of about 13.6.

Example 2-38 shows how the **bzip2** utility is used to compress the same InternetRoutes.txt file by using the command **bzip2 InternetRoutes.txt** and then uncompress the file by using the command **bunzip2 InternetRoutes.txt.bz2**.

Example 2-38 *Using the **bzip2** Utility to Compress a Text File*

```
[NetProg@localhost LinuxStudies]$ ls -l
total 17212
-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 InternetRoutes.txt

[NetProg@localhost LinuxStudies]$ bzip2 -kv InternetRoutes.txt
  InternetRoutes.txt: 19.386:1,  0.413 bits/byte, 94.84% saved, 17622037 in, 909025 out.
[NetProg@localhost LinuxStudies]$ ls -l
total 18100
-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 InternetRoutes.txt
-rw-rw-r--. 1 NetProg NetProg   909025 Feb 24 12:27 InternetRoutes.txt.bz2
```

```
[NetProg@localhost LinuxStudies]$ rm InternetRoutes.txt

[NetProg@localhost LinuxStudies]$ ls -l

total 888

-rw-rw-r--. 1 NetProg NetProg 909025 Feb 24 12:27 InternetRoutes.txt.bz2


[NetProg@localhost LinuxStudies]$ bunzip2 InternetRoutes.txt.bz2

[NetProg@localhost LinuxStudies]$ ls -l

total 17212

-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 InternetRoutes.txt

[NetProg@localhost LinuxStudies]$
```

Notice that the **-k** option also works with **bzip2**, and when used, the original uncompressed file is left intact. This option works equally well with **bunzip2** to leave the compressed file intact. As shown in Example 2-38, the **-v** option, which stands for *verbose*, provides some information and statistics on the compression process. It is worth noting that the verbose option is available for the vast majority of Linux commands, and it is available for use with all archiving and compression utilities in this chapter. If you look at the sizes of the original and compressed files, you see that the compression ratio in the example is approximately 19.4, which is in line with the verbose output.

Example 2-39 shows how to use the **xz** utility to compress the same InternetRoutes.txt file, using the command **xz InternetRoutes.txt**, and then uncompress the file by using the command **xz -d InternetRoutes.txt.xz**. The **-v** option is used here as well to provide some insight into the compression process.

Example 2-39 *Using the **xz** Utility to Compress a Text File*

```
[NetProg@localhost LinuxStudies]$ ls -l

total 17212

-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 InternetRoutes.txt


[NetProg@localhost LinuxStudies]$ xz -v InternetRoutes.txt

InternetRoutes.txt (1/1)

  100 %        711.9 KiB / 16.8 MiB = 0.041   2.3 MiB/s       0:07

[NetProg@localhost LinuxStudies]$ ls -l

total 712

-rw-rw-r--. 1 NetProg NetProg 728936 Feb 24 12:27 InternetRoutes.txt.xz


[NetProg@localhost LinuxStudies]$ xz -dv InternetRoutes.txt.xz

InternetRoutes.txt.xz (1/1)

  100 %        711.9 KiB / 16.8 MiB = 0.041

[NetProg@localhost LinuxStudies]$ ls -l

total 17212

-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 InternetRoutes.txt

[NetProg@localhost LinuxStudies]$
```

As you can see from the output in Example 2-39, **xz** provides a compression ratio of about 24.2, which is the best compression ratio so far. However, the **xz** utility takes a substantial amount of time (7 seconds, according to the verbose output) to compress the file. As shown in the earlier examples, compression using **gzip** is almost instantaneous, while **bzip2** takes a couple of seconds to compress the same file.

As mentioned at the beginning of this section, **tar** is an archiving utility that is used to group several files into a single archive file. Example 2-40 shows how **tar** is used to archive three files into one. To archive a number of files, you issue the command **tar -cvf** {*Archive_File.tar*} {*file1*} {*file2*} **..** {*fileX*}. The option **c** is for *create*, **v** is for *verbose*, and **f** is for stating the archive *filename* in the command. To view the constituent files of the archive, you use the command **tar -tf** {*Archive_File.tar*}. This does not extract the files in the archive. It only lists the files that make up the archive, as shown by the **ls -l** command in the example, right after this command is used. Finally, in order to extract the files from the archive, you use the command **tar -xvf** {*Archive_File.tar*}.

Example 2-40 *Using the **tar** Utility to Archive Three Files into One*

```
[NetProg@localhost LinuxStudies]$ ls -l

total 17260

-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 BGP.txt

-rw-rw-r--. 1 NetProg NetProg    43147 Feb 24 13:33 IPRoute.txt

-rw-r--r--. 1 NetProg NetProg      796 Feb 24 13:33 QoS.txt
```

```
! Archive three files into one tarball
[NetProg@localhost LinuxStudies]$ tar -cvf Archive.tar BGP.txt IPRoute.txt QoS.txt
BGP.txt
IPRoute.txt
QoS.txt
[NetProg@localhost LinuxStudies]$ ls -l
total 34520
-rw-rw-r--. 1 NetProg NetProg 17674240 Feb 24 15:22 Archive.tar
-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 BGP.txt
-rw-rw-r--. 1 NetProg NetProg    43147 Feb 24 13:33 IPRoute.txt
-rw-r--r--. 1 NetProg NetProg      796 Feb 24 13:33 QoS.txt

[NetProg@localhost LinuxStudies]$ rm BGP.txt IPRoute.txt QoS.txt
[NetProg@localhost LinuxStudies]$ ls -l
total 17260
-rw-rw-r--. 1 NetProg NetProg 17674240 Feb 24 15:22 Archive.tar

! Display the constituent files in the archive without de-archiving the tarball
[NetProg@localhost LinuxStudies]$ tar -tf Archive.tar
BGP.txt
IPRoute.txt
QoS.txt
[NetProg@localhost LinuxStudies]$ ls -l
total 17260
-rw-rw-r--. 1 NetProg NetProg 17674240 Feb 24 15:22 Archive.tar

! De-archive the tarball into its constituent files
[NetProg@localhost LinuxStudies]$ tar -xvf Archive.tar
BGP.txt
IPRoute.txt
QoS.txt
[NetProg@localhost LinuxStudies]$ ls -l
total 34520
-rw-rw-r--. 1 NetProg NetProg 17674240 Feb 24 15:22 Archive.tar
-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 BGP.txt
-rw-rw-r--. 1 NetProg NetProg    43147 Feb 24 13:33 IPRoute.txt
-rw-r--r--. 1 NetProg NetProg      796 Feb 24 13:33 QoS.txt
[NetProg@localhost LinuxStudies]$
```

Notice that the size of the archive file is actually a little bigger than the sizes of the constituent files added together. This is because archiving utilities do not compress files. Moreover, the archive file contains extra metadata that is required for describing the archive file contents and metadata related to the archiving process.

Luckily, the **tar** command can be used with certain options to summon compression utilities to compress the files before archiving them:

• To compress the files using the **gzip** utility before the **tar** archive is created, use the syntax **tar -zcvf** {*archive-file.tar.gz*} {*file1*} {*file2*} **..** {*fileX*}. To de-archive the tarball and then decompress the constituent files, use the syntax **tar -zxvf** {*archive-file.tar.gz*}.

• To compress the files using the **bzip2** utility before the **tar** archive is created, use the syntax **tar -jcvf {***archive-file.tar.bz2*} {*file1*} {*file2*} **..** {*fileX*}. To de-archive the tarball and then decompress the constituent files, use the syntax **tar -jxvf** {*archive-file.tar.bz2*}.

• To compress the files using the **xz** utility before the **tar** archive is created, use the syntax **tar -Jcvf {***archive-file.tar.xz*} {*file1*} {*file2*} **..** {*fileX*}.

To de-archive the tarball and then decompress the constituent files, use the syntax **tar -Jxvf** {*archive-file.tar.xz*}.

Example 2-41 shows the **tar** command being used with the **-J** option, which summons the **xz** utility to compress the files before **tar** archives these files. Then the same option is used to decompress the files after they are extracted from the **tar** archive.

Example 2-41 *Using the **tar** Utility with **xz** to Compress and Archive Three Files into One*

```
[NetProg@localhost LinuxStudies]$ ls -l
total 17260
-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 BGP.txt
-rw-rw-r--. 1 NetProg NetProg    43147 Feb 24 13:33 IPRoute.txt
-rw-r--r--. 1 NetProg NetProg      796 Feb 24 13:33 QoS.txt

[NetProg@localhost LinuxStudies]$ tar -Jcvf Archive.tar.xz BGP.txt IPRoute.txt QoS.txt
BGP.txt
IPRoute.txt
QoS.txt
[NetProg@localhost LinuxStudies]$ ls -l
total 17980
-rw-rw-r--. 1 NetProg NetProg   735548 Feb 24 16:01 Archive.tar.xz
-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 BGP.txt
-rw-rw-r--. 1 NetProg NetProg    43147 Feb 24 13:33 IPRoute.txt
-rw-r--r--. 1 NetProg NetProg      796 Feb 24 13:33 QoS.txt

[NetProg@localhost LinuxStudies]$ rm BGP.txt IPRoute.txt QoS.txt
[NetProg@localhost LinuxStudies]$ ls -l
total 720
-rw-rw-r--. 1 NetProg NetProg 735548 Feb 24 16:01 Archive.tar.xz

[NetProg@localhost LinuxStudies]$ tar -Jxvf Archive.tar.xz
BGP.txt
IPRoute.txt
QoS.txt
[NetProg@localhost LinuxStudies]$ ls -l
total 17980
-rw-rw-r--. 1 NetProg NetProg   735548 Feb 24 16:01 Archive.tar.xz
-rw-rw-r--. 1 NetProg NetProg 17622037 Feb 24 12:27 BGP.txt
-rw-rw-r--. 1 NetProg NetProg    43147 Feb 24 13:33 IPRoute.txt
-rw-r--r--. 1 NetProg NetProg      796 Feb 24 13:33 QoS.txt
[NetProg@localhost LinuxStudies]$
```

Notice that the size of the **tar** file is now smaller in size than the sizes of the constituent files added together. This is due to the compression preceding the archiving. Notice also that the archive file is named with file extension .tar.xz. This is not mandatory, and the command works just fine if the archive is just a filename with no extension. However, this extension enables a user to identify the file as a **tar** archive that has been compressed using the **xz** compression utility.

Linux System Maintenance

This section discusses general maintenance of a Linux system. In order to maintain a healthy Linux system as well as troubleshoot and resolve system incidents, you need to understand how to do the following:

- Manage jobs and processes
- Monitor utilization of CPU, memory, and other resources
- Collect system information
- Locate, read, and analyze system logs

The following sections discuss these points in some depth. For further details, you can consult the man or info pages for each command.

Job, Process, and Service Management

In operating systems jargon, a *thread* is a sequence of instructions that are executed by the CPU. A thread is a basic building block and cannot be broken up into smaller components to be executed simultaneously.

A *process* is a group of threads. Two or more of those threads may be executed simultaneously in a multithreaded system in order to run a process faster. A process has its own address space in memory. Linux virtualizes memory such that each process thinks that it has exclusive access to all the physical memory on the system even though it actually only has access to its own process address space. Utilities such as **ls**, **cp**, and **cat** run as processes.

A *job* may be composed of two or more processes. For example, running the command **ls** starts a process, while piping **ls** to **less** using the command **ls | less** starts a job composed of more than one process.

A *service* is composed of one or more processes and provides a specific function; examples are the HTTP, NTP, and SSH services. A service is also usually run in the background and is therefore referred to as a *daemon*. Service names in Linux almost always end with the letter **d**. Services are briefly introduced earlier in this chapter.

As you progress through this section, the differences between processes, jobs, and services will become more apparent.

The command **ps** lists the processes currently running on the system. Without any options or arguments, the command lists the running processes associated with the current user and terminal, as shown in Example 2-42.

Example 2-42 ***ps*** *Command Output*

```
[NetProg@localhost ~]$ ps
  PID TTY          TIME CMD
 2897 pts/0    00:00:00 bash
 2954 pts/0    00:00:00 ps
[NetProg@localhost ~]$
```

For each process, the **ps** command output lists the following fields:

• **PID**: This is the process ID, which is a number that uniquely identifies each process.

• **TTY**: This is the terminal number from which the process was started. pts/0 in the output stands for pseudo-terminal slave 0. The first terminal window you open will be pts/0, the second pts/1, and so forth.

• **TIME**: This is the total amount of time the process spent consuming the CPU throughout the duration of its lifetime.

• **CMD**: This is the process name.

For more detailed output, several options can be added to the **ps** command. Adding the **-A** or **-e** options lists all processes running on the system for all users and all TTY lines, as shown in Example 2-43. The output is in the same format as the vanilla **ps** command output. In order to compare the output from both commands, you can pipe the output to **wc -l**. The command **wc** stands for *word count*, and when used with the **-l** option, the command returns the number of lines in the command argument (in this case, the output of the **ps** command). As you can see, both commands return 189 lines of output. The purpose of this example is two-fold: to display the output of the **ps** command using both options and to introduce the very handy command **wc -l**.

Example 2-43 ***ps -e*** *and* ***ps -A*** *Commands*

```
[NetProg@localhost ~]$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:02 systemd
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    5 ?        00:00:00 kworker/0:0H
    7 ?        00:00:00 migration/0
    8 ?        00:00:00 rcu_bh
    9 ?        00:00:01 rcu_sched
   10 ?        00:00:00 watchdog/0
   12 ?        00:00:00 kdevtmpfs


--------- output truncated for brevity ---------


[NetProg@localhost ~]$ ps -A
  PID TTY          TIME CMD
```

```
    1 ?        00:00:02 systemd
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    5 ?        00:00:00 kworker/0:0H
    7 ?        00:00:00 migration/0
    8 ?        00:00:00 rcu_bh
    9 ?        00:00:01 rcu_sched
   10 ?        00:00:00 watchdog/0
   12 ?        00:00:00 kdevtmpfs


--------- output truncated for brevity ---------


[NetProg@localhost ~]$ ps -e | wc -l
189
[NetProg@localhost ~]$ ps -A | wc -l
189
[NetProg@localhost ~]$
```

Notice in Example 2-43 that the TTY field shows a question mark (?) throughout the output. This indicates that these processes are not associated with a terminal window, referred to as a *controlling terminal*.

The command **ps -u** lists all the processes owned by the *current* user and adds to the information displayed for each process. To display the processes for any *other* user, you use the syntax **ps -u** {*username*}. Example 2-44 shows the output of **ps -u**, which lists the processes owned by the user NetProg (the current user).

Example 2-44 ***ps -u*** *Command Output*

```
[NetProg@localhost ~]$ ps -u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
NetProg   4312  0.0  0.0 116564  3280 pts/1    Ss   12:22   0:00 bash
NetProg   4517  0.0  0.0 116564  3280 pts/2    Ss   12:33   0:00 bash
NetProg   5609  0.0  0.0 119552  2284 pts/2    S+   13:59   0:00 man ps
NetProg   5620  0.0  0.0 110260   944 pts/2    S+   13:59   0:00 less -s
NetProg   7990  0.0  0.0 119552  2284 pts/1    S+   17:40   0:00 man ps
NetProg   8001  0.0  0.0 110260   948 pts/1    S+   17:40   0:00 less -s
NetProg   8827  0.0  0.0 116564  3288 pts/0    Ss   18:28   0:00 bash
NetProg  10108  0.0  0.0 151064  1792 pts/0    R+   19:49   0:00 ps -u
[NetProg@localhost ~]$
```

Notice that the output of **ps -u** adds seven more fields to the output:

- **User**: The user ID of the process owner
- **%CPU**: The CPU time the process used divided by the process runtime (process lifetime), expressed as a percentage
- **%MEM**: The ratio of the main memory used by the process (resident set size) to the total amount of main memory on the system, expressed as a percentage
- **VSZ**: Virtual memory size, the amount of virtual memory used by the process, expressed in kilobytes
- **RSS**: Resident set size, the amount of main memory (RAM) used by the process, expressed in kilobytes
- **STAT**: The state of the process
- **START**: The start time of the process

The process STAT field contains 1 or more of the 14 characters describing the state of the process. For example, state S indicates that the process is in the sleep state (that is, waiting for an event to happen in order to resume running, such as waiting for input from the user). The **+** indicates that the process is running in the foreground rather than running in the background.

Processes are grouped into *process groups*, and one or more process groups make up a *session*. All the processes in one pipeline, such as **cat**, **sort**, and **tail** in **cat file.txt | sort | tail -n 10**, are in the same process group and have the same process group ID (PGID). The process whose PID is the same as its PGID is the *process group leader*, and it is the first member of the process group. On the other hand, all process

groups started by a shell are in the same session, and they have the same session ID (SID). The process whose PID is the same as its SID is the *session leader*. In Example 2-44, as expected, the two shell processes (**bash**) are session leaders, as indicated by the s in their STAT field. To check the PID, PGID, and SID of a process all at once, issue the command **ps -j**.

To list all processes that have a specific name, use the **-C** option. In Example 2-45, **ps -C bash** lists all processes that are named **bash**.

Example 2-45 ***ps -C bash*** *Command Output*

```
[NetProg@localhost ~]$ ps -C bash
  PID TTY          TIME CMD
 4312 pts/1    00:00:00 bash
 4517 pts/2    00:00:00 bash
 8827 pts/0    00:00:00 bash
[NetProg@localhost ~]$
```

Finally, to see the parent process ID (PPID) of a process, you can use the option **-f**. As the name implies, the parent process is the process that started this process. In Example 2-46, the command **ps -ef | head -n 10** is used to display the first 10 processes in the list, along with the **PPID** of each process.

Example 2-46 ***ps -ef | head -n 10*** *Command Output*

```
[NetProg@localhost ~]$ ps -ef | head -n 10
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 09:29 ?        00:00:02 /usr/lib/systemd/systemd --switched-root --system --deserialize 21
root         2     0  0 09:29 ?        00:00:00 [kthreadd]
root         3     2  0 09:29 ?        00:00:00 [ksoftirqd/0]
root         5     2  0 09:29 ?        00:00:00 [kworker/0:0H]
root         7     2  0 09:29 ?        00:00:00 [migration/0]
root         8     2  0 09:29 ?        00:00:00 [rcu_bh]
root         9     2  0 09:29 ?        00:00:00 [rcu_sched]
root        10     2  0 09:29 ?        00:00:00 [watchdog/0]
root        12     2  0 09:29 ?        00:00:00 [kdevtmpfs]
[NetProg@localhost ~]$
```

Note that the process with PID 0 is the kernel, and the process with PID 1 is the **systemd** process, or the **init** process in some systems (recall the Linux boot process?). Knowing this, the output in Example 2-46 should make more sense.

As you have seen from the output of **ps -e | wc -l** in Example 2-43, the list of running processes can be very long. While the output can be piped to **grep** in order to list specific lines of the command output, the use of the command **pgrep** may be a little more intuitive. Example 2-47 shows the output of the command **pgrep -u NetProg -l bash**, showing all processes owned by user NetProg and named **bash**.

Example 2-47 ***pgrep -u NetProg - l bash*** *Command Output*

```
[NetProg@localhost ~]$ pgrep -u NetProg -l bash
3173 bash
5815 bash
6561 bash
6667 bash
6730 bash
[NetProg@localhost ~]$
```

You can start and stop processes by using the **kill** command. The **kill** command sends 1 of 64 signals to a process or process group. This signal may be a **SIGTERM** signal to request the process to terminate gracefully, or it may be a **SIGKILL** signal to force the termination of the process. The signals **SIGSTOP** and **SIGCONT** are also used to pause and resume a process, respectively. To view all the available signals, you can issue the command **kill -l**, as shown in Example 2-48. The default signal **SIGTERM** is used if no signal is explicitly specified in the command. The command **kill** may be used with the signal numeric value or signal name.

Example 2-48 *Using* ***kill -l*** *to List Signals Used with the* ***kill*** *Command*

```
[NetProg@localhost ~]$ 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
[NetProg@localhost ~]$
```

Example 2-49 shows how to pause, resume, and kill the process **bash** with process ID 3173. The **-p** option is used with the **ps** command to list a specific process using its PID.

Example 2-49 *Pausing, Resuming, and Killing the Process **bash** Using the **kill** Command*

```
[NetProg@localhost ~]$ ps -C bash
  PID TTY          TIME CMD
 3173 pts/1    00:00:00 bash
 5815 ?        00:00:00 bash
 8501 pts/3    00:00:00 bash
 8980 pts/4    00:00:00 bash
 9233 pts/2    00:00:00 bash
[NetProg@localhost ~]$ ps -up 3173
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
NetProg   3173  0.0  0.0 116696  3440 pts/1    Ss   11:41   0:00 bash
[NetProg@localhost ~]$ kill -SIGSTOP 3173
[NetProg@localhost ~]$ ps -up 3173
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
NetProg   3173  0.0  0.0 116696  3440 pts/1    Ts   11:41   0:00 bash
[NetProg@localhost ~]$ kill -SIGCONT 3173
[NetProg@localhost ~]$ ps -up 3173
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
NetProg   3173  0.0  0.0 116696  3440 pts/1    Ss   11:41   0:00 bash
[NetProg@localhost ~]$ kill -SIGTERM 3173
[NetProg@localhost ~]$ ps -up 3173
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
NetProg   3173  0.0  0.0 116696  3440 pts/1    Ss   11:41   0:00 bash
[NetProg@localhost ~]$ kill -SIGKILL 3173
[NetProg@localhost ~]$ ps -up 3173
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
[NetProg@localhost ~]$ ps -C bash
  PID TTY          TIME CMD
 5815 ?        00:00:00 bash
 8501 pts/3    00:00:00 bash
 8980 pts/4    00:00:00 bash
 9233 pts/2    00:00:00 bash
[NetProg@localhost ~]$
```

As you can see from Example 2-49, when process **bash** with PID 3137 receives the **SIGSTOP** signal, its state changes from **Ss** (interruptible sleep, indicated by **S**, and session leader, indicated by **s**) to **Ts** (stopped by job control signal, indicated by **T**, and session leader, indicated

by **s**).

The process returns to the **Ss** state when it receives the **SIGCONT** signal. When the **SIGTERM** signal is then used in an attempt to terminate the process, its state does not change; therefore, **SIGKILL** is used, and it successfully forces the process to terminate. It should be noted, however, that it is generally not recommended to terminate a process by using the **SIGKILL** signal unless the process is suspected to be malicious or is not properly responding.

Jobs, on the other hand, can be displayed by using the **jobs** command. The **jobs** command lists all jobs run by the current shell. In Example 2-50, a simple **for** loop is used to create a job that runs indefinitely. (Loops and control structures in Bash are covered in Chapter 4.) In addition, you can enter the command **gedit** to start the text editor program. An **&** is added at the end of both command lines shown in Example 2-50. This instructs the shell to run both jobs in the background, so the running process will not hog the shell prompt, and the prompt will be available for you to enter other commands. A third job is created by running the **ping** command to google.com in the foreground. The **ping** command is then stopped (paused) by using the Ctrl+z key combination. The command **jobs** then lists all three jobs.

Example 2-50 *Using the **jobs** Command to Display Job Status*

```
[NetProg@localhost ~]$ jobs

[NetProg@localhost ~]$ i=0; while true; do ((i++)); sleep 5; done &

[1] 19332

[NetProg@localhost ~]$ gedit &

[2] 19347

[NetProg@localhost ~]$ ping google.com

PING google.com (172.217.18.46) 56(84) bytes of data.

64 bytes from ham02s12-in-f46.1e100.net (172.217.18.46): icmp_seq=1 ttl=63 time=220 ms

64 bytes from ham02s12-in-f46.1e100.net (172.217.18.46): icmp_seq=2 ttl=63 time=280 ms

^Z

[3]+  Stopped                 ping google.com

[NetProg@localhost ~]$ jobs

[1]   Running      while true; do ((i++)); sleep 5; done &

[2]-  Running      gedit &

[3]+  Stopped      ping google.com

[NetProg@localhost ~]$
```

The number in brackets in Example 2-50 is the job number, and the number after that is the PID. The jobs in the example are numbered 1 to 3. The first two jobs are in Running state, and the ping job is in the Stopped state. To list the running jobs only, you use the **jobs -r** command, and to display the stopped jobs only, you use the **jobs -s** command.

To resume a stopped process, you bring the process to the foreground by using the command **fg** {*job_number*}. To send it to the background again, you use the command **bg** {*job_number*}. When the job is running in the foreground, you can stop it by using the Ctrl+c key combination. Example 2-51 shows a **ping** process brought to the foreground and stopped.

Example 2-51 *Bringing a **ping** Job to the Foreground and Stopping It*

```
[NetProg@localhost ~]$ jobs

[1]   Running      while true; do ((i++)); sleep 5; done &

[2]-  Running      gedit &

[3]+  Stopped      ping google.com

[NetProg@localhost ~]$ fg 3

ping google.com

64 bytes from ham02s12-in-f46.1e100.net (172.217.18.46): icmp_seq=3 ttl=63 time=5463 ms

64 bytes from ham02s12-in-f46.1e100.net (172.217.18.46): icmp_seq=4 ttl=63 time=5261 ms

64 bytes from ham02s12-in-f46.1e100.net (172.217.18.46): icmp_seq=5 ttl=63 time=4698 ms

64 bytes from ham02s12-in-f46.1e100.net (172.217.18.46): icmp_seq=6 ttl=63 time=5178 ms

^C

--- google.com ping statistics ---

11 packets transmitted, 6 received, 45% packet loss, time 633961ms

rtt min/avg/max/mdev = 220.614/3517.335/5463.953/2321.238 ms, pipe 6

[NetProg@localhost ~]$
```

If a job is running in the background and you want to stop it without bringing it to the foreground first, you use the **kill** command in exactly the

same way you use it with processes. Example 2-52 shows the **kill** command being used to terminate the two running jobs. Notice that the **-l** option is used with the **jobs** command to add a PID column to the output.

Example 2-52 *Terminating a Job Using the **kill** Command*

```
[NetProg@localhost ~]$ jobs -l
[1]- 19332 Running    while true; do ((i++)); sleep 5; done &
[2]+ 19347 Running    gedit &
[NetProg@localhost ~]$ kill 19332
[1]-  Terminated      while true; do ((i++)); sleep 5; done
[NetProg@localhost ~]$ jobs
[2]+  Running         gedit &
[NetProg@localhost ~]$ kill 19347
[2]+  Terminated      gedit
[NetProg@localhost ~]$ jobs
[NetProg@localhost ~]$
```

To view service status, start and stop services, and carry out other service-related operations, you can use the command **systemctl**. The general syntax of the command is **systemctl** {*options*} {*service_name*}. These are the most common options for this command:

- **status**: Displays the status of the service
- **start**: Starts the service
- **stop**: Stops the service
- **enable**: Enables the service so that it is automatically started at system startup
- **disable**: Disables the service so that it is not automatically started at system startup

Example 2-53 shows how to check the status of the **httpd** service, start it, stop it, and enable it.

Example 2-53 *Using the **systemctl** Command to Viewand Change the Status of the **httpd** Service*

```
[netdev@server1 LinuxStudies]$ systemctl status httpd
● httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; disabled; vendor preset: disabled)
   Active: inactive (dead)
     Docs: man:httpd.service(8)
[netdev@server1 LinuxStudies]$
```

```
[NetProg@localhost ~]$ systemctl status httpd
• httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; disabled; vendor preset: disabled)
   Active: inactive (dead)
     Docs: man:httpd(8)
           man:apachectl(8)
[NetProg@localhost ~]$ sudo systemctl enable httpd
[sudo] password for NetProg:
Created symlink from /etc/systemd/system/multi-user.target.wants/httpd.service to /usr/lib/systemd/system/httpd.service.
[NetProg@localhost ~]$ systemctl status httpd
• httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; vendor preset: disabled)
   Active: inactive (dead)
     Docs: man:httpd(8)
           man:apachectl(8)
[NetProg@localhost ~]$ sudo systemctl start httpd
[NetProg@localhost ~]$ systemctl status httpd
• httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; vendor preset: disabled)
```

```
   Active: active (running) since Sun 2018-04-08 17:35:40 +03; 3s ago
     Docs: man:httpd(8)
           man:apachectl(8)
 Main PID: 2921 (httpd)
   Status: "Processing requests..."
   CGroup: /system.slice/httpd.service
           ├─2921 /usr/sbin/httpd -DFOREGROUND
           ├─2925 /usr/sbin/httpd -DFOREGROUND
           ├─2926 /usr/sbin/httpd -DFOREGROUND
           ├─2927 /usr/sbin/httpd -DFOREGROUND
           ├─2928 /usr/sbin/httpd -DFOREGROUND
           └─2929 /usr/sbin/httpd -DFOREGROUND

Apr 08 17:35:40 localhost.localdomain systemd[1]: Starting The Apache HTTP Ser....
Apr 08 17:35:40 localhost.localdomain httpd[2921]: AH00558: httpd: Could not r...e
Apr 08 17:35:40 localhost.localdomain systemd[1]: Started The Apache HTTP Server.
Hint: Some lines were ellipsized, use -l to show in full.
[NetProg@localhost ~]$ sudo systemctl stop httpd
[sudo] password for NetProg:

[NetProg@localhost ~]$ systemctl status httpd
• httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled; vendor preset: disabled)
   Active: inactive (dead) since Sun 2018-04-08 17:41:14 +03; 8s ago
     Docs: man:httpd(8)
           man:apachectl(8)
  Process: 3132 ExecStop=/bin/kill -WINCH ${MAINPID} (code=exited, status=0/SUCCESS)
  Process: 2921 ExecStart=/usr/sbin/httpd $OPTIONS -DFOREGROUND (code=exited, status=0/SUCCESS)
 Main PID: 2921 (code=exited, status=0/SUCCESS)
   Status: "Total requests: 0; Current requests/sec: 0; Current traffic:   0 B/sec"

Apr 08 17:35:40 localhost.localdomain systemd[1]: Starting The Apache HTTP Ser....
Apr 08 17:35:40 localhost.localdomain httpd[2921]: AH00558: httpd: Could not r...e
Apr 08 17:35:40 localhost.localdomain systemd[1]: Started The Apache HTTP Server.
Apr 08 17:41:13 localhost.localdomain systemd[1]: Stopping The Apache HTTP Ser....
Apr 08 17:41:14 localhost.localdomain systemd[1]: Stopped The Apache HTTP Server.
Hint: Some lines were ellipsized, use -l to show in full.
[NetProg@localhost ~]$
```

In Example 2-53, the **httpd** service is both inactive and disabled. When the **enable** option is used, the **httpd** service changes its state to enabled, which means the service will be automatically started when the system is booted. However, the service is still inactive; that is, it is *not* currently running. When you use the **start** option, the **httpd** service becomes active. Finally, the service is stopped using the **stop** option. Note that starting and stopping the service is independent of the service's enabled/disabled status. The former describes the *current* status of the service, while the latter describes whether the service should be started automatically at system startup time.

Resource Utilization

Resource utilization, at a very basic level, refers to CPU, memory, and storage utilization. While checking disk space on a system tends to be a straightforward process, checking the CPU and memory utilization can be quite challenging if you don't know exactly what tools to use. The single most important Linux command to use to check resource utilization is **top**.

Example 2-54 shows the output of the **top** command. The list of processes is live—that is, updated in real time as the output is being viewed. The output is also limited by the shell window size. The bigger the window, the longer the list of processes that you can view.

Example 2-54 *Output of the* ***top*** *Command*

```
top - 23:52:06 up 3 min,  3 users,  load average: 0.23, 0.34, 0.16

Tasks: 205 total,   1 running, 204 sleeping,   0 stopped,   0 zombie

%Cpu(s):  0.9 us,  0.6 sy,  0.0 ni, 98.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st

KiB Mem :  8010152 total,  6713388 free,   783364 used,   513400 buff/cache

KiB Swap:  5242876 total,  5242876 free,        0 used.  6940600 avail Mem


  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND

 2051 NetProg   20   0 2520892 226276  47980 S   4.3  2.8   0:17.71 gnome-shell

 1183 root      20   0  354540  55024  10848 S   2.6  0.7   0:05.44 X

 2675 NetProg   20   0  723988  25624  15312 S   1.3  0.3   0:02.18 gnome-terminal-

 2788 NetProg   20   0  157860   2368   1532 R   1.0  0.0   0:01.06 top

 2789 NetProg   20   0  157944   2316   1532 S   1.0  0.0   0:01.00 top

 1992 NetProg   20   0  214904   1312    880 S   0.3  0.0   0:00.41 VBoxClient

    1 root      20   0  193708   6844   4068 S   0.0  0.1   0:01.85 systemd

    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd

    3 root      20   0       0      0      0 S   0.0  0.0   0:00.02 ksoftirqd/0

    4 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0


--------- Output truncated for brevity ---------
```

The following is a list of keys that, when pressed, change the formatting of the output while the **top** command is running:

- **m**: Pressing this key shows memory usage, as a percentage.
- **t**: Pressing this key shows CPU usage, as a percentage.
- **1**: Pressing this key shows all processors on the system.
- **Shift+m**: Pressing this key combination sorts processes by memory usage, in descending order.
- **Shift+p**: Pressing this key combination sorts processes by CPU usage, in descending order.
- **Shift+r**: Pressing this key combination sorts processes by PID.
- **k**-{*PID*}-{*Signal_No*|*Signal_Name*}: Pressing **k** starts a dialog above the first column of the process list. This dialog requests a PID. After you type a PID and press Enter, it requests the signal you want to send to that process. This can be used to send any of the 64 signals to any of the processes on the system.

Example 2-55 shows the output you get when you use the **top** command and press 1 followed by t. As you can see, all four processors on the system are listed, with the percentage utilization of each.

Example 2-55 *Output of the* ***top*** *Command Showing Each of the Four CPUs Being Used*

```
top - 00:06:33 up 17 min,  4 users,  load average: 0.11, 0.08, 0.10

Tasks: 201 total,   2 running, 198 sleeping,   1 stopped,   0 zombie

%Cpu0  :   6.8/2.3     9[|||||                                                  ]

%Cpu1  :   8.2/0.0     8[||||                                                   ]

%Cpu2  :   4.2/2.1     6[|||                                                    ]

%Cpu3  :   1.9/0.0     2[|                                                      ]

KiB Mem :  8010152 total,  6692660 free,   803756 used,   513736 buff/cache

KiB Swap:  5242876 total,  5242876 free,        0 used.  6920176 avail Mem


  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND

 2051 NetProg   20   0 2521404 226756  47984 S  31.4  2.8   0:44.63 /usr/bin/gn+

 1183 root      20   0  359576  59968  10900 S   9.8  0.7   0:15.58 /usr/bin/X +

 2675 NetProg   20   0  725372  27116  15320 D   3.9  0.3   0:07.04 /usr/libexe+

  692 root      20   0    6472    652    540 S   2.0  0.0   0:00.16 /sbin/rngd +
```

```
 1992 NetProg   20   0  214904   1312    880 S   2.0  0.0   0:02.36 /usr/bin/VB+

 2212 NetProg   20   0 1520856  28752  17476 S   2.0  0.4   0:01.20 /usr/libexe+

 2788 NetProg   20   0  157888   2404   1564 R   2.0  0.0   0:06.34 top

    1 root      20   0  193708   6844   4068 S   0.0  0.1   0:02.05 /usr/lib/sy+


--------- Output truncated for brevity ---------
```

When troubleshooting an incident, it is sometimes useful to have **top** run with a refresh rate that is faster than the default. The command **top -d** {*N*} runs **top** and refreshes the output every *N* seconds. *N* does not have to be an integer; it can be a fraction of a second.

System Information

Linux provides several ways to collect information describing the hardware and software of the system it is running on, as well as set and change this information, where applicable.

The **date** command, as shown in Example 2-56, displays the date and time configured on the system. Adding the **-R** option to the command displays the same information in RFC 2822 format.

Example 2-56 *Output of the **date** Command*

```
[NetProg@localhost ~]$ date

Sun Apr  8 01:38:48 +03 2018

[NetProg@localhost ~]$ date -R

Sun, 08 Apr 2018 01:38:53 +0300

[NetProg@localhost ~]$
```

From left to right, the first command output in Example 2-56 displays the following information:

- Day of the week
- Month
- Day
- Time, in *hh*:*mm*:*ss* format
- Time zone
- Year

Another command you can use to view and set the system time and date is the **timedatectl** command, shown in Example 2-57.

Example 2-57 *Output of the **timedatectl** Command*

```
[NetProg@localhost ~]$ timedatectl

      Local time: Sun 2018-04-08 11:40:50 +03

  Universal time: Sun 2018-04-08 08:40:50 UTC

        RTC time: Sun 2018-04-08 08:40:48

       Time zone: Asia/Riyadh (+03, +0300)

     NTP enabled: no

NTP synchronized: no

 RTC in local TZ: no

      DST active: n/a

[NetProg@localhost ~]$
```

The **uptime** command displays how long the system has been running as well as CPU load average. Example 2-58 shows the output from the **uptime** command.

Example 2-58 *Output of the **uptime** Command*

```
[NetProg@localhost ~]$ uptime

 22:39:25 up 41 min,  2 users,  load average: 0.02, 0.06, 0.11

[NetProg@localhost ~]$
```

The **uptime** command output in Example 2-56 displays the following information:

- The system time when the command was issued (in this example, 10:39:25 p.m.)
- How long the system has been up (in this case 41 minutes)

• How many users are logged in (in this case 2 users)

• The load average over the past 1 minute, 5 minutes, and 15 minutes

The load average is an indication of the average system utilization over a specific duration. The load average factors in all processes that are either using the CPU or waiting to use the CPU (runnable state), as well as processes waiting for I/O access, such as disk access (uninterruptable state). If one process is in either of these states for a duration of 1 minute, then the load average over 1 minute is 1 for a single-processor system.

The output of the **uptime** command shows the load average over the past 1, 5, and 15 minutes. A 0.2 in the first load average field of the output of the **uptime** command indicates an average load of 20% over the past 1 minute if the system has a single processor. For multiprocessor systems, the load average should be divided by the number of processors in the system. Therefore, a 0.2 value in a system with four processors means a load average of 5%. A value of 1 in a four-processor system indicates a load average of 25%.

When you have multiple processors on a system, you can view detailed processor information by viewing the file /proc/cpuinfo. Example 2-59 displays part of the contents of this file. The cpuinfo file lists each processor and details for each of them. Processors are numbered 0 to the number of processors minus 1. A quick way to display the number of processors on a system is to use the command **cat /proc/cpuinfo | grep processor | wc -l**.

Example 2-59 *CPU Information from the /proc/cpuinfo File*

```
[root@localhost ~]# cat /proc/cpuinfo

processor    : 0

vendor_id    : GenuineIntel

cpu family    : 6

model        : 158

model name    : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz

stepping    : 9

cpu MHz        : 2837.118

cache size    : 6144 KB

physical id    : 0

siblings    : 4

core id        : 0

cpu cores    : 4

apicid        : 0

initial apicid    : 0

fpu        : yes

fpu_exception    : yes

cpuid level    : 22

wp        : yes

flags        : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht
syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc pni pclmulqdq ssse3 cx16 sse4_1 sse4_2 x2apic movbe
popcnt aes xsave avx rdrand hypervisor lahf_lm abm 3dnowprefetch avx2 rdseed clflushopt

bogomips    : 5674.23

clflush size    : 64

cache_alignment    : 64

address sizes    : 39 bits physical, 48 bits virtual

power management:


processor    : 1

vendor_id    : GenuineIntel

cpu family    : 6

model        : 158

model name    : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz


--------- output truncated for brevity ---------
```

```
[root@localhost ~]# cat /proc/cpuinfo | grep processor | wc -l
4
[root@localhost ~]#
```

You can also view processor information by using the command **dmidecode**. You can use this command to display information about a variety of system resources, including CPU and memory. The command **dmidecode -t** lists the resources that the command can provide information on, as shown in Example 2-60.

Example 2-60 *dmidecode Command Options*

```
[root@localhost ~]# dmidecode -t
dmidecode: option requires an argument -- 't'
Type number or keyword expected
Valid type keywords are:
  bios
  system
  baseboard
  chassis
  processor
  memory
  cache
  connector
  slot
[root@localhost ~]#
```

For example, to display the details of the system memory, you can use the command **dmidecode -t memory**, as shown in Example 2-61.

Example 2-61 *Getting Memory Information by Using the dmidecode - t memory Command*

```
[root@localhost ~]# dmidecode -t memory
# dmidecode 3.1
Getting SMBIOS data from sysfs.
SMBIOS 3.0.0 present.

Handle 0x0003, DMI type 16, 23 bytes
Physical Memory Array
    Location: System Board Or Motherboard
    Use: System Memory
    Error Correction Type: None
    Maximum Capacity: 32 GB
    Error Information Handle: Not Provided
    Number Of Devices: 2

Handle 0x0004, DMI type 17, 40 bytes
Memory Device
    Array Handle: 0x0003
    Error Information Handle: Not Provided
    Total Width: 64 bits
    Data Width: 64 bits
    Size: 16384 MB
    Form Factor: SODIMM
    Set: None
    Locator: ChannelA-DIMM0
    Bank Locator: BANK 0
```

```
    Type: DDR4

    Type Detail: Synchronous Unbuffered (Unregistered)

    Speed: 2400 MT/s

    Manufacturer: 0443

    Serial Number: 124AB741

    Asset Tag: None

    Part Number: RMSA3300MH78HBF-2666


--------- output truncated for brevity ---------
```

The output in Example 2-61 shows that the system has 32 GB of memory in two DDR4 SODIMM modules.

Additional tools can be used to display information about the system hardware and drivers. Two such tools, which are commonly available on most distros, are **dmesg** and **lspci**.

dmesg (which stands for *display message* or *driver message*) is a commonly used command that displays all messages from the kernel ring buffer. The output of this command typically contains the messages produced by the device drivers. This would be the first place to look when you suspect hardware or driver problems. Example 2-62 shows sample output of the **dmesg** command, which in this case is filtering the output for the word **usb**.

Example 2-62 *USB Device Driver Messages from the* ***dmesg*** *Command*

```
[root@localhost ~]# dmesg | grep -i usb
[    0.189199] ACPI: bus type USB registered
[    0.189213] usbcore: registered new interface driver usbfs
[    0.189220] usbcore: registered new interface driver hub
[    0.189226] usbcore: registered new device driver usb
[    0.749477] ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
[    0.749505] ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
[    0.749521] usbcore: registered new interface driver usb-storage
[    0.749539] usbcore: registered new interface driver usbserial
[    0.749543] usbcore: registered new interface driver usbserial_generic
[    0.749548] usbserial: USB Serial support registered for generic
[    0.750640] usbcore: registered new interface driver usbhid
[    0.750641] usbhid: USB HID core driver
[root@localhost ~]#
```

The **lspci** utility displays information about PCI buses in the system and devices connected to them. By default, it shows a brief list of devices. Example 2-63 shows the use of the **lspci** command with the **-tv** options to display verbose output in a tree-like format.

Example 2-63 *Verbose Tree-Like Output of the* ***lspci -tv*** *Command*

```
[root@localhost ~]# lspci -tv
-[0000:00]-+-00.0  Intel Corporation 440FX - 82441FX PMC [Natoma]
           +-01.0  Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
           +-01.1  Intel Corporation 82371AB/EB/MB PIIX4 IDE
           +-02.0  InnoTek Systemberatung GmbH VirtualBox Graphics Adapter
           +-04.0  InnoTek Systemberatung GmbH VirtualBox Guest Service
           +-05.0  Intel Corporation 82801AA AC'97 Audio Controller
           +-07.0  Intel Corporation 82371AB/EB/MB PIIX4 ACPI
           +-08.0  Intel Corporation 82545EM Gigabit Ethernet Controller (Copper)
           +-09.0  Intel Corporation 82545EM Gigabit Ethernet Controller (Copper)
           +-0a.0  Intel Corporation 82545EM Gigabit Ethernet Controller (Copper)
           \-11.0  Intel Corporation 82545EM Gigabit Ethernet Controller (Copper)
[root@localhost ~]#
```

System Logs

Most system log files, including the following, are stored in the /var/log directory:

• **lastlog**: Lists users and services, as well as the last successful login of each

• **messages**: Stores general system log messages

• **secure**: Stores user login information, including failed attempts

• **yum.log**: Stores YUM logs (YUM is covered in the next section of this chapter.)

Under the /var/log directory are other directories, such as the **httpd** directory, which contains the access and error log files for the **httpd** service. Some log files have the same name, such as secure, with a date appended to it. This indicates that the same log file is being rotated by the system. *Rotation* means periodically archiving a file by appending a date to it and then starting a fresh log file with the same name, without the appended date. Eventually, based on the system settings, rotated files are deleted.

Most log files can be viewed using any of the file view commands covered earlier in this chapter, including **cat**, **more**, **less**, **head**, or **tail**. The **tail** command is commonly used to check the most recent messages in a file, using the syntax **tail -n** {*N*} {*log_file*}, where *N* is the number of messages to view, starting from the end of the file. Another useful alternative is the command **tail -f** {*log_file*}, which runs the **tail** process in the foreground and maintains a live view of the log file and displays any changes to the log file in real time.

Some log files are binary files and cannot be viewed using regular file view utilities. An example is the lastlog file. To view this file, you use the **lastlog** command.

The default protocol used to log events in Linux is the syslog protocol, and the two services responsible for logging syslog messages are **rsyslogd** and **journald**.

The first service responsible for receiving syslog messages and populating the corresponding log files is **rsyslogd**. The configuration file for **rsyslogd** is /etc/rsyslog.conf, which is shown in Example 2-64.

Example 2-64 *Default Content of the rsyslog.conf File*

```
[root@localhost ~]# cat /etc/rsyslog.conf
# rsyslog configuration file


# For more information see /usr/share/doc/rsyslog-*/rsyslog_conf.html
# If you experience problems, see http://www.rsyslog.com/doc/troubleshoot.html


#### MODULES ####


# The imjournal module bellow is now used as a message source instead of imuxsock.
$ModLoad imuxsock # provides support for local system logging (e.g. via logger command)
$ModLoad imjournal # provides access to the systemd journal
#$ModLoad imklog # reads kernel messages (the same are read from journald)
#$ModLoad immark  # provides --MARK-- message capability


# Provides UDP syslog reception
#$ModLoad imudp
#$UDPServerRun 514


# Provides TCP syslog reception
#$ModLoad imtcp
#$InputTCPServerRun 514


#### GLOBAL DIRECTIVES ####


# Where to place auxiliary files
$WorkDirectory /var/lib/rsyslog


# Use default time stamp format
$ActionFileDefaultTemplate RSYSLOG_TraditionalFileFormat
```

```
# File syncing capability is disabled by default. This feature is usually not required,

# not useful and an extreme performance hit

#$ActionFileEnableSync on


# Include all config files in /etc/rsyslog.d/

$IncludeConfig /etc/rsyslog.d/*.conf


# Turn off message reception via local log socket;

# local messages are retrieved through imjournal now.

$OmitLocalLogging on


# File to store the position in the journal

$IMJournalStateFile imjournal.state


#### RULES ####


# Log all kernel messages to the console.

# Logging much else clutters up the screen.

kern.*                                          /dev/console


# Log anything (except mail) of level info or higher.

# Don't log private authentication messages!

*.info;mail.none;authpriv.none;cron.none            /var/log/messages


--------- Output truncated for brevity ---------
```

The configuration file is split into sections, and each section covers one aspect of the **rsyslogd** configuration.

The section titled "Include all config files in /etc/rsyslog.d/" (highlighted in Example 2-64) configures **rsyslogd** to read any file with a .conf extension in the /etc/rsyslog.d/ directory and load any configuration in that file. In other words, for any extra configuration required, you can write this configuration to a .conf file and save it under the /etc/rsyslog.d/ directory.

The section titled "RULES" (also highlighted in Example 2-64) instructs **rsyslogd** to redirect syslog messages to specific files based on each message's facility and/or severity. The facility of a syslog message indicates where the message originated. For example, all syslog messages from the kernel have kern listed as the facility. Under the "RULES" section in the configuration file, you can see that any message with the facility kern and *any* severity, represented by the string **kern.***, will be directed to the file /dev/console, and any message with severity **info** from any facility, represented by the string ***.info**, will be directed to file /var/log/messages.

To disable any configuration in the file without deleting it, you use **#** to comment out the configuration.

To check the status of the rsyslogd

Let's now examine a specific syslog message. In the **message** log file in Example 2-65, the first part of the message is the time stamp, followed by the host name, then the service or process that generated the message, and finally the message content.

Example 2-65 *Sample Syslog Message*

```
Apr  9 13:01:30 localhost systemd: Started Fingerprint Authentication Daemon.
```

The second service that receives syslog messages is the **journald** service. The difference between **journald** and **rsyslogd** is that **journald** receives and stores all syslog messages for all facilities and severities in one file, named system.journal. The file resides in the directory **/run/log/journal/**{*arbitrary_number*}. This file is a binary file, which means it cannot be viewed using the regular utilities such as **cat** and **tail**; instead, the log file can be viewed using the **journalctl** command. By default the contents of the log file are wiped out with every reboot of the system.

Example 2-66 shows part of the log file stored by **journald**.

Example 2-66 *Syslog Messages Stored by **journald** in system.journal*

```
-- Logs begin at Mon 2018-04-09 09:22:55 +03, end at Mon 2018-04-09 15:24:39 +03. --

Apr 09 09:22:55 localhost.localdomain systemd-journal[106]: Runtime journal is using 8.0M (max allowed 391.1

Apr 09 09:22:55 localhost.localdomain kernel: Initializing cgroup subsys cpuset
```

```
Apr 09 09:22:55 localhost.localdomain kernel: Initializing cgroup subsys cpu

Apr 09 09:22:55 localhost.localdomain kernel: Initializing cgroup subsys cpuacct

Apr 09 09:22:55 localhost.localdomain kernel: Linux version 3.10.0-693.21.1.el7.x86_64 (builder@kbuilder.dev

Apr 09 09:22:55 localhost.localdomain kernel: Command line: BOOT_IMAGE=/vmlinuz-3.10.0-693.21.1.el7.x86_64 r

Apr 09 09:22:55 localhost.localdomain kernel: e820: BIOS-provided physical RAM map:

Apr 09 09:22:55 localhost.localdomain kernel: BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable


--------- output truncated for brevity ---------
```

The following are several variations of the **journalctl** command that you can use to make the output more relevant to your requirements:

• **journalctl -n** {*N*}: Displays the last *N* log messages in the log file

• **journalctl -p** {*severity*}: Displays messages with a particular severity

• **journalctl --since** {**today|yesterday**}: Displays messages logged today or logged since yesterday

• **journalctl --since** {*date/time_1*} **--until** {*date/time_2*}: Displays messages logged from *date/time_1* through *date/time_2*

• **journalctl -f**: Displays a live view of the log file

Installing and Maintaining Software on Linux

There are several ways to install and maintain programs on Linux, from manual compilation and installation of source code to using high-level package managers. Package management constitutes one of the most notable differences between Linux distros. In keeping with the rest of this chapter, this section covers software and package management for the Red Hat family, so it applies to all distros branching out from Red Hat, such as CentOS and Fedora. The same concepts apply to other distros, but the file formats and the package management programs may be different.

The following are some important components of software and package management on Linux:

• **Source code**: Source code is a list of human-readable instructions that a programmer writes. Source code is run through a compiler to turn it into machine code that an OS can understand and execute.

• **Compile**: Compilation is the process of converting high-level source code into machine code that can be passed to the Linux kernel for direct execution without the need for other software.

• **Package management software**: Package management software is used to install, remove, or update programs on an operating system without the need to manually compile the source code of the program and then place each of the compiled files in its respective directory for correct program execution. Recall the **tar** archiving utility. Typically all files required to install a certain application on the computer are grouped into one archive file called a *package*. Package managers verify the integrity of a package, de-archive the package, and, at a minimum, identify software dependencies required to install the package. Many package managers also resolve these dependencies automatically. The package manager then places each file from the archive into the correct directory for proper program execution and, finally, updates any configuration files as required. A package manager also maintains a database that tracks the installed software so that the process of uninstalling software from the system is as automated as the process of installing it.

• **Software dependencies**: When one software or program requires that another software or program be installed first, before that first software can be installed, there is a *software dependency* between the programs. The real complication with software dependencies arises when one program has a dependency on a number of other programs that, in turn, each, have their own dependencies, and so forth. A package includes its list of dependencies in the metadata from which the package manager gets the information it requires to flag, and possibly resolve, these dependencies.

• **RPM**: RPM a recursive acronym and stands for RPM Package Manager; it was previously known as Red Hat Package Manager before other distros besides Red Hat started using it. The archive files managed by this manager have the .rpm file extension. RPM has some limitations, the most significant being the inability to resolve software dependencies (although it can detect them and alert the user).

• **YUM**: YUM, which stands for Yellowdog Updater Modified, is a high-level package manager that is based on the RPM file format. YUM is different from RPM in that it automatically resolves dependencies by downloading and installing all the software required to complete a program installation. Whereas RPM acts directly on .rpm files, YUM uses software repositories to download packages and all their dependencies.

• **DNF**: DNF, which stands for Dandified YUM, is a next-generation YUM package manager. DNF introduces several enhancements over YUM and is the default package manager on Fedora and CentOS 8.

• **Depsolve**: This is the dependency resolution module for YUM. When *depsolve* or *depsolving* is a part of a message you receive while using a package manager, you know the message is related to software dependency resolution.

• **Libsolv**: This is an external dependency resolver used by DNF.

• **Software repository**: Also known as a *repo* for short, a software repository is a location on a local disk, on a network, or on the Internet that contains a group of packages, either for a specific software, programming language, or operating system. Simply put, a repo is a collection of RPM files. High-level package managers such as YUM and DNF search all configured repos for packages that the user wishes to install. The package manager also automatically searches those repos for packages required for resolving software dependencies. A local repo exists on the system as a file with a .repo extension. The repos configured on the system are typically listed in the /etc/yum.repos.d/ directory.

Understanding each of the items in this list and where it fits in the puzzle will make the rest of this section easier to digest. The rest of the

chapter builds on these terms, providing a more comprehensive understanding of software maintenance on Linux systems.

Manual Compilation and Installation

Before the advent of package managers, the only way to install programs on a Linux OS was by downloading the source code, compiling it, and then placing each compiled file in the correct directory. Configuration files then had to be amended manually, such as adding paths to the **PATH** environment variable. This process was usually different for different programs, and the process was provided by the application developer as a custom installation process, typically in a readme file. The process of manual compilation and installation typically involves four steps:

Step 1. Download and extract the archive files that make up the program installation package. The archive files are commonly provided as tarballs.

Step 2. Run the **configure** script provided by the application developer as part of the application installation package. The **configure** script detects the specific hardware and software it is to run on. The script then creates a file named makefile that contains all required instructions to compile and install the program on that particular system. Running the **configure** script is as simple as issuing the **./configure** command in the directory in which the tarball was extracted.

Step 3. Using the makefile generated by the **configure** script in step 3, invoke the **make** utility by using the **make** command to compile the application source code, generate new libraries, or link existing shared libraries. **make** is a Linux utility that is used to determine automatically which pieces of a large program need to be recompiled and issue the commands to recompile them.

Step 4. Issue the **make install** command under the same installation directory used in steps 2 and 3. The **install** option causes **make** to execute the "install" part of the makefile and installs the application by moving each compiled binary file generated in the previous step into its respective directory.

This manual process can still be used to install applications on a Linux system. However, this method is usually used by programmers and application developers who have specialized requirements, such as customizing open-source software before installing (or redistributing) it. For day-to-day software maintenance, Linux system administrators and users can use higher-level methods that are more streamlined and use the RPM, YUM, and DNF utilities, as described in the following subsections.

RPM

RPM is a package manager that automates the process of manual compilation and installation described in the previous section. All the compiled files that constitute a program are archived in an RPM file. When the **rpm** utility is used to install a program, the utility verifies the integrity of the RPM file and determines the software dependencies from the RPM file metadata. It then prompts the user to resolve those dependencies. When dependencies do not exist or when all dependencies have been resolved, the **rpm** utility de-archives the RPM file, places each compiled file in the correct directory, and updates all relevant configuration files.

The **rpm** utility is invoked using the **rpm** command and an option to run one of several basic modes. Table 2-1 lists some of the most commonly used basic modes, briefly describes each mode, and lists the option to use in order to invoke **rpm** in that mode.

Table 2-1 *RPM Basic Modes*

Mode	Description	Option
Query	Provide information on the package	**-q**
Verify	Verifies whether a package is installed correctly	**-V**
Install	Installs a package	**-i**
Upgrade	Upgrades a package: Installs the package if a previous version is not installed, and if a previous version is installed, it is uninstalled first	**-U**
Freshen	Upgrades a package only if a previous version exists	**-F**
Uninstall	Removes all files installed by the package	**-e**

Example 2-67 shows how to query an installed package by using the command **rpm -qi** {*package_name*}. The **-q** option indicates query mode, and the **-i** option in this case stands for information. If you omit the **-i** option, the command returns the package name only. Example 2-67 uses the **gedit** text editor program that is installed by default on CentOS to showcase the command.

Note that the first option used with the **rpm** command defines the mode. So **rpm -q** runs query mode, and **rpm -i** runs install mode. Any options that follow the mode are defined according to the mode of operation. For example, the command **rpm -qi** operates in query mode, since the first option is **-q**. The **-i** option that follows does not mean that rpm is operating in install mode; it means that the information option is specified for the query mode. As noted previously, if you omit the **-i** option, the command returns the package name only.

Example 2-67 *Querying the **gedit** Package by Using **rpm -qi gedit***

```
[NetProg@localhost ~]$ rpm -q gedit

gedit-3.22.0-3.el7.x86_64

[NetProg@localhost ~]$ rpm -qi gedit

Name        : gedit

Epoch       : 2

Version     : 3.22.0

Release     : 3.el7
```

```
Architecture: x86_64
Install Date: Wed 20 Dec 2017 04:57:34 PM +03
Group       : Unspecified
Size        : 14304200
License     : GPLv2+ and GFDL
Signature   : RSA/SHA256, Thu 10 Aug 2017 07:03:22 PM +03, Key ID 24c6a8a7f4a80eb5
Source RPM  : gedit-3.22.0-3.el7.src.rpm
Build Date  : Mon 07 Aug 2017 06:34:42 AM +03
Build Host  : clbm.rdu2.centos.org
Relocations : (not relocatable)
Packager    : CentOS BuildSystem <http://bugs.centos.org>
Vendor      : CentOS
URL         : https://wiki.gnome.org/Apps/Gedit
Summary     : Text editor for the GNOME desktop
Description :
gedit is a small, but powerful text editor designed specifically for
the GNOME desktop. It has most standard text editor functions and fully
supports international text in Unicode. Advanced features include syntax
highlighting and automatic indentation of source code, printing and editing
of multiple documents in one window.

gedit is extensible through a plugin system, which currently includes
support for spell checking, comparing files, viewing CVS ChangeLogs, and
adjusting indentation levels. Further plugins can be found in the
gedit-plugins package.
[NetProg@localhost ~]$
```

In order to query a package that is *not installed*, you add the **-p** option to the **-qi** options used in Example 2-67. The **-p** option instructs the **rpm** utility to extract the information from the RPM package itself, so whether it is installed or not is irrelevant. The Google Chrome package is downloaded from the Internet and queried in Example 2-68 without being installed.

Example 2-68 *Querying the Google Chrome Package by Using **rpm -qpi {rpm_file}***

```
[NetProg@localhost Downloads]$ ls
google-chrome-stable_current_x86_64.rpm
[NetProg@localhost Downloads]$ rpm -qi google-chrome-stable_current_x86_64.rpm
package google-chrome-stable_current_x86_64.rpm is not installed
[NetProg@localhost Downloads]$ rpm -qpi google-chrome-stable_current_x86_64.rpm
warning: google-chrome-stable_current_x86_64.rpm: Header V4 DSA/SHA1 Signature, key ID 7fac5991: NOKEY
Name        : google-chrome-stable
Version     : 65.0.3325.181
Release     : 1
Architecture: x86_64
Install Date: (not installed)
Group       : Applications/Internet
Size        : 188171280
License     : Multiple, see https://chrome.google.com/
Signature   : DSA/SHA1, Tue 20 Mar 2018 08:25:11 AM +03, Key ID a040830f7fac5991
Source RPM  : google-chrome-stable-65.0.3325.181-1.src.rpm
Build Date  : Tue 20 Mar 2018 08:01:42 AM +03
Build Host  : lin64-4-m0.official.chromium.org
```

```
Relocations : /opt

Packager    : Chrome Linux Team <chromium-dev@chromium.org>

Vendor      : Google Inc.

URL         : https://chrome.google.com/

Summary     : Google Chrome

Description :

The web browser from Google



Google Chrome is a browser that combines a minimal design with sophisticated technology to make the web faster, safer, and
easier.

[NetProg@localhost Downloads]$
```

Two more options that can be used in query mode are **-ql**, which lists all files installed by that package (including the full path), and **-qR**, which lists all other packages on which this package has a dependency. Adding the **-p** option to either of these options provides the same information for a package that is *not installed*. Issuing the **rpm** command with the **-qa** option (without any arguments) queries (and lists) all installed packages on the system.

To install a package, use the command **rpm -i** {*package_name*}. If the package or an older version of the package is already installed on the system, the command fails to execute and returns an error. The **rpm -U** {*package_name*} command updates the package; that is, it installs the package, whether an older version exists or not. If an older version does not exist, a fresh installation is done. If an older version is already installed, the old version is updated to the newer version being installed. The **rpm -F** {*package_name*} command freshens the package; that is, it installs the package only if an older version is already installed. Otherwise, the command execution fails and returns an error. The **-v** option, which stands for *verbose*, provides you with extra information on the installation process.

Example 2-69 shows an attempt to install the Google Chrome browser using the **rpm -i** {*package_name*} command. Recall that **rpm** tests for dependencies and alerts the user to them but does not actually resolve them. In this case, the installation fails because, for Google Chrome to work, two additional packages need to be installed first. One of the dependencies, libXss.so.1()(64bit), is a shared library, as indicated by the .so in the name. The RPM for this library is downloaded and installed with no issues. The RPM for the second dependency, /usr/bin/lsb_release, is also downloaded, but during an installation attempt, a failed dependency message indicates that this second dependency has two dependencies of its own and that two additional packages need to be installed before it is possible to proceed any further. This situation, where one dependency has one or more dependencies, and those dependencies, in turn, have their own dependencies, is referred to as *dependency hell*. This is why we referred to this as *an attempt* to install Google Chrome; the installation was not completed due to the complexity of dependency resolution.

Note that an RPM package can contain an arbitrary set of files. Most RPM files are *binary RPMs* (*BRPMs*) that contain compiled versions of software. There are also *source RPMs* (*SRPMs*) that contain the source code used to build a binary package. These have an appropriate tag in the file header that distinguishes them from normal BRPMs, causing them to be extracted to /usr/src on installation.

Example 2-69 *Installing the Google Chrome Package by Using* ***rpm -i {****rpm_file****}***

```
[NetProg@localhost Downloads]$ rpm -i google-chrome-stable_current_x86_64.rpm

warning: google-chrome-stable_current_x86_64.rpm: Header V4 DSA/SHA1 Signature, key ID 7fac5991: NOKEY

error: Failed dependencies:

    /usr/bin/lsb_release is needed by google-chrome-stable-65.0.3325.181-1.x86_64

    libXss.so.1()(64bit) is needed by google-chrome-stable-65.0.3325.181-1.x86_64


! Dependencies are downloaded and an attempt is made to install them

[NetProg@localhost Downloads]$ ls -l

total 50932

-rw-rw-r--. 1 NetProg NetProg 52087337 Mar 21 09:48 google-chrome-stable_current_x86_64.rpm

-rw-rw-r--. 1 NetProg NetProg    24120 Mar 23 21:50 libXScrnSaver-1.2.2-6.1.el7.x86_64.rpm

-rw-rw-r--. 1 NetProg NetProg    38428 Mar 23 21:52 redhat-lsb-core-4.1-27.el7.centos.1.x86_64.rpm


! The first package in the dependency list is installed fine

[NetProg@localhost Downloads]$ sudo rpm -iv libXScrnSaver-1.2.2-6.1.el7.x86_64.rpm

warning: libXScrnSaver-1.2.2-6.1.el7.x86_64.rpm: Header V3 RSA/SHA256 Signature, key ID f4a80eb5: NOKEY

Preparing packages...

libXScrnSaver-1.2.2-6.1.el7.x86_64


! The second package in the dependency list: installation fails because it has a dependency of its own
```

```
[NetProg@localhost Downloads]$ sudo rpm -iv redhat-lsb-core-4.1-27.el7.centos.1.x86_64.rpm
warning: redhat-lsb-core-4.1-27.el7.centos.1.x86_64.rpm: Header V3 RSA/SHA256 Signature, key ID f4a80eb5: NOKEY
error: Failed dependencies:
    redhat-lsb-submod-security(x86-64) = 4.1-27.el7.centos.1 is needed by redhat-lsb-core-4.1-27.el7.centos.1.x86_64
    spax is needed by redhat-lsb-core-4.1-27.el7.centos.1.x86_64
[NetProg@localhost Downloads]$
```

YUM

Unlike RPM, which acts on RPM files directly, YUM is a package manager that utilizes software repositories. In this context, a software repo is simply a collection of RPM files. A centralized repo provides highly available, secure, consistent, and efficient means to download and install software. Consider, for example, a company with hundreds of hosts running on Linux. All the hosts have access to a central server cluster on which the software repo resides. Downloading from a software repo does not require root privileges on the server hosting the repo; therefore, administrative access to the repo is restricted, and the repo is secure. All hosts accessing the repo have access to the same versions of the same RPM archives; therefore, software consistency is maintained throughout. The process of configuring a repo on a machine is a one-time process, after which the configured repos are searched automatically for software when software installation is initiated on the local host. Moreover, a script can be configured locally on each server to download and update the software on the local host at preset times of day, which keeps all company servers updated and patched regularly. A software repo can be configured on a local machine, on a server on the network, or on a publicly available server over the Internet.

YUM also detects and automatically resolves dependencies by recursively reading RPM archive metadata, then searching for and downloading all required dependencies, and then performing the same actions for all identified dependencies until all dependencies are resolved. YUM searches for packages in all configured repositories.

You invoke YUM by using the **yum** command, which requires root privileges. Information may be retrieved using one of the command modes listed in Table 2-2. Note that *REGEX* in the table refers to *regular expression*.

Note

If you are a network engineer, you have probably worked with regular expressions before. If not, do not worry as regular expressions are covered in some detail in Chapter 4. For now, you need to know that a regular expression is a sequence of symbols and characters that is used to represent one or more strings. It is typically used in programming when a single expression needs to represent several different strings that follow the same set of rules. For example, the regular expression **^Net[0-9].$** would match on the strings **Net5Q** and **Net8X** but not **QNet2E**, **Net9DP**, or **NetPS**. In regular expressions, the caret (**^**) stands for the beginning of a line, and the dollar sign (**$**) stands for the end of a line. Two numbers separated by a hyphen (**-**) and enclosed in brackets represent a single number out of this range (inclusive). The dot (**.**) represents any single character. Therefore, the regular expression **^Net[0-9].$** represents any string that starts with **Net**, since the **^** sign represents the start of the line. This should be followed by any single number from zero to nine represented by the range in the brackets. The word that matches this regular expression ends in any single character represented by the dot. Nothing should follow, since the **$** sign represents the end of the line. **QNet2E**, **Net9DP**, and **NetPS** would not match because **QNet2E** violates the beginning-of-the-line rule with the **Q** at the beginning, **Net9DP** violates the end-of-the-line rule with the extra **P** at the end, and **NetPS** does not have a single number from the zero to nine range after **Net**.

Table 2-2 *YUM Commands to Query and Retrieve Package Information*

Command	Description
yum list	Lists all installed and available packages on the system
yum list {*REGEX*}	Lists all installed and available packages whose names match the regular expression *REGEX*
yum list installed	Lists all installed packages
yum list available	Lists all available packages
yum search {*REGEX*}	Lists all packages that have the expression whose names or summary fields match the regular expression *REGEX*, and if no results are returned, the description and URL of the packages are searched
yum search all {*REGEX*}	Works the same as **yum search** {*REGEX*}, but all four fields are searched without waiting for the search to fail on the first two fields, so it typically returns more results
yum info {*REGEX*}	Returns detailed information on packages whose names match *REGEX*. The returned information include the package name, architecture, version, size, repo, summary, and description
yum provides {*file_name*}	Returns the package that provides this file
yum group info {*group_name*}	Provides information for this group
yum history	Provides a list of the past YUM transactions
yum history info {*transaction_id*}	Provides detailed information on a specific transaction in the history

Example 2-70 shows the output of the **yum list gedit** command, which displays all installed and available packages that have **gedit** in the

package name.

Example 2-70 *Output from the* ***yum list*** *Command*

```
[NetProg@localhost ~]$ yum list gedit
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirror.airenetworks.es
 * extras: mirror.airenetworks.es
 * updates: mirror.airenetworks.es
Installed Packages
gedit.x86_64          2:3.22.0-3.el7                    @anaconda
Available Packages
gedit.i686            2:3.22.0-3.el7                    base
[NetProg@localhost ~]$
```

The output in this example shows that there is one installed package, and there is one package that is available but not installed.

When the package name is not known but something *is* known about the package, you can use the command **yum search** {*REGEX*} to search for packages that have a specific expression in the package name or summary. Example 2-71 shows how to search for all packages that have the phrase **text editor** in either the name or the summary of the package.

Example 2-71 *Output from the* ***yum search*** *Command*

```
[NetProg@localhost ~]$ yum search 'text editor'
Loaded plugins: fastestmirror, langpacks
Determining fastest mirrors
 * base: centoss5.centos.org
 * extras: centos.aumix.net
 * updates: centose5.centos.org
========================= N/S matched: text editor =========================
emacs.x86_64 : GNU Emacs text editor
emacs-nox.x86_64 : GNU Emacs text editor without X support
gedit.i686 : Text editor for the GNOME desktop
gedit.x86_64 : Text editor for the GNOME desktop
gedit-devel.i686 : Support for developing plugins for the gedit text editor
gedit-devel.x86_64 : Support for developing plugins for the gedit text
                   : editor
kate.x86_64 : Advanced Text Editor
kdelibs-ktexteditor.i686 : KDE4 Text Editor component library
kdelibs-ktexteditor.x86_64 : KDE4 Text Editor component library
kwrite.x86_64 : Text Editor
nano.x86_64 : A small text editor
perl-Syntax-Highlight-Engine-Kate.noarch : Port to Perl of the syntax
     ...: highlight engine of the Kate text editor
sed.x86_64 : A GNU stream text editor


  Name and summary matches only, use "search all" for everything.
[NetProg@localhost ~]$
```

If the search returns no results, the expression used in the command is used to match on the description or URL of the package. To search all four fields without waiting for the search to return no values on the first two fields, you use the command **yum search all** {*REGEX*}.

When you need detailed information about a package, you can use the command **yum info** {*REGEX*}. Example 2-72 shows output from the command **yum info gedit**. It displays detailed information for the two packages found that have **gedit** in their name. Note that in this case, the two packages displayed provide the same application, **gedit**, but each package is for a different architecture. Therefore, only one of them is installed, and the other is available.

Example 2-72 *Output from the* ***yum info*** *Command*

```
[NetProg@localhost ~]$ yum info gedit
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirror.airenetworks.es
 * extras: mirror.airenetworks.es
 * updates: ct.mirror.garr.it
Installed Packages
Name        : gedit
Arch        : x86_64
Epoch       : 2
Version     : 3.22.0
Release     : 3.el7
Size        : 14 M
Repo        : installed
From repo   : anaconda
Summary     : Text editor for the GNOME desktop
URL         : https://wiki.gnome.org/Apps/Gedit
License     : GPLv2+ and GFDL
Description : gedit is a small, but powerful text editor designed specifically for
            : the GNOME desktop. It has most standard text editor functions and fully
            : supports international text in Unicode. Advanced features include syntax
            : highlighting and automatic indentation of source code, printing and editing
            : of multiple documents in one window.
            :
            : gedit is extensible through a plugin system, which currently includes
            : support for spell checking, comparing files, viewing CVS ChangeLogs, and
            : adjusting indentation levels. Further plugins can be found in the
            : gedit-plugins package.


Available Packages
Name        : gedit
Arch        : i686
Epoch       : 2
Version     : 3.22.0
Release     : 3.el7
Size        : 2.4 M
Repo        : base/7/x86_64
Summary     : Text editor for the GNOME desktop
URL         : https://wiki.gnome.org/Apps/Gedit
License     : GPLv2+ and GFDL
Description : gedit is a small, but powerful text editor designed specifically for
            : the GNOME desktop. It has most standard text editor functions and fully
            : supports international text in Unicode. Advanced features include syntax
            : highlighting and automatic indentation of source code, printing and editing
            : of multiple documents in one window.
            :
```

```
            : gedit is extensible through a plugin system, which currently includes
            : support for spell checking, comparing files, viewing CVS ChangeLogs, and
            : adjusting indentation levels. Further plugins can be found in the
            : gedit-plugins package.
[NetProg@localhost ~]$
```

To install a package on the system, you use the command **yum install** {*package_name*}. The command **sudo** needs to precede that if the user doing the installation is not the user root.

In Example 2-73, the Apache web server is installed on the system. First, assuming that the package name of the Apache server is not known, **yum search 'http server'** is used to list the packages that have the phrase **http server** in the package summary. Then **yum info httpd.x86_64** is used to display complete information for package httpd.x86_64. From the summary and description, you can see that **httpd** is in fact the Apache web server package, or just Apache for short. Once the package is identified to be the correct one, **sudo yum install httpd** is used to install it.

Example 2-73 *Installing* ***httpd*** *by Using* ***sudo yum install httpd***

```
! Search for a package containing the phrase 'http server' in its summary
[NetProg@localhost ~]$ yum search 'http server'
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirror.airenetworks.es
 * extras: mirror.airenetworks.es
 * updates: mirrors.prometeus.net
====================== N/S matched: http server ================================
httpd.x86_64 : Apache HTTP Server
httpd-devel.x86_64 : Development interfaces for the Apache HTTP server
httpd-manual.noarch : Documentation for the Apache HTTP server
httpd-tools.x86_64 : Tools for use with the Apache HTTP Server
mod_auth_openidc.x86_64 : OpenID Connect auth module for Apache HTTP Server
mod_ldap.x86_64 : LDAP authentication modules for the Apache HTTP Server
mod_nss.x86_64 : SSL/TLS module for the Apache HTTP server
mod_proxy_html.x86_64 : HTML and XML content filters for the Apache HTTP Server
mod_revocator.x86_64 : CRL retrieval module for the Apache HTTP server
mod_security.x86_64 : Security module for the Apache HTTP Server
mod_session.x86_64 : Session interface for the Apache HTTP Server
mod_ssl.x86_64 : SSL/TLS module for the Apache HTTP Server
perl-HTTP-Daemon.noarch : Simple HTTP server class

  Name and summary matches only, use "search all" for everything.

! Display detailed information for the package
[NetProg@localhost ~]$ yum info httpd.x86_64
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirror.airenetworks.es
 * extras: mirror.airenetworks.es
 * updates: mirrors.prometeus.net
Available Packages
Name        : httpd
Arch        : x86_64
Version     : 2.4.6
Release     : 67.el7.centos.6
```

```
Size        : 2.7 M
Repo        : updates/7/x86_64
Summary     : Apache HTTP Server
URL         : http://httpd.apache.org/
License     : ASL 2.0
Description : The Apache HTTP Server is a powerful, efficient, and extensible
            : web server.


! Install the package
[NetProg@localhost ~]$ sudo yum install httpd
[sudo] password for NetProg:
Loaded plugins: fastestmirror, langpacks
base                                                     | 3.6 kB  00:00:00
extras                                                   | 3.4 kB  00:00:00
updates                                                  | 3.4 kB  00:00:00
(1/4): base/7/x86_64/group_gz                            | 156 kB  00:00:04
(2/4): extras/7/x86_64/primary_db                        | 184 kB  00:00:04
(3/4): base/7/x86_64/primary_db                          | 5.7 MB  00:00:24
(4/4): updates/7/x86_64/primary_db                       | 6.9 MB  00:00:32
Loading mirror speeds from cached hostfile
 * base: mirror.crazynetwork.it
 * extras: mirror.airenetworks.es
 * updates: mirror.crazynetwork.it
Resolving Dependencies
--> Running transaction check
---> Package httpd.x86_64 0:2.4.6-67.el7.centos.6 will be installed
--> Processing Dependency: httpd-tools = 2.4.6-67.el7.centos.6 for package: httpd-2.4.6-67.el7.centos.6.x86_64
--> Processing Dependency: /etc/mime.types for package: httpd-2.4.6-67.el7.centos.6.x86_64
--> Running transaction check
---> Package httpd-tools.x86_64 0:2.4.6-67.el7.centos.6 will be installed
---> Package mailcap.noarch 0:2.1.41-2.el7 will be installed
--> Finished Dependency Resolution

Dependencies Resolved
================================================================================
 Package            Arch          Version                      Repository      Size
================================================================================
Installing:
 httpd              x86_64        2.4.6-67.el7.centos.6        updates          2.7 M
Installing for dependencies:
 httpd-tools        x86_64        2.4.6-67.el7.centos.6        updates          88 k
 mailcap            noarch        2.1.41-2.el7                 base             31 k

Transaction Summary
================================================================================
Install  1 Package (+2 Dependent packages)

Total download size: 2.8 M
```

```
Installed size: 9.6 M
Is this ok [y/d/N]: y
Downloading packages:
No Presto metadata available for base
updates/7/x86_64/prestodelta                                | 957 kB  00:00:02
(1/3): mailcap-2.1.41-2.el7.noarch.rpm                      |  31 kB  00:00:00
(2/3): httpd-tools-2.4.6-67.el7.centos.6.x86_64.rpm         |  88 kB  00:00:01
(3/3): httpd-2.4.6-67.el7.centos.6.x86_64.rpm               | 2.7 MB  00:00:07
--------------------------------------------------------------------------------
Total                                               181 kB/s | 2.8 MB  00:00:15
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  Installing : httpd-tools-2.4.6-67.el7.centos.6.x86_64                    1/3
  Installing : mailcap-2.1.41-2.el7.noarch                                 2/3
  Installing : httpd-2.4.6-67.el7.centos.6.x86_64                          3/3
  Verifying  : mailcap-2.1.41-2.el7.noarch                                 1/3
  Verifying  : httpd-2.4.6-67.el7.centos.6.x86_64                          2/3
  Verifying  : httpd-tools-2.4.6-67.el7.centos.6.x86_64                    3/3

Installed:
  httpd.x86_64 0:2.4.6-67.el7.centos.6

Dependency Installed:
  httpd-tools.x86_64 0:2.4.6-67.el7.centos.6            mailcap.noarch 0:2.1.41-2.el7

Complete!
[NetProg@localhost ~]$
```

In Example 2-73, you can see that **yum** automatically determines the dependencies for installing the Apache web server, searches the configured repos for these extra packages, downloads them, and automatically installs them as part of the installation process. This is a big change from RPM's dependency resolution process, and it would not be possible unless YUM depended on repos to get the required RPM archives.

In order to uninstall the **httpd** package, the command **sudo yum remove httpd** is used, as shown in Example 2-74.

Example 2-74 *Uninstalling **httpd** by Using **sudo yum remove httpd***

```
[NetProg@localhost ~]$ sudo yum remove httpd
Loaded plugins: fastestmirror, langpacks
Resolving Dependencies
--> Running transaction check
---> Package httpd.x86_64 0:2.4.6-67.el7.centos.6 will be erased
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package         Arch             Version                       Repository    Size
================================================================================
Removing:
 httpd           x86_64           2.4.6-67.el7.centos.6         @updates      9.4 M
```

```
Transaction Summary
================================================================================
Remove  1 Package

Installed size: 9.4 M
Is this ok [y/N]: y
Downloading packages:
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  Erasing    : httpd-2.4.6-67.el7.centos.6.x86_64                          1/1
  Verifying  : httpd-2.4.6-67.el7.centos.6.x86_64                          1/1

Removed:
  httpd.x86_64 0:2.4.6-67.el7.centos.6

Complete!
[NetProg@localhost ~]$
```

To update a package on the system, you use the command **sudo yum update** {*package_name*}, and to update all packages installed on the system, you use the command **sudo yum update**.

Let's now move on to managing repositories. You can use the command **yum repolist all** to list all repositories configured on the system. If you drop the keyword **all**, the command lists only the enabled repos, as shown in Example 2-75.

Example 2-75 *Listing Repos on CentOS 7 by Using* ***yum repolist (all)***

```
[NetProg@localhost ~]$ yum repolist
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirror.crazynetwork.it
 * extras: ct.mirror.garr.it
 * updates: mirror.crazynetwork.it
repo id                              repo name                              status
base/7/x86_64                        CentOS-7 - Base                        9,591
extras/7/x86_64                      CentOS-7 - Extras                        444
updates/7/x86_64                     CentOS-7 - Updates                     2,411
repolist: 12,446
[NetProg@localhost ~]$ yum repolist all
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirror.crazynetwork.it
 * extras: ct.mirror.garr.it
 * updates: mirror.crazynetwork.it
repo id                            repo name                            status
C7.0.1406-base/x86_64              CentOS-7.0.1406 - Base               disabled
C7.0.1406-centosplus/x86_64        CentOS-7.0.1406 - CentOSPlus         disabled
C7.0.1406-extras/x86_64            CentOS-7.0.1406 - Extras             disabled
C7.0.1406-fasttrack/x86_64         CentOS-7.0.1406 - CentOSPlus         disabled
```

```
C7.0.1406-updates/x86_64          CentOS-7.0.1406 - Updates          disabled
C7.1.1503-base/x86_64             CentOS-7.1.1503 - Base             disabled
C7.1.1503-centosplus/x86_64       CentOS-7.1.1503 - CentOSPlus       disabled
C7.1.1503-extras/x86_64           CentOS-7.1.1503 - Extras           disabled
C7.1.1503-fasttrack/x86_64        CentOS-7.1.1503 - CentOSPlus       disabled
C7.1.1503-updates/x86_64          CentOS-7.1.1503 - Updates          disabled
C7.2.1511-base/x86_64             CentOS-7.2.1511 - Base             disabled
C7.2.1511-centosplus/x86_64       CentOS-7.2.1511 - CentOSPlus       disabled
C7.2.1511-extras/x86_64           CentOS-7.2.1511 - Extras           disabled
C7.2.1511-fasttrack/x86_64        CentOS-7.2.1511 - CentOSPlus       disabled
C7.2.1511-updates/x86_64          CentOS-7.2.1511 - Updates          disabled
C7.3.1611-base/x86_64             CentOS-7.3.1611 - Base             disabled
C7.3.1611-centosplus/x86_64       CentOS-7.3.1611 - CentOSPlus       disabled
C7.3.1611-extras/x86_64           CentOS-7.3.1611 - Extras           disabled
C7.3.1611-fasttrack/x86_64        CentOS-7.3.1611 - CentOSPlus       disabled
C7.3.1611-updates/x86_64          CentOS-7.3.1611 - Updates          disabled
base/7/x86_64                     CentOS-7 - Base                    enabled: 9,591
base-debuginfo/x86_64             CentOS-7 - Debuginfo               disabled
base-source/7                     CentOS-7 - Base Sources            disabled
c7-media                          CentOS-7 - Media                   disabled
centosplus/7/x86_64               CentOS-7 - Plus                    disabled
centosplus-source/7               CentOS-7 - Plus Sources            disabled
cr/7/x86_64                       CentOS-7 - cr                      disabled
extras/7/x86_64                   CentOS-7 - Extras                  enabled:   444
extras-source/7                   CentOS-7 - Extras Sources          disabled
fasttrack/7/x86_64                CentOS-7 - fasttrack               disabled
updates/7/x86_64                  CentOS-7 - Updates                 enabled: 2,411
updates-source/7                  CentOS-7 - Updates Sources         disabled
repolist: 12,446
[NetProg@localhost ~]$
```

The number to the right of the repo status in Example 2-75 is the number of packages available through that repo. Each repo in the list has a configuration file in the directory /etc/yum.repos.d/ that you can manually edit to change the configuration of that repo.

To enable one of the disabled repos from the list in Example 2-75, you can use the command **yum-config-manager --enable** {*repo_name*}. To disable a repo, you can use the command **yum-config-manager --disable** {*repo_name*}. Alternatively, you can edit the repo configuration file so that the value of the field enabled= is changed to 0 for disabled or 1 for enabled.

To add a new repo, use the command **yum-config-manager --add-repo=**{*repo_url*}. Example 2-76 shows how to add the EPEL repo to CentOS 7. EPEL stands for *Extra Packages for Enterprise Linux*, and is a Fedora special interest group that maintains the EPEL repo. The repo contains extra packages that were originally developed for the Fedora distro but that work with other distros, including CentOS, Scientific Linux, and Red Hat. After adding the repo using its URL, the command **yum repolist** confirms that the repo has been added, is enabled, and contains 12,499 packages. Furthermore, the repo configuration file is created in the /etc/yum.repos.d/ directory.

Example 2-76 *Adding the EPEL Repo to CentOS 7*

```
[NetProg@localhost ~]$ sudo yum-config-manager --add-repo="https://dl.fedoraproject.org/pub/epel/7/x86_64/"
[sudo] password for NetProg:
Loaded plugins: fastestmirror, langpacks
adding repo from: https://dl.fedoraproject.org/pub/epel/7/x86_64/
[dl.fedoraproject.org_pub_epel_7_x86_64_]
name=added from: https://dl.fedoraproject.org/pub/epel/7/x86_64/
baseurl=https://dl.fedoraproject.org/pub/epel/7/x86_64/
enabled=1
[NetProg@localhost ~]$ yum repolist
```

```
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: centosh5.centos.org
 * extras: centosq4.centos.org
 * updates: centosh5.centos.org
repo id                                repo name                                          status
base/7/x86_64                          CentOS-7 - Base                                     9,591
dl.fedoraproject.org_pub_epel_7_x86_64_ added from: https://dl.fedoraproject.org/pub/ep 12,449
extras/7/x86_64                        CentOS-7 - Extras                                     446
updates/7/x86_64                       CentOS-7 - Updates                                  2,416
repolist: 24,902
[NetProg@localhost ~]$ ls /etc/yum.repos.d/
CentOS-Base.repo       CentOS-fasttrack.repo  CentOS-Vault.repo
CentOS-CR.repo         CentOS-Media.repo      dl.fedoraproject.org_pub_epel_7_x86_64_.repo
CentOS-Debuginfo.repo  CentOS-Sources.repo
[NetProg@localhost ~]$
```

For EPEL and some other popular repos, there are alternative ways to add a repo on the system. For example, you can install the EPEL repo by using the command **yum install epel-release**, which is an RPM package available through the default CentOS-7 - Extras repo.

Creating a local repo on your machine involves four steps:

Step 1. Create a directory on the local machine that will hold all the RPM archives that will be available through that repo.

Step 2. Create and build the repo configuration file under the /etc/yum.repos.d/ directory.

Step 3. Run the command **createrepo** {*rpm_archives_directory*} to build the repo. Building the repo using this command creates a subdirectory named repodata that contains the metadata required for the repo.

Step 4. Run the command **yum clean all** to clear the YUM cache and the command **yum update** to update the cache with the new repo data.

Example 2-77 shows how to create a local repo named LocalRepo. The command **mkdir LocalRepo** is used to create directory LocalRepo in the user's home directory. The RPM package for Google Chrome is copied to the newly created directory. Then the file LocalRepo.repo is created under the directory /etc/yum.repos.d and edited to include a minimal configuration that includes the repo name, the base URL, and the enabled and gpgcheck switches. The **echo** command is used to add additional lines to the file, and the **cat** command is used at the end to verify the final content of the file. Alternatively, the file could have been edited using **vi** or some other text editor. Due to the default permissions of both the /etc/yum.repo.d directory and all files created under it, the .repo files can only be edited by the root user—not any other user, even if it is in the root group. (You will learn more about users and groups in Chapter 3.) The command **createrepo LocalRepo** is used to build the repo. Then **yum clean all** and **yum update** are used to clear the YUM cache and then update it with the new repo information. For more information on the repo configuration file, check the man page for **yum.conf**.

Example 2-77 *Adding a Local Repo*

```
[NetProg@localhost ~]$ mkdir LocalRepo
[NetProg@localhost ~]$ cp Downloads/google-chrome-stable_current_x86_64.rpm LocalRepo/
[NetProg@localhost ~]$ ls -l | grep LocalRepo
drwxrwxr-x. 2 NetProg NetProg  53 Mar 30 16:34 LocalRepo
[NetProg@localhost ~]$ su -
Password:
Last login: Fri Mar 30 16:34:54 +03 2018 on pts/0
[root@localhost ~]# touch /etc/yum.repos.d/LocalRepo.repo
[root@localhost ~]# cd /etc/yum.repos.d/
[root@localhost yum.repos.d]# echo "[LocalRepo]" > LocalRepo.repo
[root@localhost yum.repos.d]# echo "Name=LocalRepo" >> LocalRepo.repo
[root@localhost yum.repos.d]# echo "Baseurl=file:///home/NetProg/LocalRepo" >> LocalRepo.repo
[root@localhost yum.repos.d]# echo "Enabled=1" >> LocalRepo.repo
[root@localhost yum.repos.d]# echo "Gpgcheck=0" >> LocalRepo.repo
[root@localhost yum.repos.d]# cat LocalRepo.repo
[LocalRepo]
```

```
Name=LocalRepo
Baseurl=file:///home/NetProg/LocalRepo
Enabled=1
Gpgcheck=0
[NetProg@localhost ~]$ createrepo /home/NetProg/LocalRepo/
Spawning worker 0 with 1 pkgs
Workers Finished
Saving Primary metadata
Saving file lists metadata
Saving other metadata
Generating sqlite DBs
Sqlite DBs complete
[NetProg@localhost ~]$ yum clean all
Loaded plugins: fastestmirror, langpacks
Cleaning repos: LocalRepo base extras updates
Cleaning up everything
Maybe you want: rm -rf /var/cache/yum, to also free up space taken by orphaned data from disabled or removed repos
Cleaning up list of fastest mirrors
[NetProg@localhost ~]$ yum update
Loaded plugins: fastestmirror, langpacks
LocalRepo                                                | 2.9 kB     00:00
base                                                     | 3.6 kB     00:00
extras                                                   | 3.4 kB     00:00
updates                                                  | 3.4 kB     00:00
(1/5): LocalRepo/primary_db                                | 3.6 kB   00:00
(2/5): base/7/x86_64/group_gz                              | 156 kB   00:01
(3/5): extras/7/x86_64/primary_db                          | 185 kB   00:01
(4/5): base/7/x86_64/primary_db                            | 5.7 MB   00:29
(5/5): updates/7/x86_64/primary_db                         | 6.9 MB   00:32
Determining fastest mirrors
 * base: centosq4.centos.org
 * extras: centosn4.centos.org
 * updates: centosq4.centos.org
No packages marked for update
[NetProg@localhost ~]$ yum repolist
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: centosq4.centos.org
 * extras: centosn4.centos.org
 * updates: centosq4.centos.org
repo id                          repo name                            status
LocalRepo                        LocalRepo                               1
base/7/x86_64                    CentOS-7 - Base                      9,591
extras/7/x86_64                  CentOS-7 - Extras                      446
updates/7/x86_64                 CentOS-7 - Updates                   2,416
repolist: 12,454
[NetProg@localhost ~]$
```

As you can see from the **yum repolist** output, the repo has been configured and has one file. In order to test the functionality of the new repo, the Chrome package that has been copied to the new repo is then installed. Example 2-78 uses the command **yum info google-chrome*** to list the package information. The * at the end of the command is required when you do not have the full package name. Alternatively, as discussed earlier, **yum search chrome** can be used to locate the package first. As the output in Example 2-78 shows, the package is in the repo LocalRepo. Then the **yum install google-chrome** command is used to install the package.

Example 2-78 *Installing Google Chrome from LocalRepo*

```
[NetProg@localhost ~]$ yum info google-chrome*
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: centoss5.centos.org
 * extras: centosg4.centos.org
 * updates: centost5.centos.org
Available Packages
Name        : google-chrome-stable
Arch        : x86_64
Version     : 65.0.3325.181
Release     : 1
Size        : 50 M
Repo        : LocalRepo
Summary     : Google Chrome
URL         : https://chrome.google.com/
License     : Multiple, see https://chrome.google.com/
Description : The web browser from Google
            :
            : Google Chrome is a browser that combines a minimal design with
            : sophisticated technology to make the web faster, safer, and easier.


[NetProg@localhost ~]$ sudo yum install google-chrome-stable
[sudo] password for NetProg:
Loaded plugins: fastestmirror, langpacks
LocalRepo                                                | 2.9 kB  00:00:00
Loading mirror speeds from cached hostfile
 * base: centosq4.centos.org
 * extras: centosn4.centos.org
 * updates: centosq4.centos.org
Resolving Dependencies
--> Running transaction check
---> Package google-chrome-stable.x86_64 0:65.0.3325.181-1 will be installed
--> Processing Dependency: /usr/bin/lsb_release for package: google-chrome-stable-65.0.3325.181-1.x86_64
--> Running transaction check
---> Package redhat-lsb-core.x86_64 0:4.1-27.el7.centos.1 will be installed
--> Processing Dependency: redhat-lsb-submod-security(x86-64) = 4.1-27.el7.centos.1 for package: redhat-lsb-core-4.1-
27.el7.centos.1.x86_64
--> Processing Dependency: spax for package: redhat-lsb-core-4.1-27.el7.centos.1.x86_64
--> Running transaction check
---> Package redhat-lsb-submod-security.x86_64 0:4.1-27.el7.centos.1 will be installed
---> Package spax.x86_64 0:1.5.2-13.el7 will be installed
--> Finished Dependency Resolution
```

```
Dependencies Resolved

================================================================================
 Package                      Arch      Version                   Repository    Size
================================================================================
Installing:
 google-chrome-stable         x86_64    65.0.3325.181-1           LocalRepo     50 M
Installing for dependencies:
 redhat-lsb-core              x86_64    4.1-27.el7.centos.1       base          38 k
 redhat-lsb-submod-security   x86_64    4.1-27.el7.centos.1       base          15 k
 spax                         x86_64    1.5.2-13.el7              base         260 k

Transaction Summary
================================================================================
Install  1 Package (+3 Dependent packages)

Total size: 50 M
Total download size: 50 M
Installed size: 180 M
Is this ok [y/d/N]: y
Downloading packages:
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  Installing : spax-1.5.2-13.el7.x86_64                                     1/4
  Installing : redhat-lsb-submod-security-4.1-27.el7.centos.1.x86_64        2/4
  Installing : redhat-lsb-core-4.1-27.el7.centos.1.x86_64                   3/4
  Installing : google-chrome-stable-65.0.3325.181-1.x86_64                  4/4
Redirecting to /bin/systemctl start atd.service
  Verifying  : google-chrome-stable-65.0.3325.181-1.x86_64                  1/4
  Verifying  : redhat-lsb-submod-security-4.1-27.el7.centos.1.x86_64        2/4
  Verifying  : spax-1.5.2-13.el7.x86_64                                     3/4
  Verifying  : redhat-lsb-core-4.1-27.el7.centos.1.x86_64                   4/4

Installed:
  google-chrome-stable.x86_64 0:65.0.3325.181-1

Dependency Installed:
  redhat-lsb-core.x86_64 0:4.1-27.el7.centos.1
  redhat-lsb-submod-security.x86_64 0:4.1-27.el7.centos.1
  spax.x86_64 0:1.5.2-13.el7

Complete!

[NetProg@localhost ~]$ yum info google-chrome*
Loaded plugins: fastestmirror, langpacks
google-chrome                                            |  951 B  00:00:00
google-chrome/primary                                    | 1.9 kB  00:00:01
```

```
Loading mirror speeds from cached hostfile
 * base: centoss5.centos.org
 * extras: centosg4.centos.org
 * updates: centost5.centos.org
google-chrome                                                              3/3
Installed Packages
Name        : google-chrome-stable
Arch        : x86_64
Version     : 65.0.3325.181
Release     : 1
Size        : 179 M
Repo        : installed
From repo   : LocalRepo
Summary     : Google Chrome
URL         : https://chrome.google.com/
License     : Multiple, see https://chrome.google.com/
Description : The web browser from Google
            :
            : Google Chrome is a browser that combines a minimal design with
            : sophisticated technology to make the web faster, safer, and easier.
[NetProg@localhost ~]$
```

As you can see, YUM is smart enough to search and locate the google-chrome package in the LocalRepo repo, resolve the dependencies, and search and locate those dependencies in the base repo. This means that a package could be in one repo and its dependencies in another, and YUM will take care of the whole installation process.

DNF

DNF, also known as Dandified YUM, is a next-generation YUM package manager. At the time of writing, the latest version of DNF is 2.7.5, and work on Version 3 has begun. DNF is currently the default package manager on Fedora and CentOS 8 but not on earlier versions of CentOS. DNF offers several enhancements over YUM, mainly related to performance and stability, especially with dependency resolution. DNF is not covered in detail in this section, but you can use **man dnf** or **info dnf** on a Fedora or CentOS 8 distro to check it out.

Summary

Although a lot of essential material is covered in this chapter, this chapter does not even scratch the surface of Linux administration. Our goal in this chapter is to make you feel comfortable enough to start diving into more advanced topics in Chapter 3, which covers Linux storage, security and networks. Toward this goal, this chapter covers the following topics:

- The history of Linux, its status today, and the Linux development process
- Linux distros and Linux architecture
- The Linux boot process
- The Linux Bash shell
- How to find help in Linux
- The Linux file hierarchy
- File and directory operations
- Input and output redirection
- Archiving and compression
- Linux system maintenance
- Software installation and maintenance

with Linux, where block devices are always associated with block device files. The difference between block devices and block device files is sometimes a source of confusion.

The first step in analyzing a storage and file system is getting to know the hard disks. Each hard disk and partition has a corresponding device file in the /dev directory. By listing the contents of this directory, you find the sda file for the first hard disk, and, if installed, sdb for the second hard disk, sdc for the third hard disk, and so forth. Partitions are named after the hard disk that the partition belongs to, with the partition number appended to the name. For example, the *first* partition on the *second* hard disk is named sdb1. The hard disk naming convention follows the configuration in the /lib/udev/rules.d/60-persistent-storage.rules file, and the configuration is per hard disk type (ATA, USB, SCSI, SATA, and so on). Example 3-2 lists the relevant files in the /dev directory on a CentOS 7 distro. As you can see, this system has two hard disks. The first hard disk is named sda and has two partitions—sda1 and sda2—and the second is named sdb and has three partitions—sdb1, sdb2, and sdb3.

Example 3-2 *Hard Disks and Partitions in the /dev Directory*

```
[root@localhost ~]# ls -l /dev | grep sd
brw-rw----. 1 root    disk      8,   0 Jun  8 04:55 sda
brw-rw----. 1 root    disk      8,   1 Jun  8 04:55 sda1
brw-rw----. 1 root    disk      8,   2 Jun  8 04:55 sda2
brw-rw----. 1 root    disk      8,  16 Jun  8 04:55 sdb
brw-rw----. 1 root    disk      8,  17 Jun  8 04:55 sdb1
brw-rw----. 1 root    disk      8,  18 Jun  8 04:55 sdb2
brw-rw----. 1 root    disk      8,  19 Jun  8 04:55 sdb3
```

Notice the letter b at the beginning of each line of the output in Example 3-2. This indicates a block device file. A character device file would have the letter c instead.

The command **fdisk -l** lists all the disks and partitions on a system, along with some useful details. Example 3-3 shows the output of this command for the same system as in Example 3-2.

Example 3-3 *Using the **fdisk -l** Command to Get Hard Disk and Partition Details*

```
[root@localhost ~]# fdisk -l


Disk /dev/sda: 26.8 GB, 26843545600 bytes, 52428800 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x000b4fba


   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *        2048     2099199     1048576   83  Linux
/dev/sda2         2099200    52428799    25164800   8e  Linux LVM


Disk /dev/sdb: 107.4 GB, 107374182400 bytes, 209715200 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x149c8964


   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1            2048    41945087    20971520   83  Linux
/dev/sdb2        41945088    83888127    20971520   83  Linux
/dev/sdb3        83888128   115345407    15728640   83  Linux


Disk /dev/mapper/centos-root: 23.1 GB, 23081254912 bytes, 45080576 sectors
Units = sectors of 1 * 512 = 512 bytes
```

```
Sector size (logical/physical): 512 bytes / 512 bytes

I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/mapper/centos-swap: 2684 MB, 2684354560 bytes, 5242880 sectors

Units = sectors of 1 * 512 = 512 bytes

Sector size (logical/physical): 512 bytes / 512 bytes

I/O size (minimum/optimal): 512 bytes / 512 bytes

[root@localhost ~]#
```

In addition to physical disks /dev/sda and /dev/sdb and their respective partitions, the command output in Example 3-3 lists two other disks: /dev/mapper/centos-root and /dev/mapper/centos-swap. These are two logical volumes. (Logical volumes are discussed in detail in the next section.) Notice that there is an asterisk (*) under the title Boot for partition /dev/sda1. As you may have guessed, this indicates that this is the partition on which the boot sector resides, containing the boot loader. The boot loader is the software that will eventually load the kernel image into memory during the system boot process, as you have read in Section "The Linux Boot Process" in Chapter 2.

In addition to displaying existing partition details, **fdisk** can create new partitions and delete existing ones. For example, after a third hard disk, sdc, is added to the system, the **fdisk** utility can be used to create two partitions, sdc1 and sdc2, as shown in Example 3-4.

Example 3-4 *Creating NewHard Disk Partitions by Using the **fdisk** Utility*

```
! Current status of the sdc hard disk: no partitions exist

[root@localhost ~]# fdisk -l /dev/sdc


Disk /dev/sdc: 21.5 GB, 21474836480 bytes, 41943040 sectors

Units = sectors of 1 * 512 = 512 bytes

Sector size (logical/physical): 512 bytes / 512 bytes

I/O size (minimum/optimal): 512 bytes / 512 bytes


! Using fdisk to create two new partitions on sdc

[root@localhost ~]# fdisk /dev/sdc

Welcome to fdisk (util-linux 2.23.2).


Changes will remain in memory only, until you decide to write them.

Be careful before using the write command.


Device does not contain a recognized partition table

Building a new DOS disklabel with disk identifier 0x4cd00767.


Command (m for help): m

Command action

   a   toggle a bootable flag

   b   edit bsd disklabel

   c   toggle the dos compatibility flag

   d   delete a partition

   g   create a new empty GPT partition table

   G   create an IRIX (SGI) partition table

   l   list known partition types

   m   print this menu

   n   add a new partition

   o   create a new empty DOS partition table

   p   print the partition table

   q   quit without saving changes

   s   create a new empty Sun disklabel
```

```
/dev/sdc1            2048    10487807     5242880   83  Linux

/dev/sdc2        10487808    41943039    15727616   83  Linux3

[root@localhost ~]#
```

The interactive dialogue of the **fdisk** utility is self-explanatory. After the **fdisk /dev/sdc** command is issued, you can enter **m** to see all available options. You can enter **n** to start the new partition dialogue. Note the different methods to specify the size of the partition. If you go with the default option (by simply pressing Enter), the command uses all the remaining space on the disk to create that particular partition.

Before a hard disk partition can be used to store data, the partition needs to be formatted; that is, a file system has to be created. (File systems are discussed in some detail in Chapter 2.) At the time of writing, the two most common file systems used on Linux are ext4 and xfs. A partition is formatted using the **mkfs** utility. In Example 3-5, the sdc1 partition is formatted to use the ext4 file system, and sdc2 is formatted to use the xfs file system.

Example 3-5 *Creating File Systems by Using the* ***mkfs*** *Command*

```
[root@localhost ~]# mkfs -t ext4 /dev/sdc1

mke2fs 1.42.9 (28-Dec-2013)

Filesystem label=

OS type: Linux

Block size=4096 (log=2)

Fragment size=4096 (log=2)

Stride=0 blocks, Stripe width=0 blocks

327680 inodes, 1310720 blocks

65536 blocks (5.00%) reserved for the super user

First data block=0

Maximum filesystem blocks=1342177280

40 block groups

32768 blocks per group, 32768 fragments per group

8192 inodes per group

Superblock backups stored on blocks:

        32768, 98304, 163840, 229376, 294912, 819200, 884736


Allocating group tables: done

Writing inode tables: done

Creating journal (32768 blocks): done

Writing superblocks and filesystem accounting information: done


[root@localhost ~]# mkfs -t xfs /dev/sdc2

meta-data=/dev/sdc2              isize=512    agcount=4, agsize=982976 blks

         =                       sectsz=512   attr=2, projid32bit=1

         =                       crc=1        finobt=0, sparse=0

data     =                       bsize=4096   blocks=3931904, imaxpct=25

         =                       sunit=0      swidth=0 blks

naming   =version 2              bsize=4096   ascii-ci=0 ftype=1

log      =internal log           bsize=4096   blocks=2560, version=2

         =                       sectsz=512   sunit=0 blks, lazy-count=1

realtime =none                   extsz=4096   blocks=0, rtextents=0

[root@localhost ~]#
```

To specify a file system type, you use **mkfs** with the **-t** option. Keep in mind that the command output depends on the file system type used with the command.

The final step toward making a partition usable is to mount that partition or file system. Mounting is usually an ambiguous concept to engineers who are new to Linux. As discussed in Chapter 2, the Linux file hierarchy always starts at the root directory, represented by */*, and branches down. For a file system to be accessible, it has to be *mounted* to a *mount point*—that is, attached (mounted) to the file hierarchy at a specific path in that hierarchy (mount point). The mount point is the path in the file hierarchy that the file system is attached to and through which the contents of that file system can be accessed. For example, mounting the /dev/sdc1 partition to the /Operations directory maps the content of

```
  Metadata Sequence No  3
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                2
  Open LV               0
  Max PV                0
  Cur PV                4
  Act PV                4
  VG Size               44.98 GiB
  PE Size               4.00 MiB
  Total PE              11516
  Alloc PE / Size       10240 / 40.00 GiB
  Free  PE / Size       1276 / 4.98 GiB
  VG UUID               PSi3RJ-91kc-1ZFE-oCVA-RaXC-HDh5-K0VuV3
[root@server1 ~]#
```

Example 3-13 shows the **lvdisplay** command being used to display the logical volumes that have been created. A logical volume is addressed using its full path in the /dev/ directory, as shown in the example.

Example 3-13 *Displaying Logical Volumes by Using the* ***lvdisplay*** *Command*

```
[root@server1 ~]# lvdisplay /dev/VGNetProg/LVNetAutom
  --- Logical volume ---
  LV Path                /dev/VGNetProg/LVNetAutom
  LV Name                LVNetAutom
  VG Name                VGNetProg
  LV UUID                Y09QdN-J8Fw-s3Nb-RB84-bBPs-1USv-tzMfAw
  LV Write Access        read/write
  LV Creation host, time server1, 2018-08-05 21:57:42 +0300
  LV Status              available
  # open                 0
  LV Size                10.00 GiB
  Current LE             2560
  Segments               1
  Allocation             inherit
  Read ahead sectors     auto
  - currently set to     8192
  Block device           253:2
[root@server1 ~]# lvdisplay /dev/VGNetProg/LVNetDev
  --- Logical volume ---
  LV Path                /dev/VGNetProg/LVNetDev
  LV Name                LVNetDev
  VG Name                VGNetProg
  LV UUID                Z9VRTv-CUe6-uSa8-S821-jGY5-ymKh-zsKfHZ
  LV Write Access        read/write
  LV Creation host, time server1, 2018-08-05 21:58:17 +0300
  LV Status              available
  # open                 0
  LV Size                30.00 GiB
```

```
[root@server1 ~]# mkdir /Development
[root@server1 ~]# ls /
Automation  dev          hd3   lib64  opt          root  srv  usr
bin         Development  home  media  proc         run   sys  var
boot        etc          lib   mnt    Programming  sbin  tmp

[root@server1 ~]# mount /dev/VGNetProg/LVNetAutom /Automation
[root@server1 ~]# mount /dev/VGNetProg/LVNetDev /Development/
[root@server1 ~]# df -h
Filesystem                        Size  Used Avail Use% Mounted on
/dev/mapper/centos-root            44G  6.7G   38G  16% /
devtmpfs                          3.9G     0  3.9G   0% /dev
tmpfs                             3.9G     0  3.9G   0% /dev/shm
tmpfs                             3.9G  8.8M  3.9G   1% /run
tmpfs                             3.9G     0  3.9G   0% /sys/fs/cgroup
/dev/sda1                        1014M  233M  782M  23% /boot
tmpfs                             783M   20K  783M   1% /run/user/1001
/dev/mapper/VGNetProg-LVNetAutom  9.8G   37M  9.2G   1% /Automation
/dev/mapper/VGNetProg-LVNetDev     30G   33M   30G   1% /Development
[root@server1 ~]#
```

Of course, the mounting done in Example 3-15 is not persistent. To mount both logical volumes during system boot, two entries need to be added to the /etc/fstab/ file—one entry for each LV.

You may have noticed in the output of the **df -h** command in Example 3-15 that each LV appears as a subdirectory to the directory /dev/mapper/. The *device mapper* is a kernel space driver that provides the generic function of creating mappings between different storage volumes. The term *generic* is used here because the mapper is not particularly aware of the constructs used by LVM to implement logical volumes. LVM uses the device mapper to create the mappings between a volume group and its constituent logical volumes, without the device mapper explicitly knowing that the latter is a logical volume (rather than a physical one).

The examples in this section show only the very basic functionality of LVM—that is, creating the basic building blocks for having and using logical volumes on a system. However, the real power of LVM becomes clear when you use advanced features such increasing or decreasing the size of a logical volume, without having to delete the volume and re-create it, or the several options for high availability of logical volumes. Red Hat has a 147-page document titled "Logical Volume Manager Administration" on managing logical volumes. You can check out the document for RHEL 8 at https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_logical_volumes/index.

Linux Security

Linux security is a massive and complex topic so it is important to establish the intended scope of this section early on. The purpose of this section is two-fold. The first purpose is to familiarize you with basic Linux security operations that would enable you to effectively manage your development environment without being stumped. For example, you can't execute a script unless your user on the system has the privileges to execute that script, based on the script's file permissions and your group memberships. The second purpose of this section is to show you how to accomplish a minimal level of hardening for your development environment. Using an unsecured device to run scripts that access network devices—and possibly push configuration to those devices—is not a wise thing to do. Accordingly, this section covers user, group, file, and directory security, including access control lists. This chapter also covers the Linux firewall.

User and Group Management

Linux is a multiuser operating system, which means more than one user can access a single Linux system at a time.

For a user to access a Linux system, the user's account must be configured on the system. The user will then have a username and user ID (UID). A user with an account on the system is a member of one or more groups. Each group has a group name and a group ID (GID). By default, when a user is created on the system, a new group is also created; it has the same name as the username, and this becomes the primary group of the user. A user typically has a password, and each group also has a password.

Each user has a *home* directory that contains that user's files. One way that Linux maintains user segregation and security is by maintaining permissions on files and directories and allowing users with the appropriate authorization level to set those permissions. File permissions are classified into permissions for the owner of the file, the group of the file, and everyone else. The root user and any other user with root privileges can access all resources on the system, including other users' files and directories. The root user and users with root privileges are members of a group named wheel.

You can find user information by using the command **id** {*username*}, as shown in Example 3-16 for user NetProg.

Example 3-16 *Getting User Information by Using the* ***id*** *Command*

```
[root@localhost ~]# id NetProg
```

```
uid=1001(NetProg) gid=1002(NetProg) groups=1002(NetProg),10(wheel)

[root@localhost ~]#
```

User NetProg's UID is 1001. The output in Example 3-16 shows that the user's default (primary) group has the same name as the username. User NetProg in the example is also a member of the wheel group and therefore has root privileges that can be invoked by using the **sudo** {*command*} command, where *command* requires root privileges to be executed. The number to the left of each group name is the group ID.

User information is also stored in the /etc/passwd file, and group information is stored in the /etc/group file. Hashed user passwords are stored in the file /etc/shadow, and hashed group passwords are stored in the file /etc/gshadow. Example 3-17 displays the last five entries of each of the files.

Example 3-17 *Last Five Entries from the /etc/passwd, /etc/group, /etc/shadowand /etc/gshadowFiles*

```
! Sample entries from the /etc/passwd file

[netdev@server1 ~]$ tail -n 5 /etc/passwd

netdev:x:1000:1000:Network Developer:/home/netdev:/bin/bash

vboxadd:x:970:1::/var/run/vboxadd:/bin/false

cockpit-wsinstance:x:969:969:User for cockpit-ws instances:/nonexisting:/sbin/nologin

flatpak:x:968:968:User for flatpak system helper:/:/sbin/nologin

rngd:x:967:967:Random Number Generator Daemon:/var/lib/rngd:/sbin/nologin

[netdev@server1 ~]$


! Sample entries from the /etc/group file

[netdev@server1 ~]$ tail -n 5 /etc/group

netdev:x:1000:

vboxsf:x:970:

cockpit-wsinstance:x:969:

flatpak:x:968:

rngd:x:967

[netdev@server1 ~]$


! file /etc/shadow requires root privileges to be read

[netdev@server1 ~]$ tail -n 5 /etc/shadow

tail: cannot open '/etc/shadow' for reading: Permission denied

[netdev@server1 ~]$


! Sample entries from the /etc/shadow file

[netdev@server1 ~]$ sudo tail -n 5 /etc/shadow

[sudo] password for netdev:

netdev:$6$.JUG9NvdC/NzqiYq$zpCkMR3eENFgk906PjFVLJ526qFR19L2n13rFApiyPS0lgb2FlCTjJvcldqvvE3XV9lq2fK.p3hvlEYtKciD2.:18489:0:
99999:7:::

vboxadd:!!:18473::::::

cockpit-wsinstance:!!:18473::::::

flatpak:!!:18473::::::

rngd:!!:18473::::::

[netdev@server1 ~]$


! file /etc/gshadow requires root privileges to be read

[netdev@server1 ~]$ tail -n 5 /etc/gshadow

tail: cannot open '/etc/gshadow' for reading: Permission denied

[netdev@server1 ~]$


! Sample entries from the /etc/gshadow file
```

```
[netdev@server1 ~]$ sudo tail -n 5 /etc/gshadow

netdev:!::

vboxsf:!::

cockpit-wsinstance:!::

flatpak:!::

rngd:!::

[netdev@server1 ~]$
```

Each line in the /etc/passwd file is a record containing the information for one user account. Each record is formatted as follows: *username*:*x*:*user_id*:*primary_group_id*:*user_extra_information*:*user_home_directory*:*user_default_shell*.

The /etc/passwd and /etc/group files can be read by any user on the system but can only be edited by a user with root privileges. For this reason, as a security measure, the second field in the record, which historically contained the user password hash, now shows only the letter x. The user password hashes are now maintained in the /etc/shadow file, which can only be read by users with root privileges. The same arrangement is true for the /etc/group and the /etc/gshadow files. Whenever a user does not have a password, the x is omitted. Two consecutive colons in any record indicate missing information for the respective field.

Each line in the /etc/group file is a record containing information for one group. Each record is formatted as follows: *groupname*:*x*:*group_id*:*group_members*. The last field is a comma-separated list of non-default users in the group. For example, the record for the netdev group shows all users who are members of the group netdev except the user netdev itself.

Each line in the /etc/shadow file is a record containing the password information for one user. Each record is formatted as follows: *username*:*password_hash*:*last_changed*:*min*:*max*:*warn*:*expired*:*disabled*:*reserved*.

The field *last_changed* is the number of days between January 1, 1970, and the date the password was last changed. The field *min* is the minimum number of days to wait before the password can be changed. The value 0 indicates that it may be changed at any time. The field *max* is the number of days after which the password must be changed. The value 99999 means that the user can keep the same password practically forever. The field *warn* is the number of days to send a warning to the user prior to the password expiring. The field *expired* is the number of days after the password expires before the account should be disabled. The field *disabled* is the number of days since January 1, 1970, that an account has been disabled. The last field is reserved.

Finally, each line in the /etc/gshadow file is a record that contains the password information for one group. Each record is formatted as follows: *groupname*:*group_password_hash*:*group_admins*:*group_members*. The *group_password_hash* field contains an exclamation symbol (!) if no user is allowed to access the group by using the **newgrp** command. (This command is covered later in this section.)

You use the command **useradd** {*username*} to create a new user, and the command **passwd** {*username*} to set or change the password for a user. After switching to user root by using the **su** command in Example 3-18, the **id NetDev** command is used to verify that user NetDev does not already exist. The new user NetDev is then created by issuing the command **useradd NetDev**.

Next, the example shows the **su** command being used to attempt to log in as user NetDev. Notice that although a password was requested, no password will actually work. This is because, by default, when a new user is created, a password entry is created in the /etc/shadow file, but until this password is actually set by using the **passwd** command, you cannot log in as the user because the *default* password hash in the shadow file is an invalid hash. The example shows the password being removed altogether with the command **passwd -d NetDev**. Only at this point are you able to log in without getting a password prompt. The password is then set using the command **passwd NetDev**, and a warning is displayed because the password entered was **Cisco123**. Once the password is set, it is possible to log in as the user in question. Note that creating a user also creates a home directory—in this case /home/NetDev—as shown in the output of the **pwd** command. The files /etc/passwd, /etc/group, and /etc/shadow are also updated to reflect the new user details, as shown in the example.

Example 3-18 *Creating a New User and Setting the Password*

```
[NetProg@localhost ~]$ su -

Password:

Last login: Sun Apr 15 14:26:29 +03 2018 on pts/1

[root@localhost ~]#


! Verify whether the user NetDev exists

[root@localhost ~]# id NetDev

id: NetDev: no such user

[root@localhost ~]#


! Add user NetDev and log in to it

[root@localhost ~]# useradd NetDev

[root@localhost ~]# exit

Logout

[NetProg@localhost ~]$
```

```
! Authentication will fail due to invalid "default" hash
[NetProg@localhost ~]$ su NetDev
Password:
su: Authentication failure
[NetProg@localhost ~]$

! Switch back to user root and remove the password
[NetProg@localhost ~]$ su -
Password:
Last login: Sun Apr 15 14:27:07 +03 2018 on pts/1
[root@localhost ~]# passwd -d NetDev
Removing password for user NetDev.
passwd: Success
[root@localhost ~]# exit
logout
[NetProg@localhost ~]$ su NetDev
[NetDev@localhost NetProg]$ exit
Exit
[NetProg@localhost ~]$

! Switch to user root and set the password manually then test
[NetProg@localhost ~]$ su -
Password:
Last login: Sun Apr 15 14:28:12 +03 2018 on pts/1
[root@localhost ~]# passwd NetDev
Changing password for user NetDev.
New password:
BAD PASSWORD: The password fails the dictionary check - it is based on a dictionary word
Retype new password:
passwd: all authentication tokens updated successfully.
[root@localhost ~]# exit
logout
[NetProg@localhost ~]$ su NetDev
Password:
[NetDev@localhost NetProg]$

! Check the home directory and other details for user NetDev
[NetDev@localhost NetProg]$ cd
[NetDev@localhost ~]$ pwd
/home/NetDev
[NetDev@localhost ~]$ id NetDev
uid=1002(NetDev) gid=1003(NetDev) groups=1003(NetDev)
[NetDev@localhost ~]$ tail -n 1 /etc/passwd
NetDev:x:1002:1003::/home/NetDev:/bin/bash
[NetDev@localhost ~]$ tail -n 1 /etc/group
NetDev:x:1003:
[NetDev@localhost ~]$
```

```
! Switch to user root and check file /etc/shadow

[NetDev@localhost ~]$ su -

Password:

Last login: Sun Apr 15 14:50:37 +03 2018 on pts/0

[root@localhost ~]# tail -n 1 /etc/shadow

NetDev:$6$y27JA0id$i8Wze1ShSptxy5wRS8f7fOkPeeAezo2cayDl/sqikRkYp2VseEXNrzwqDQXqvMeAqzMs2Jd./jj5fm05PK.Wi/:17636:0:99999:7:
::

[root@localhost ~]# exit

logout

[NetDev@localhost ~]$
```

A user can change her own password by simply typing **passwd** without any arguments. The user is then prompted to enter the current password and then the new password and then to confirm the new password.

To delete a user, you use the command **userdel** {*username*}. This command deletes the user from the system; to delete that user's home directory and print spool as well, you use the option **-r** with the command. You use the option **-f** to force the delete action even if the user is still logged in.

You can add groups separately from users by using the command **groupadd** {*group_name*}. You can use the option **-g** to set the GID manually instead of allowing automatic assignment of the next available GID. You delete groups by using the command **groupdel** {*group_name*}. Example 3-19 shows how to create a new group called engineers and set its GID to 1111.

Example 3-19 *Creating a New Group engineers*

```
[root@localhost ~]# tail -n 2 /etc/group

NetProg:x:1002:

NetDev:x:1003:

[root@localhost ~]# groupadd -g 1111 engineers

[root@localhost ~]# tail -n 3 /etc/group

NetProg:x:1002:

NetDev:x:1003:

engineers:x:1111:

[root@localhost ~]#
```

To delete a group, you use the command **groupdel {***group_name*}. You change a group's details by using the command **groupmod**. The command **groupmod -g** {*new_gid*} {*group_name*} changes the group gid to *new_gid*, and the command **groupmod -n** {*new_name*} {*old_name*} changes the group's name from *old_name* to *new_name*. Finally, you change the group password by using the command **gpasswd** {*group_name*}. In Example 3-20, the group engineers is changed to NetDevOps, and its GID is changed to 2222. Then its password is modified to Cisco123.

Example 3-20 *Modifying Group Details*

```
[root@localhost ~]# tail -n -1 /etc/group

engineers:x:1111:

[root@localhost ~]#


! Change the group name to NetDevOps

[root@localhost ~]# groupmod -n NetDevOps engineers

[root@localhost ~]# tail -n -1 /etc/group

NetDevOps:x:1111:

[root@localhost ~]#


! Change the gid to 2222

[root@localhost ~]# groupmod -g 2222 NetDevOps

[root@localhost ~]# tail -n -1 /etc/group

NetDevOps:x:2222:

[root@localhost ~]#
```

```
! Change the group password to Cisco123

[root@localhost ~]# gpasswd NetDevOps

Changing the password for group NetDevOps

New Password:

Re-enter new password:

[root@localhost ~]#
```

A user has one primary group and one or more secondary groups. A user's primary group is the group that the user is placed in when logging in. You modify user group membership by using the command **usermod**. To change a user's primary group, you use the syntax **usermod -g** {*primary_group*} {*username*}. To change a user's secondary group, you use the syntax **usermod -G** {*secondary_group*} {*username*}; note that this command removes all secondary group memberships for this user and adds the group *secondary_group* specified in the command. To add a user to a secondary group while maintaining his current group memberships, you use the syntax **usermod -aG** {*new_secondary_group*} {*username*}. To lock a user account, you use the option **-L** with the **usermod** command, and to unlock an account, you use the **-U** option with this command. Example 3-21 shows how to change the primary group of user NetDev from NetDev to NetOps and add the wheel group to the list of secondary groups to give the user root privileges through the **sudo** command.

Example 3-21 *Modifying User Details*

```
[root@localhost ~]# id NetDev

uid=1002(NetDev) gid=1003(NetDev) groups=1003(NetDev)

[root@localhost ~]# usermod -g NetOps NetDev

[root@localhost ~]# id NetDev

uid=1002(NetDev) gid=2222(NetOps) groups=2222(NetOps)

[root@localhost ~]# usermod -aG wheel NetDev

[root@localhost ~]# id NetDev

uid=1002(NetDev) gid=2222(NetOps) groups=2222(NetOps),10(wheel)

[root@localhost ~]#
```

Notice that when the **-g** option is used to change the primary group, the secondary group is also changed. This is because user NetDev was only a member of a single group, NetDev, and that group was both the user's primary group and secondary group. When the primary and secondary groups are different, the **-g** option changes only the primary group of the user.

File Security Management

Chapter 2 describes the output of the **ls -l** command and introduces file permissions, also known as the file mode bits. This section builds on that introduction and expands on how to manage access to files and directories by modifying their permissions. It also discusses changing the file owner (user) and group. Keep in mind that in Linux, everything is represented by a file. Therefore, the concepts discussed here have a wider scope than what seems to be obvious. Also, whenever a reference is made to a file, the same concept applies to a directory, unless explicitly stated otherwise.

Example 3-22 shows the output of **ls -l** for the NetProg home directory.

Example 3-22 *Output of the **ls -l** Command*

```
[NetProg@localhost ~]$ ls -l

total 0

drwxr-xr-x. 2 NetProg NetProg  40 Apr  9 09:41 Desktop

drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Documents

drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Downloads

drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Music

drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Pictures

drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Public

drwxrwxr-x. 2 NetProg NetProg 183 Apr  7 22:53 Scripts

drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Templates

-rw-rw-r--. 1 NetProg NetProg   0 Apr  9 17:51 Testfile.txt

drwxr-xr-x. 2 NetProg NetProg   6 Mar 31 17:34 Videos

[NetProg@localhost ~]$
```

Here is a quick recap on the file permissions: The very first bit indicates whether this is a file (-) or a directory (d). Then the following 3 bits define the permissions for the file owner. By default, the owner is the user who created the file. The following 3 bits define the permissions for the users who are members of the file group. By default, this is the primary group of the user who created the file. The last 3 bits define the

permissions for everyone else, referred to as *other*. The letter r stands for read permission, w for write permission, and x for execute permission.

The dot right after the mode bits indicates that this file has an SELinux context. SELinux is a kernel security module that defines the access rights of every user, application, process, and file on the system. SELinux then governs the interactions of these entities using a security policy, where an entity referred to as a subject attempts to access another entity referred to as an object. SELinux is an important component of Linux security but is beyond the scope of this book. When a file or a directory has a + symbol in place of the dot (.), it means the file has an access control list (ACL) applied to it. ACLs, which are covered later in this chapter, provide more granular access control to files on a per-user basis.

The output of the **ls -l** command also displays the file owner (more formally referred to as user) and the file group.

File permissions can be represented (and modified) by either using *symbolic* notation or *octal* notation.

Symbolic notation is the type of notation described so far, where user, group, and others are represented by u, g, and o, respectively, and the access permissions are write, read, and execute, represented by w, r, and x, respectively. The following syntax is used to set the file permissions: **chmod** [**u=**{*permissions*}][**,g=**{*permissions*}][**,o=**{*permissions*}] {*file_name*}.

Example 3-23 shows how to modify the file permissions for file TestFile.txt to the following:

• **User**: Read, write, and execute

• **Group**: Read and write

• **Other**: No access

Example 3-23 *Setting File Permissions by Using Symbolic Notation*

```
! Current file permissions

[NetProg@localhost ~]$ ls -l Testfile.txt

-rw-rw-r--. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt

[NetProg@localhost ~]$


! Change the file permissions as listed

[NetProg@localhost ~]$ chmod u=rwx,g=rw,o= Testfile.txt


! New file permissions

[NetProg@localhost ~]$ ls -l Testfile.txt

-rwxrw----. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt

[NetProg@localhost ~]$
```

Notice that in order to remove all permissions for one of the categories, you just leave the right side of the **=** symbol blank.

One of the challenges with the symbolic notation syntax as used in Example 3-23 is that you have to know beforehand what permissions the file already has and make sure to align the current permissions with the new permissions you are trying to set. For example, if a file already has read and write permissions set for the file group and you would like to add the execute permission, you have to know this fact prior to the change, and then you need to make sure you do not delete the already existing write or read permissions while setting the execute permission. In order to just add or remove permissions for a specific category, without explicitly knowing or considering the existing permissions, you replace the **=** symbol in the previous syntax with either a **+** or a **-** symbol, as follows: **chmod** [**u**[**+**|**-**]{*permissions*}][**,g**[**+**|**-**]{*permissions*}][**,o**[**+**|**-**]{*permissions*}] {*file_name*}.

In Example 3-24 the permissions for the file TestFile.txt are modified as follows:

• **User**: Unchanged

• **Group**: Write permission removed and execute permission added

• **Other**: Execute permission added

Notice that when using this syntax, you do not need to know what permissions the file already has. You only need to consider the changes that you want to implement.

Example 3-24 *Adding and Removing File Permissions by Using Symbolic Notation*

```
[NetProg@localhost ~]$ ls -l Testfile.txt

-rwxrw----. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt

[NetProg@localhost ~]$ chmod g-w,g+x,o+x Testfile.txt

[NetProg@localhost ~]$ ls -l Testfile.txt

-rwxr-x--x. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt

[NetProg@localhost ~]$
```

Notice that you can mix the **+** and **-** symbols in the same command and for the same category, as shown in Example 3-24 for the file group,

where **g-w** is used to remove the write permission for the group, and **g+x** is used to add the execute permission for the group.

When a certain permission has to be granted or revoked from all categories, the letter a is used to collectively mean u, g, and o. The letter a in this case stands for *all*. The letter a may be dropped altogether, and the command then applies to all categories. For example, the command **chmod +w Example.py** adds the write permission to all categories for the file Example.py.

Octal notation, on the other hand, uses the following syntax: **chmod** {*user_permission*}{*group_permission*}{*other_permission*} {*file_name*}. The *user*, *group*, and *other* categories are represented by their positions in the command. The permission granted to each category is represented as a numeric value that is equal to the summation of each permission's individual value. To elaborate, note the following permission values:

• Read=4

• Write=2

• Execute=1

To set the read permission only, you need to use the value 4; for write permission only, you use the value 2; and for execute permission only, you use the value 1. To set all permissions for any category, you need to use 4+2+1=7. To set the read and write permissions only, you need to use 4+2=6, and so forth. Example 3-25 illustrates this concept and uses octal notation to set the read, write, and execute permissions for both user and group, and set only the execute permission for the category other for file Testfile.txt.

Example 3-25 *Setting File Permissions Using Octal Notation*

```
[NetProg@localhost ~]$ ls -l Testfile.txt

-rwxr-x--x. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt

[NetProg@localhost ~]$ chmod 771 Testfile.txt

[NetProg@localhost ~]$ ls -l Testfile.txt

-rwxrwx--x. 1 NetProg NetProg 0 Apr  9 17:51 Testfile.txt

[NetProg@localhost ~]$
```

The number **7** in each of the first two positions in the command **chmod 771 Testfile.txt** represents the sum of 4, 2, and 1 and is used to set all permissions for user and group. The number **1** in the last position sets the execute only permission for other.

While octal notation looks snappier than symbolic notation, it does not provide the option of adding or removing permissions without considering the existing file permissions, as provided by the **+** and **-** symbols used with symbolic notation.

Besides modifying file and directory permissions, you can control access to a file or directory by changing the file's user and/or group through the **chown** command. The command syntax is **chown** {*user*}**:**{*group*} {*file*}. Example 3-26 shows how to change the *user* and *group* of file TestFile.txt to NetDev and networks, respectively.

Example 3-26 *Changing File User and Group by Using the **chown** Command*

```
[root@localhost ~]# ls -l /home/NetProg/Testfile.txt

-rwxrwx--x. 1 NetProg NetProg 0 Apr  9 17:51 /home/NetProg/Testfile.txt

[root@localhost ~]# chown NetDev:networks /home/NetProg/Testfile.txt

[root@localhost ~]# ls -l /home/NetProg/Testfile.txt

-rwxrwx--x. 1 NetDev networks 0 Apr  9 17:51 /home/NetProg/Testfile.txt

[root@localhost ~]#
```

You use the **-R** option (which stands for *recursive*) with both the **chmod** and the **chown** commands if the operation is being performed on a directory, and you want the changes to also be made to all subdirectories and files in that directory.

By default, any file or directory created by a user is assigned to the primary group of that user. For example, if user NetDev is in the NetOps group, any file created by user NetDev has NetDev as the file user and NetOps as the file group. You can change this default behavior by either using the **sg** command when creating the file or by logging in to another group by using the command **newgrp**. If that other group is one of the user's secondary groups, no password is required. If that other group is not one of the user's secondary groups, the user is prompted for a password.

Example 3-27 shows the default behavior when creating a file. In this case, a new file named NewFile is created by user NetDev. As expected, the file user is NetDev, and the file group is NetOps.

Example 3-27 *Default User and Group of a Newly Created File*

```
[NetDev@localhost ~]$ id NetDev

uid=1002(NetDev) gid=2222(NetOps) groups=2222(NetOps),10(wheel)

[NetDev@localhost ~]$ touch NewFile

[NetDev@localhost ~]$ ls -l NewFile

-rw-r--r--. 1 NetDev NetOps 0 Apr 17 00:59 NewFile

[NetDev@localhost ~]$
```

Example 3-28 shows how to use the **sg** command to create file NewFile_1 but under the group networks.

Example 3-28 *Using the **sg** Command to Create a File Under a Different Group*

```
[NetDev@localhost ~]$ id NetDev

uid=1002(NetDev) gid=2222(NetOps) groups=2222(NetOps),10(wheel)

[NetDev@localhost ~]$ sg networks 'touch NewFile_1'

Password:

[NetDev@localhost ~]$ ls -l NewFile_1

-rw-r--r--. 1 NetDev networks 0 Apr 17 01:03 NewFile_1

[NetDev@localhost ~]$
```

Notice that the command **touch** {*file_name*}, which itself is an argument to the **sg** command, has to be enclosed in quotes because it is a multi-word command. Notice also that because the user NetDev is not a member in the networks group, as you can see from the output of the **id** command, the user is prompted for the group password, which was set earlier by using the command **gpasswd networks**.

Alternatively, the user can log in to another group by using the command **newgrp** and create a file or directory under that group. Example 3-29 shows the user NetProg logging in to group systems and not being prompted for a password since this is one of NetProg's secondary groups. When the file NewFile_2 is created, the user of the file is NetProg, and the group is systems, not NetProg.

Example 3-29 *Using the **newgrp** Command to Log In to a Different Group*

```
[NetProg@localhost ~]$ id NetProg

uid=1001(NetProg) gid=1002(NetProg) groups=1002(NetProg),10(wheel),2224(systems)

[NetProg@localhost ~]$ newgrp systems

[NetProg@localhost ~]$ touch NewFile_2

[NetProg@localhost ~]$ ls -l NewFile_2

-rw-r--r--. 1 NetProg systems 0 Apr 17 01:15 NewFile_2

[NetProg@localhost ~]$
```

Access Control Lists

So far in this chapter, you have seen how to set file and directory access permissions for either user, or collectively for group, or other. What if you want to set those permissions individually for a specific user who is not the file owner or for a group of users who belong to a group other than the file group? File mode bits do not help in such situations. Using the file mode bits, the only user whose permissions can be changed individually is the file or directory owner (user) and the only group of users whose permissions can be changed collectively are the users who are members of the file or directory group.

Access control lists (*ACLs*) provide more granular control over file and directory access. ACLs allow a system administrator (or any other user who has root privileges) to set file and directory permissions for any user or group on the system.

Before you can configure ACLs, three prerequisites must be met:

- The kernel must support ACLs for the file system type on which ACLs will be applied.
- The file system on which ACLs will be used must be mounted with the ACL option.
- The ACL package must be installed.

Most common distros today—including CentOS 7 and Red Hat Enterprise Linux (RHEL) 7 and later versions—have these prerequisites configured by default, and you do not need to do any further configuration.

If you are running a different distro or an older version of CentOS, you can check the first prerequisite by using either the **findmnt** or **blkid** command to determine the file system type on your system. The command **findmnt** works only if the file system has been mounted, and **blkid** works whether it is mounted or not. Then you need to inspect the kernel configuration file /boot/conf-<*version.architecture*> to determine whether ACLs have been enabled for this file system type. Example 3-30 shows the relevant output for the file system on the sda1 partition.

Example 3-30 *ACL Support for the sda1 File System*

```
[root@server1 ~]# findmnt /dev/sda1

TARGET SOURCE    FSTYPE OPTIONS

/boot  /dev/sda1 xfs    rw,relatime,seclabel,attr2,inode64,noquota

[root@server1 ~]# cat /boot/config-3.10.0-693.el7.x86_64 | grep ACL

CONFIG_EXT4_FS_POSIX_ACL=y

CONFIG_XFS_POSIX_ACL=y

CONFIG_BTRFS_FS_POSIX_ACL=y

CONFIG_FS_POSIX_ACL=y

CONFIG_GENERIC_ACL=y
```

```
CONFIG_TMPFS_POSIX_ACL=y

CONFIG_NFS_V3_ACL=y

CONFIG_NFSD_V2_ACL=y

CONFIG_NFSD_V3_ACL=y

CONFIG_NFS_ACL_SUPPORT=m

CONFIG_CEPH_FS_POSIX_ACL=y

CONFIG_CIFS_ACL=y

[root@server1 ~]#
```

The kernel configuration file lists different configuration options, each followed by an **=** symbol and then the letter y, n, or m. The letter y means that this option (module) was configured as part of the kernel when the kernel was first compiled. In this example, CONFIG_XFS_POSIX_ACL=y means that the kernel supports ACLs for the xfs file system. The letter n indicates that this module was not compiled into the kernel, and the letter m means that this module was compiled as a loadable kernel module (introduced in Chapter 2).

The second prerequisite is that the partition on which the ACLs will be used has to be mounted with the ACL option. By default, on ext3/4 and xfs file systems, ACL support is enabled. In older CentOS versions and other distros where the ACL option is not enabled by default, the file system can be mounted with the ACL option by using the syntax **mount -o acl** {*partition*} {*mount_point*}. On the other hand, if the ACL option is enabled by default, and you want to disable ACL support while mounting the file system, you can use the **noacl** option with the **mount** command. As discussed in the previous section, mounting using the **mount** command is non-persistent. For persistent mounting with the ACL option, you can add an entry to the /dev/fstab file (or amend an existing entry) and add the **acl** option (right after the **defaults** keyword). The /dev/fstab file is discussed in detail earlier in this chapter.

Finally, by using the **yum info acl** command, you can confirm whether the ACL package has been installed. The **yum** command is covered in detail in Chapter 2.

When ACL support has been established, you can use the command **getfacl** {*filename*|*directory*} to display the ACL configuration for a file or directory. Example 3-31 shows the output of the **getfacl** command for the directory /Programming and then for the file NewFile.txt.

Example 3-31 *Output of the **getfacl** Command*

```
[root@localhost /]# ls -ld Programming

drwxr-xr-x. 2 root root 25 Jun  9 05:46 Programming

[root@localhost /]# ls -l Programming

total 0

-rw-r--r--. 1 root root 0 Jun  9 05:46 NewFile.txt

[root@localhost /]# getfacl Programming

# file: Programming

# owner: root

# group: root

user::rwx

group::r-x

other::r-x

[root@localhost /]# getfacl Programming/NewFile.txt

# file: Programming/NewFile.txt

# owner: root

# group: root

user::rw-

group::r--

other::r-

[root@localhost /]#
```

As you can see from the output in Example 3-31, both the directory and file are owned by the user root, and the group of both is also root. So far, there is no additional information provided by the **getfacl** command beyond what is already displayed by **ls -l**; the format is the only difference.

For the file NewFile.txt, the user NetProg is not the file owner and is not a member of the file group. As per the permissions for *other*, the user NetProg should be able to only read the file but not write to it or execute it. In Example 3-32, the user NetProg attempts to write to the file NewFile.txt by using the **echo** command, but a "Permission denied" error message is displayed. The **setfacl -m u:NetProg:rw /Programming/Newfile.txt** command grants write permission to the user NetProg. When the write operation is attempted again, it is successful due to the new elevated permissions.

Example 3-32 *Changing the Permissions for the User NetProg by Using **setfacl***

```
! Echo(write) operation fails since NetProg has no write permissions

[NetProg@localhost /]$ echo "This is a write test" > /Programming/NewFile.txt

bash: /Programming/NewFile.txt: Permission denied


! Grant user NetProg write permission (requires root permissions)

[NetProg@localhost /]$ su

Password:

[root@localhost /]# setfacl -m u:NetProg:rw /Programming/NewFile.txt

[root@localhost /]# getfacl /Programming/NewFile.txt

getfacl: Removing leading '/' from absolute path names

# file: Programming/NewFile.txt

# owner: root

# group: root

user::rw-

user:NetProg:rw-

group::r--

mask::rw-

other::r--


! Write operation now successful

[root@localhost /]# su NetProg

[NetProg@localhost /]$ echo "This is a write test" > /Programming/NewFile.txt

[NetProg@localhost /]$ cat /Programming/NewFile.txt

This is a write test

[NetProg@localhost /]$ ls -l /Programming/NewFile.txt

-rw-rw-r--+ 1 root root 21 Jun  9 07:24 /Programming/NewFile.txt

[NetProg@localhost /]$
```

Notice the **+** symbol that now replaces the dot to the right of file permission bits at the end of Example 3-32. This indicates that an ACL has been applied to this file. The new write permission has been granted to the user NetProg only, and not to any other user. This was done without amending the file permissions for the user, group, or other categories. It was also done without modifying the group memberships of the user NetProg. The same permission could also be applied to a group instead of an individual user. The level of granularity provided by ACLs should be clear by now.

The **setfacl** command used in Example 3-32was issued with the option **-m**, which is short for *modify* and is used to apply a new ACL or modify an existing ACL. To remove an ACL, you use the option **-x** instead of **-m**; the remainder of the command remains the same, except that the ACL in the command is an existing ACL that is now being removed.

In Example 3-32 you can see the three-field argument **u:NetProg:rw**. When setting an ACL for a user, the first field is u, as in the example. For a group, the first field would be g, and for other, the first field would be o. The second field is the user or group name, which is NetProg in this example. If the ACL is for other, this field remains empty. The third field is the permissions you wish to grant to the user or group.

Finally, after the three-field argument is the name of the directory or file to which the ACL is applied. Note that whether a full path or only a relative path is required depends on the current working directory relative to the location of the file or directory to which the ACL is being applied. The same rules apply here as with any other Linux command that operates on a file or directory.

Therefore, the general syntax of the **setfacl** command to add, modify, or remove an ACL is **setfacl** {**-m|-x**} {**u|g|o**}:{*username|group*}:{*permissions*} {*file|directory*}. To remove all ACL entries applied to a file, you use the option **-b** followed by the filename, omitting the three-field argument.

In Example 3-32, notice the text mask::rw- in the output of the **getfacl** command, after the ACL has been applied. The mask provides one more level of control over the permissions granted by the ACL. Say that after granting several users different permissions to a file, you decide to remove a specific permission, such as the write permission, from *all* named users. The ACL mask then comes in handy. The permissions in the mask override the permissions for all named users and the file group. For example, if the mask permissions are r-x and the user NetProg has been granted rwx permissions, that user's effective permissions are r-x after the mask is set. The effective mask permissions are applied using the command **setfacl -m m:**{*permissions*} {*filename*}. In Example 3-33, the user NetProg has permissions rw-, and so does the mask. The mask is modified to r--. Notice the effective permissions that appear on the right side of the output of the **getfacl** command after the mask has been modified. After you remove the write permission from the mask, NetProg's write attempt to the file fails.

Example 3-33 *Changing the Mask Permissions by Using* ***setfacl***

```
! Set the effective rights mask
```

```
[root@localhost /]# setfacl -m m:r /Programming/NewFile.txt

[root@localhost /]# getfacl /Programming/NewFile.txt

getfacl: Removing leading '/' from absolute path names

# file: Programming/NewFile.txt

# owner: root

# group: root

user::rw-

user:NetProg:rw-        #effective:r--

group::r--

mask::r--

other::r--


! Write operation to file by user NetProg now fails

[root@localhost /]# su NetProg

[NetProg@localhost /]$ echo "Testing mask permissions" > /Programming/NewFile.txt

bash: /Programming/NewFile.txt: Permission denied

[NetProg@localhost /]$
```

When ACLs are applied to directories, by default, these ACLs are not inherited by files and subdirectories in that directory. In order to achieve inheritance, the option **-R** has to be added to the same **setfacl** command used earlier. In Example 3-34, an ACL setting rwx permissions for the user NetProg is applied to the directory Programming. Attempting to write to file NewFile.txt under the directory by user NetProg fails because the write permission has not been inherited by the file.

Example 3-34 *ACLs Are Not Inherited by Default by Subdirectories and Files Under a Directory*

```
! Apply an acl to the /Programming directory

[root@localhost ~]# setfacl -m u:NetProg:rwx /Programming

[root@localhost ~]# getfacl /Programming

getfacl: Removing leading '/' from absolute path names

# file: Programming

# owner: root

# group: root

user::rwx

user:NetProg:rwx

group::r-x

mask::rwx

other::r-x


! The acl is not applied to NewFile.txt under the directory

[root@localhost ~]# getfacl /Programming/NewFile.txt

getfacl: Removing leading '/' from absolute path names

# file: Programming/NewFile.txt

# owner: root

# group: root

user::rw-

group::r--

other::r--


! And the write operation fails as expected

[root@localhost ~]# su - NetProg

[NetProg@localhost ~]$ echo "This is a write test" > /Programming/NewFile.txt

bash: /Programming/NewFile.txt: Permission denied
```

```
[NetProg@localhost ~]$
```

After the ACL has been removed and then reapplied in Example 3-35 using the **-R** option, the user NetProg can write to the file successfully. The **getfacl** command also shows that the ACL has been applied to the file as if the **setfacl** command had been applied to the file directly.

Example 3-35 *ACL Inheritance by Subdirectories and Files Under a Directory Using the -R Option*

```
! Clear the acl from the /Programming directory
[root@localhost ~]# setfacl -b /Programming


! Apply the acl to directory /Programming using the -R option
[root@localhost ~]# setfacl -R -m u:NetProg:rwx /Programming
[root@localhost ~]# getfacl /Programming
getfacl: Removing leading '/' from absolute path names
# file: Programming
# owner: root
# group: root
user::rwx
user:NetProg:rwx
group::r-x
mask::rwx
other::r-x


! The acl is inherited by the file NewFile.txt
[root@localhost ~]# getfacl /Programming/NewFile.txt
getfacl: Removing leading '/' from absolute path names
# file: Programming/NewFile.txt
# owner: root
# group: root
user::rw-
user:NetProg:rwx
group::r--
mask::rw-
other::r--


! And the write operation is successful
[root@localhost ~]# su - NetProg
[NetProg@localhost ~]$ echo "This is to test inheritance" > /Programming/NewFile.txt
[NetProg@localhost ~]$ cat /Programming/NewFile.txt
This is to test inheritance
[NetProg@localhost ~]$
```

It is important to remember that the ACL applied to a directory and inherited by all subdirectories and files will *not* be applied to any files created *after* the ACL has been applied. Only the files that existed before the ACL was applied will be affected.

The ACLs described so far are called access ACLs. Another type of ACLs, called default ACLs, may be used with directories (only) if the requirement is that all files and subdirectories, when created, should inherit the parent directory ACLs. The syntax for applying a default ACL is **setfacl -m d:{u|g|o}:**{*username*|*group*}**:**{*permissions*} {*directory*}. Try to experiment with default ACLs and note how newly created files inherit the directory ACL without your having to explicitly issue the **setfacl** command after the file or subdirectory has been created.

The same concepts discussed previously for a single user apply to a group when you set the ACL for a group of users other than the file or directory group by using the letter **g** along with the group name in the **setfacl** command instead of a **u** with the username.

In addition to using the **setfacl** command to set permissions for a specific user or group, you can use this command to set permissions for the file user, group, or other categories, similar to what can be accomplished using the **chmod** command as shown in the previous section. Note that if the **setfacl** command is used to apply an ACL to a file or directory, it is recommended that you *not* use **chmod**.

When a file or directory is copied or moved, ACLs are moved along with the file or directory.

Linux System Security

CentOS 7 and later versions come with a default built-in firewall service named **firewalld**. This service functions in a similar manner to a regular firewall in terms of providing *security zones* with different *trust levels*. Each zone constitutes a group of permit/deny rules for incoming traffic. Each physical interface on the server is bound to one of the firewall zones. However, **firewalld** provides only a subset of the services provided by a full-fledged firewall.

You can check the status of the **firewalld** service and start, stop, enable, and disable the service just as you would any other service on Linux by using the **systemctl** command. Example 3-36 shows the status of the **firewalld** service: In this example, you can see that it is active and enabled.

Example 3-36 *The **firewalld** Service Status*

```
[NetProg@localhost ~]$ systemctl status firewalld

• firewalld.service - firewalld - dynamic firewall daemon

   Loaded: loaded (/usr/lib/systemd/system/firewalld.service; enabled; vendor preset: enabled)

   Active: active (running) since Sat 2018-04-21 21:37:06 +03; 30min ago

     Docs: man:firewalld(1)

 Main PID: 787 (firewalld)

   CGroup: /system.slice/firewalld.service

           └─787 /usr/bin/python -Es /usr/sbin/firewalld --nofork --nopid


Apr 21 21:37:05 localhost.localdomain systemd[1]: Starting firewalld - dynamic firewall daemon...

Apr 21 21:37:06 localhost.localdomain systemd[1]: Started firewalld - dynamic firewall daemon.


--------- output truncated for brevity ---------
```

The **firewalld** service has a set of zones created by default when the service is first installed; these zones are sometimes referred to as the *base* or *predefined* zones. Custom zones can also be created and deleted. However, base zones cannot be deleted. One zone is designated as the default zone and is the zone to which all interfaces are bound, by default, unless the interface is explicitly moved to another zone. By default, the default zone is the public zone. Each zone has a set of rules attached to it and a list of interfaces bound to it. Rules and interfaces can be added to or removed from a zone.

Example 3-37 shows how to list the base zones of **firewalld** by using the command **firewall-cmd --get-zones** and how to identify the default zone by using the command **firewall-cmd --get-default-zone**.

Example 3-37 *Listing the Base and Default Zones of a Firewall*

```
[root@localhost ~]# firewall-cmd --get-zones

block dmz drop external home internal public trusted work

[root@localhost ~]# firewall-cmd --get-default-zone

public

[root@localhost ~]#
```

You can change the default zone by using the command **firewall-cmd --set-default-zone=**{*zone_name*}.

You can list the details of a zone by using the command **firewall-cmd --list-all --zone=**{*zone_name*}, as shown in Example 3-38. To list the details of the default zone, you omit the **--zone=**{*zone_name*} option.

Example 3-38 *Listing Zone Details*

```
[root@localhost ~]# firewall-cmd --list-all --zone=internal

internal

  target: default

  icmp-block-inversion: no

  interfaces:

  sources:

  services: ssh mdns samba-client dhcpv6-client

  ports:

  protocols:

  masquerade: no
```

```
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:


[root@localhost ~]# firewall-cmd --list-all
public (active)
  target: default
  icmp-block-inversion: no
  interfaces: enp0s3 enp0s9 enp0s10 enp0s8
  sources:
  services: ssh dhcpv6-client
  ports:
  protocols:
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:


[root@localhost ~]#
```

Example 3-39 shows how to add rules to the zone dmz to permit specific incoming traffic on interfaces bound to this zone. The first rule added permits traffic from the source IP address 10.10.1.0/24 by using a *source-based* rule. Then BGP traffic on TCP port 179 is permitted by using a *port-based* rule. HTTP service is then permitted by defining a *service-based* rule. Finally, interface enp0s9 is removed from the public zone and bound to the dmz zone. Notice how the rules appear when the details of the zone are listed at the end of the example.

Example 3-39 *Adding Rules to Zone* ***dmz***

```
[root@localhost ~]# firewall-cmd --list-all --zone=dmz
dmz
  target: default
  icmp-block-inversion: no
  interfaces:
  sources:
  services: ssh
  ports:
  protocols:
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:


[root@localhost ~]# firewall-cmd --zone=dmz --add-source=10.10.1.0/24
success
[root@localhost ~]# firewall-cmd --zone=dmz --add-port=179/tcp
success
[root@localhost ~]# firewall-cmd --zone=dmz --add-service=http
success
[root@localhost ~]# firewall-cmd --zone=dmz --add-interface=enp0s9
```

```
The interface is under control of NetworkManager, setting zone to 'dmz'.

success

[root@localhost ~]# firewall-cmd --zone=dmz --list-all

dmz (active)

  target: default

  icmp-block-inversion: no

  interfaces: enp0s9

  sources: 10.10.1.0/24

  services: ssh http

  ports: 179/tcp

  protocols:

  masquerade: no

  forward-ports:

  source-ports:

  icmp-blocks:

  rich rules:


[root@localhost ~]#
```

Note that in order to remove a rule, instead of using the **--add** option, you use the **--remove** option. For example, to remove the rule for TCP port 179, you use the command **firewall-cmd --zone=dmz --remove-port=179/tcp**.

Much like running and startup configurations on routers and switches, **firewalld** supports both runtime and permanent configurations. A *runtime configuration* is not persistent and is lost after a reload. A *permanent configuration* is persistent but takes effect only after a reload when the configuration has been changed. Any configuration commands that have already been used are reflected in the runtime configuration. To make a configuration permanent, you use the option **--permanent** with the command. You reload the **firewalld** service by using the command **firewall-cmd --reload**.

Linux Networking

Linux provides several methods for managing network devices and interfaces on a system. Usually, a system administrator can accomplish the same task using several different methods. A network device or an interface is managed by the kernel, and each method accesses the Linux kernel via a different path. There are three popular methods for managing Linux networking:

• Using the command-line **ip** utility

• Using the NetworkManager service

• Using network configuration files

This section covers these three methods listed. It should be fairly easy to use the help resources on your Linux distro, such as the man and info pages, to learn about any utility not covered here.

Note

Keep in mind that some commands and utilities for managing Linux networking, such as **ifconfig**, **netstat**, **arp**, and **route**, are considered legacy utilities. These utilities have not been updated for years and have been deprecated on some distros but are still available on others. Even if any of these commands are available in the distro you are using, we do not recommend using them; instead, use the methods described in this section. Basically, the way legacy utilities function, particularly how these utilities speak with the kernel, is not very efficient. You will probably run into these legacy utilities at some point while working on Linux. For example, at the time of this writing, all four legacy utilities mentioned here are still supported on the Bash shell exposed by IOS XR and NX-OS.

The ip Utility

ip is a command-line utility that is part of the **iproute2** group of utilities. It is invoked using the command **ip** [*options*]{*object*}{*action*}. This syntax is quite intuitive in that the *action* in the command indicates what action you would want to apply to an *object*. For example, the command **ip link show** applies the action **show** to the object **link**. As you may have guessed, this command displays the state of all network interfaces (links) on the system, as shown in Example 3-40. To limit the output to one specific interface, you can add **dev** {*intf*} to the end of the command, as also shown in the example.

Note

The man pages for the **ip** command refer to the *action* part in the previous syntax as *command*. We took the liberty to call it *action* in the upcoming few paragraphs in order to avoid the obvious confusion that will result from calling it *command*.

Example 3-40 *Output of the Command* ***ip link show***

```
[NetProg@localhost ~]$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 08:00:27:a7:32:f7 brd ff:ff:ff:ff:ff:ff
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 08:00:27:83:40:75 brd ff:ff:ff:ff:ff:ff
4: enp0s9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 08:00:27:b4:ce:55 brd ff:ff:ff:ff:ff:ff
5: enp0s10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 08:00:27:48:59:02 brd ff:ff:ff:ff:ff:ff
6: virbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT qlen 1000
    link/ether 52:54:00:ea:c5:d4 brd ff:ff:ff:ff:ff:ff
7: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast master virbr0 state DOWN mode DEFAULT qlen 1000
    link/ether 52:54:00:ea:c5:d4 brd ff:ff:ff:ff:ff:ff
[NetProg@localhost ~]$ ip link show dev enp0s3
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 08:00:27:a7:32:f7 brd ff:ff:ff:ff:ff:ff
[NetProg@localhost ~]$
```

Table 3-1 lists some of the objects that are commonly used with the **ip** command.

Table 3-1 *Objects That Are Commonly Used with the* ***ip*** *Command*

Object	Description
address	IPv4 or IPv6 protocol address
link	Network interface
route	Routing table entry
maddress	Multicast address
neigh	ARP entry

As of this writing, there are 19 objects that can be acted upon by using the **ip** command. A full list of objects can be found in the man pages for the **ip** command. Objects can be written in full or in abbreviated form, such as **address** or **addr**. The actions that can be used with the **ip** command are limited to three options listed in Table 3-2.

Table 3-2 *Actions That Can Be Used with the* ***ip*** *Command*

Action	Description
add	Adds the object
delete	Deletes the object
show (or **list**)	Displays information about the object

The keyword **show** or **list** can be dropped from a command, and the command will still be interpreted as a **show** action. For example, the command **ip link show** is equivalent to just **ip link**.

The **ip addr** command lists all interfaces on the system, each with its IP address information, and the **ip maddr** command displays the multicast information for each and every interface. The **ip neigh** command displays the ARP table. The ARP table consists of a list of neighbors on each interface on the local network. The examples in this section show how to use these **show** commands.

You can bring an interface on Linux up or down by using the command **ip link set** {*intf*} {**up|down**}. Example 3-41 shows how to bring interface enp0s8 down and then up again. Note that changing networking configuration on Linux, including toggling an interface's state, requires root privileges. The **show** commands, however, do not. To keep Example 3-41 short and avoid the frequent password prompt, all commands in the example are issued by the root user. However, running commands as root in general is *not* a recommended practice. On a production network, make sure to avoid logging in as root. It is best practice to log in with your regular user account and use the **sudo** command whenever a command requires root privileges to execute, as explained in Chapter 2.

Example 3-41 *Toggling Interface State*

```
[root@localhost ~]# ip link show dev enp0s8
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
```

In the routing table, the list of routes on the system can be displayed by using the command **ip route**. Example 3-43 shows that the routing table is empty when no IP addresses are configured on any of the interfaces. When the IP address 10.2.0.30/24 is configured on interface enp0s3, one entry, corresponding to that interface, is added to the routing table.

Example 3-43 *Viewing a Routing Table by Using the* ***ip route*** *Command*

```
[NetProg@server4 ~]$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    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: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:2c:61:d0 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::a00:27ff:fe2c:61d0/64 scope link
       valid_lft forever preferred_lft forever
[NetProg@server4 ~]$ ip route


[NetProg@server4 ~]$ sudo ip addr add 10.2.0.30/24 dev enp0s3
[sudo] password for NetProg:
[NetProg@server4 ~]$ ip addr show dev enp0s3
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:2c:61:d0 brd ff:ff:ff:ff:ff:ff
    inet 10.2.0.30/24 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe2c:61d0/64 scope link
       valid_lft forever preferred_lft forever
[NetProg@server4 ~]$ ip route
10.2.0.0/24 dev enp0s3 proto kernel scope link src 10.2.0.30
[NetProg@server4 ~]$
```

Routing tables on Linux systems are very similar to routing tables on routers. In fact, a Linux server could easily function as a router. In order to display routing table functionality in Linux, server1 in the topology in Figure 3-2 is used as a router to route traffic between server2 and server3. server2 is connected to network 10.1.0.0/24, and server3 is connected to network 10.2.0.0/24. All three servers are configured such that server1 routes between the two networks, and eventually server2 should be able to ping server3.

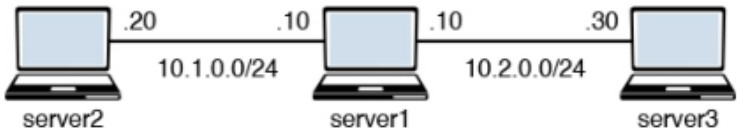


Figure 3-2 *Server1 Configured to Route Between server2 and server3, Each on a Different Subnet*

IP addressing needs to be configured first. server1 is configured with IP addresses ending with .10, server2 with an IP address ending in .20, and server3 with an IP address ending in .30, as shown in Example 3-44.

Example 3-44 *Configuring IP Addresses on the Interfaces Connecting The Three Servers*

```
! server1
[root@server1 ~]# ip addr add 10.1.0.10/24 dev enp0s8
[root@server1 ~]# ip addr add 10.2.0.10/24 dev enp0s9
[root@server1 ~]# ip addr show enp0s8 | grep "inet "
    inet 10.1.0.10/24 scope global enp0s8
[root@server1 ~]# ip addr show enp0s9 | grep "inet "
    inet 10.2.0.10/24 scope global enp0s9
[root@server1 ~]#
```

control center, which provides basic network configuration. The other GUI tool is the Connection Editor and is used to configure more advanced settings. You can start the Connection Editor from the terminal by entering the command **nm-connection-editor**.

• **NetworkManager command-line interface (nmcli)**: The NetworkManager CLI is a command-line utility that you can use to control NetworkManager. You start this interface by issuing the **nmcli** command in the shell.

• **NetworkManager text user interface (nmtui)**: Similar to the interface used to configure a computer's BIOS settings or old DOS-based programs, the **nmtui** provides an interface to NetworkManager that displays graphics in text mode. You start the text user interface by issuing the **nmtui** command in the shell.

• **API**: NetworkManager provides an API that can be used by applications for programmatic access to NetworkManager.

Because the majority of automation is typically performed through CLI tools (and API calls) and not the GUI, this section cover NetworkManager configuration via the **nmcli** interface.

NetworkManager deals with objects called connections. A *connection* is a representation of a link to the outside world and may represent, for example, a wired connection, a wireless connection, or a VPN connection. To display the current status of all network connections on a system, use the command **nmcli con show**, as shown in Example 3-48.

Example 3-48 *Listing All Connections on a System*

```
[root@server1 ~]# nmcli con show
NAME                UUID                                  TYPE            DEVICE
Wired connection 1  d8323782-5cf2-3afc-abcd-e603605ac4f8  802-3-ethernet  --
Wired connection 2  669fefb4-bc57-3d19-b83b-2b2125e0036b  802-3-ethernet  --
[root@server1 ~]#
```

The output in Example 3-48 indicates that there are two connections, named Wired connection 1 and Wired connection 2. These connections are not bound (applied) to any interfaces, as indicated by the -- in the last column. Both connections are of type Ethernet. A connection is uniquely identified by its universally unique identifier (UUID). Although not shown in the command output, a connection can either be active or inactive. To activate an inactive connection, you use the command **nmcli con up** {*connection_name*}. To deactivate a connection, you replace the keyword **up** with the keyword **down**.

Each connection is known as a *connection profile* and contains several attributes or properties that you can set. These properties are known as *settings*. Connection profile settings are created and then applied to a device or device type. Settings are represented in a dot notation. For example, a connection's IPv4 addresses are represented by the setting **ipv4.addresses**. To drill down on the details for a specific connection and list its settings and their values, you can use the command **nmcli con show** {*connection_name*}. Example 3-49 lists the connection profile settings for Wired connection 1. The output is truncated due to the length of the list. A full list of settings and their meanings can be found in the man pages for the **nmcli** command.

Example 3-49 *Connection Attributes for Wired Connection 1*

```
[root@server1 ~]# nmcli con show "Wired connection 1"
connection.id:                          Wired connection 1
connection.uuid:                        d8323782-5cf2-3afc-abcd-e603605ac4f8
connection.stable-id:                   --
connection.interface-name:              --
connection.type:                        802-3-ethernet
connection.autoconnect:                 yes
connection.autoconnect-priority:        -999
connection.autoconnect-retries:         -1 (default)
connection.timestamp:                   1525512827
connection.read-only:                   no
connection.permissions:                 --
connection.zone:                        --
connection.master:                      --
connection.slave-type:                  --
connection.autoconnect-slaves:          -1 (default)
connection.secondaries:                 --
connection.gateway-ping-timeout:        0
connection.metered:                     unknown
connection.lldp:                        -1 (default)
802-3-ethernet.port:                    --
```

```
[root@server1 ~]# nmcli con show
NAME                UUID                                  TYPE            DEVICE
Wired connection 1  d8323782-5cf2-3afc-abcd-e603605ac4f8  802-3-ethernet  --
Wired connection 2  669fefb4-bc57-3d19-b83b-2b2125e0036b  802-3-ethernet  --
[root@server1 ~]# nmcli con del "Wired connection 1"
Connection 'Wired connection 1' (d8323782-5cf2-3afc-abcd-e603605ac4f8) successfully deleted.
[root@server1 ~]# nmcli con del "Wired connection 2"
Connection 'Wired connection 2' (669fefb4-bc57-3d19-b83b-2b2125e0036b) successfully deleted.
[root@server1 ~]# nmcli con show
NAME  UUID  TYPE  DEVICE
[root@server1 ~]# nmcli con add con-name NetDev_1 type ethernet ifname enp0s8 ip4 10.1.0.10/24 gw4 10.1.0.254
Connection 'NetDev_1' (a8ac9116-697a-4a0a-85a2-63428d6e75a3) successfully added.
[root@server1 ~]# nmcli con show
NAME      UUID                                  TYPE            DEVICE
NetDev_1  a8ac9116-697a-4a0a-85a2-63428d6e75a3  802-3-ethernet  enp0s8
[root@server1 ~]# nmcli con show --active
NAME      UUID                                  TYPE            DEVICE
NetDev_1  a8ac9116-697a-4a0a-85a2-63428d6e75a3  802-3-ethernet  enp0s8
[root@server1 ~]# nmcli dev status
DEVICE  TYPE      STATE         CONNECTION
enp0s8  ethernet  connected     NetDev_1
enp0s9  ethernet  disconnected  --
lo      loopback  unmanaged     --
[root@server1 ~]# nmcli dev show enp0s8
GENERAL.DEVICE:                         enp0s8
GENERAL.TYPE:                           ethernet
GENERAL.HWADDR:                         08:00:27:83:40:75
GENERAL.MTU:                            1500
GENERAL.STATE:                          100 (connected)
GENERAL.CONNECTION:                     NetDev_1
GENERAL.CON-PATH:                       /org/freedesktop/NetworkManager/ActiveConnection/359
WIRED-PROPERTIES.CARRIER:               on
IP4.ADDRESS[1]:                         10.1.0.10/24
IP4.GATEWAY:                            10.1.0.254
IP6.ADDRESS[1]:                         fe80::8c1f:4c4a:51a5:6423/64
IP6.GATEWAY:                            --
[root@server1 ~]# ping 10.1.0.20 -c 3
PING 10.1.0.20 (10.1.0.20) 56(84) bytes of data.
64 bytes from 10.1.0.20: icmp_seq=1 ttl=64 time=0.604 ms
64 bytes from 10.1.0.20: icmp_seq=2 ttl=64 time=0.602 ms
64 bytes from 10.1.0.20: icmp_seq=3 ttl=64 time=0.732 ms

--- 10.1.0.20 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2011ms
rtt min/avg/max/mdev = 0.602/0.646/0.732/0.060 ms
[root@server1 ~]#
```

Notice that once a connection has been created and the device enp0s8 has been bound to it (all in the same command), the connection and device both come up, and that results in the device successfully pinging server2 on the other end of the link.

After a connection is created, you can modify its settings by using the command **nmcli con mod** {*connection_name*} {*setting*} {*value*}. When modifying a setting, the full dot format is required in the command. If the shorthand format is used, the new value in the command may be added to the existing value of the setting. For example, if the shorthand format is used to modify the IP address, the new IP address in the command is added to the device as a secondary IP address. On the other hand, if the full dot format is used, the IP address in the command replaces the IP address configured on the device. Example 3-52 shows how to modify the IP address of device enp0s8 to 10.1.0.100/24.

Example 3-52 *Deleting and Creating Connections*

```
[root@server1 ~]# nmcli con show NetDev_1 | grep ipv4.addr

ipv4.addresses:                         10.1.0.10/24

[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD

IP4.ADDRESS[1]:                         10.1.0.10/24

[root@server1 ~]# nmcli con mod NetDev_1 ip4 10.1.0.100/24


! The new IP address is added as a secondary address due to the shorthand format

[root@server1 ~]# nmcli con show NetDev_1 | grep ipv4.addr

ipv4.addresses:                         10.1.0.10/24, 10.1.0.100/24


! The new IP address is not reflected to the device enp0s8

[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD

IP4.ADDRESS[1]:                         10.1.0.10/24


[root@server1 ~]# nmcli con up NetDev_1

Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkManager/ActiveConnection/366)


! After resetting the con, the new IP address now is reflected to the device

[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD

IP4.ADDRESS[1]:                         10.1.0.10/24

IP4.ADDRESS[2]:                         10.1.0.100/24


! Using the full dot format will replace the old IP address with the new one

[root@server1 ~]# nmcli con mod NetDev_1 ipv4.address 10.1.0.100/24

[root@server1 ~]# nmcli con up NetDev_1

Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkManager/ActiveConnection/367)

[root@server1 ~]# nmcli con show NetDev_1 | grep ipv4.addr

ipv4.addresses:                         10.1.0.100/24

[root@server1 ~]# nmcli dev show enp0s8 | grep IP4.ADD

IP4.ADDRESS[1]:                         10.1.0.100/24

[root@server1 ~]#
```

Note that each time a change is made to a connection using **nmcli**, the connection needs to be reactivated in order for the changes to be reflected to the device.

Adding routes using **nmcli** is different than adding routes using the **ip** utility in that when using **nmcli**, routes are added per interface and not globally. You add routes by using the syntax **nmcli con mod** {*intf*} **+ipv4.routes** {*destination*} **ipv4.gateway** {*next_hop*}. Therefore, to accomplish the same task that was done earlier by using the **ip** utility (to add a route on server2 to direct traffic destined for network 10.2.0.0/24 using the next hop 10.1.0.10 on server1), you use the following command: **nmcli con mod enp0s3 +ipv4.routes 10.2.0.0/24 ipv4.gateway 10.1.0.10**.

Unlike with the **ip** utility, changes made through **nmcli** are, by default, persistent and will survive a system reload.

It is important to understand the difference between the **ip** utility and NetworkManager. The **ip** utility is a program. When you use the **ip** command, you run this program, which makes a system call to the kernel, either to retrieve information or configure a component of the Linux networking system.

On the other hand, NetworkManager is a system daemon. It is software that runs (lurks) in the background, by default, and oversees the operation of the Linux network system. NetworkManager may be used to configure components of the network or to retrieve information about the network by using a variety of methods discussed earlier in this section—one of them being **nmcli**.

The nuances of how the **ip** utility interacts with NetworkManager are not discussed in detail here. All you need to know for now is that changes to the network that are made via the **ip** utility are detected and preserved by NetworkManager. There is no conflict between them. As mentioned at the very beginning of this section, different software on Linux can achieve the same result via different communication channels with the kernel. However, any software that needs access to the network will eventually have to go through the kernel.

Network Scripts and Configuration Files

The third method for configuring network devices and interfaces is to modify network scripts and configuration files directly. Different files in Linux control different components of the networking ecosystem, and editing these files was the only way to configure networking on Linux before NetworkManager was developed. Configuration files and scripts can still be used instead of, or in addition to, NetworkManager.

On Linux distros in the Red Hat family, configuration files for network interfaces are located in the /etc/sysconfig/network-scripts directory, and each interface configuration file is named ifcfg-**<***intf_name***>**. The first script that is executed on system bootup is /etc/init.d/network. When the system boots up, this script reads through all interface configuration files whose names start with ifcfg. Example 3-53 shows the ifcfg file for the enp0s8 interface.

Example 3-53 *Interface Configuration File for Interface enp0s8*

```
[root@server1 network-scripts]# cat ifcfg-enp0s8

TYPE=Ethernet

PROXY_METHOD=none

BROWSER_ONLY=no

BOOTPROTO=dhcp

DEFROUTE=yes

IPV4_FAILURE_FATAL=no

IPV6INIT=yes

IPV6_AUTOCONF=yes

IPV6_DEFROUTE=yes

IPV6_FAILURE_FATAL=no

IPV6_ADDR_GEN_MODE=stable-privacy

NAME=NetDev_1

UUID=a8ac9116-697a-4a0a-85a2-63428d6e75a3

DEVICE=enp0s8

ONBOOT=yes

[root@server1 network-scripts]#
```

The filename just needs to be prefixed with ifcfg. The network script simply scans the directory and reads any file whose name has this prefix. Therefore, you can safely assume that the configuration file is for the interface *or* connection. However, while the filename has to start with ifcfg, there is general consensus that the value in the DEVICE field (interface) should follow.

The TYPE field in the file indicates the connection type, which is **Ethernet** in this case. The BOOTPROTO field is set to **dhcp,** which means the connection gets an IP address via DHCP. If a static IP address is required on the interface, then dhcp is replaced with none. The interface associated with this configuration is also shown in the DEVICE field (enp0s8 in this case), and the ONBOOT field indicates that this connection is to be brought up at system bootup. When a static IP address is required on the interface, the fields IPADDR, PREFIX, and GATEWAY and their respective values are added to the file.

When **ONBOOT=yes** is set, the /etc/init.d/network script checks whether this interface is managed by NetworkManager. If it is and the connection has already been activated, no further action is taken. If the connection has not been activated, the script requests NetworkManager to activate the connection. In case the connection is not managed by NetworkManager, the network script activates the connection by running another script, /usr/sbin/ifup. The ifup script checks the field TYPE in the ifcfg file, and based on that, it calls *another* type-specific script. For example, if the type of the connection is Ethernet, the ifup-eth script is called. Linux requires type-specific scripts because different connection types require different configuration parameters. For example, the concept of an SSID (wireless network name) does not exist for an Ethernet connection. Similarly, to bring down an interface for an unmanaged interface, the ifdown script is called. The vast majority of interface types are managed by NetworkManager by default, unless the line **NM_CONTROLLED=no** has been added to the ifcfg file.

While the recommended method for configuring interfaces is to use the **nmcli** utility, as discussed in the previous section, you can also configure interfaces by editing the corresponding ifcfg file.

Static routes configured on a system have configuration files named **route-<***intf_name***>** in the same directory as the interface configuration files. As you have probably guessed, the name has to be prefixed with **route**. However, the **-<***intf_name***>** is just a naming convention, and the file may have any name as long as the prefix **route** is there. The routing entries in the file may have one of two formats:

• The **ip** command arguments format:

{*destination*}/{*mask*} **via** {*next_hop*} [**dev** *interface*]

With this format, specifying the interface using [**dev** *interface*] is optional.

• The network/netmask directives format:

ADDRESS{*N*}:{*destination*}

NETMASK{*N*}:{*netmask*}

GATEWAY{*N*}:{*next_hop*}

where *N* is the routing table entry starting with 0 and incrementing by 1 for each entry, without skipping any values. In other words, if the routing table has four entries, the entries are numbered from 0 to 3.

Going back to the network of three servers in Figure 3-2, where server1 is required to route between server2 on subnet 10.1.0.0/24 and server3 on subnet 10.2.0.0/24, the static routes needed to route between the servers are configured and subsequently deleted, after which the ping from server2 to server3 fails, as shown in Example 3-54.

Example 3-54 *Ping Fails Due To Lack of Static Routes on server2 and server3*

```
! No routes in routing table of server2 to remote subnet 10.2.0.0/24
[root@server2 ~]# ip route
10.1.0.0/24 dev enp0s3 proto kernel scope link src 10.1.0.20 metric 100
[root@server2 ~]#


! Ping to the directly connected interface on server1 is successful
[root@server2 ~]# ping -c 1 10.1.0.10
PING 10.1.0.10 (10.1.0.10) 56(84) bytes of data.
64 bytes from 10.1.0.10: icmp_seq=1 ttl=64 time=0.828 ms


--- 10.1.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.828/0.828/0.828/0.000 ms
[root@server2 ~]#


! Ping to server3 on subnet 10.2.0.0/24 is not successful
[root@server2 ~]# ping -c 1 10.2.0.30
connect: Network is unreachable
[root@server2 ~]#


! No routes in routing table of server3 to remote subnet 10.1.0.0/24
[root@server3 ~]# ip route
10.2.0.0/24 dev enp0s3 proto kernel scope link src 10.2.0.30 metric 100
[root@server3 ~]#


! Ping to the directly connected interface on server1 is successful
[root@server3 ~]# ping -c 1 10.2.0.10
PING 10.2.0.10 (10.2.0.10) 56(84) bytes of data.
64 bytes from 10.2.0.10: icmp_seq=1 ttl=64 time=0.780 ms


--- 10.2.0.10 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.780/0.780/0.780/0.000 ms
[root@server3 ~]#


! Ping to server2 on subnet 10.1.0.0/24 is not successful
[root@server3 ~]# ping -c 1 10.1.0.20
connect: Network is unreachable
[root@server3 ~]#
```

The file route-enp0s3 is created under the directory /etc/sysconfig/network-scripts/ on both servers. A routing entry is added to the routing configuration file on server2 by using the **ip** command arguments format, and a routing entry is added to the file on server3 by using the network/netmask directives format, as shown in Example 3-55.

Example 3-55 *Routing Configuration Files Added on Both server2 and server3*

```
! server2


! No routing configuration files in the directory

[root@server2 ~]# cd /etc/sysconfig/network-scripts/

[root@server2 network-scripts]# ls -l | grep "route"

[root@server2 network-scripts]#


! Create the file route-enp0s3 and populate it with a route to the remote subnet 10.2.0.0/24 using the IP Command
Arguments format

[root@server2 network-scripts]# touch route-enp0s3

[root@server2 network-scripts]# echo "10.2.0.0/24 via 10.1.0.10" >> route-enp0s3

[root@server2 network-scripts]# ls -l | grep " route"

-rw-r--r--. 1 root root    26 Aug 17 15:52 route-enp0s3

[root@server2 network-scripts]# cat route-enp0s3

10.2.0.0/24 via 10.1.0.10

[root@server2 network-scripts]#


! Restart the network service and check the routing table

[root@server2 network-scripts]# systemctl restart network

[root@server2 network-scripts]# ip route

10.1.0.0/24 dev enp0s3 proto kernel scope link src 10.1.0.20 metric 100

10.2.0.0/24 via 10.1.0.10 dev enp0s3 proto static metric 100

[root@server2 network-scripts]#


! server3


! No routing configuration files in the directory

[root@server3 ~]# cd /etc/sysconfig/network-scripts/

[root@server3 network-scripts]# ls -l | grep " route"


! Create the file route-enp0s3 and populate it with a route to the remote subnet 10.1.0.0/24 using the Network/Netmask
Directives format

[root@server3 network-scripts]# touch route-enp0s3

[root@server3 network-scripts]# echo "ADDRESS0=10.1.0.0" >> route-enp0s3

[root@server3 network-scripts]# echo "NETMASK0=255.255.255.0" >> route-enp0s3

[root@server3 network-scripts]# echo "GATEWAY0=10.2.0.10" >> route-enp0s3

[root@server3 network-scripts]# ls -l | grep " route"

-rw-r--r--. 1 root root    60 Aug 17 16:04 route-enp0s3

[root@server3 network-scripts]# cat route-enp0s3

ADDRESS0=10.1.0.0

NETMASK0=255.255.255.0

GATEWAY0=10.2.0.10

[root@server3 network-scripts]#


! Restart the network service and check the routing table
```

```
[root@server3 network-scripts]# systemctl restart network

[root@server3 network-scripts]# ip route

10.1.0.0/24 via 10.2.0.10 dev enp0s3 proto static metric 100

10.2.0.0/24 dev enp0s3 proto kernel scope link src 10.2.0.30 metric 100

[root@server3 network-scripts]#
```

The ping test is now successful, and server2 can reach server3, as shown in Example 3-56.

Example 3-56 *Ping from server2 to server3 and Vice Versa Is Successful Now*

```
[root@server2 network-scripts]# ping -c 1 10.2.0.30

PING 10.2.0.30 (10.2.0.30) 56(84) bytes of data.

64 bytes from 10.2.0.30: icmp_seq=1 ttl=63 time=2.11 ms


--- 10.2.0.30 ping statistics ---

1 packets transmitted, 1 received, 0% packet loss, time 0ms

rtt min/avg/max/mdev = 2.119/2.119/2.119/0.000 ms

[root@server2 network-scripts]#


[root@server3 network-scripts]# ping -c 1 10.1.0.20

PING 10.1.0.20 (10.1.0.20) 56(84) bytes of data.

64 bytes from 10.1.0.20: icmp_seq=1 ttl=63 time=1.58 ms


--- 10.1.0.20 ping statistics ---

1 packets transmitted, 1 received, 0% packet loss, time 0ms

rtt min/avg/max/mdev = 1.585/1.585/1.585/0.000 ms

[root@server3 network-scripts]#
```

The network script is run as a service and, like any other service, can be controlled by using the command **systemctl {start|stop|restart|status} network**. To enable/disable the network service at startup, you use the command **chkconfig network {on|off}**. Keep in mind that after a configuration file is changed, the network service has to be restarted for the changes to take effect. It goes without saying that any configuration done via amending the network configuration files is persistent and will remain intact after a system reload.

Network Services: DNS

Domain Name System (*DNS*) is a hierarchical naming system used on the Internet and some private networks to assign domain names to resources on the network. Domain names tend to be easier to remember than IP addresses. Using domain names provides the additional capability to resolve a domain name to multiple IP addresses for purposes such as high availability or routing user traffic based on the geographically closest server.

DNS uses the concept of a *resolver*, commonly referred to as a *DNS server*, which is a server or a database that contains mappings between domain names and the information related to each of those domain names, such as the IP addresses. These mappings are called *records*. DNS is hierarchical and distributed. The majority of DNS servers maintain records for only some domain names and then initiate queries to other DNS servers for the rest of the domain names, for which it does not maintain records.

Performing a DNS query means sending a request to a DNS server to resolve the domain name and return the data associated with that domain name. To resolve a domain name on Linux to its corresponding information, including its IP address, you use the **dig** command, which stands for *domain information groper*. Example 3-57 shows **dig** being used to resolve google.com to its public IP address. The public IP address received from the DNS response is highlighted in the example.

Example 3-57 *Using the **dig** Command to Resolve google.com*

```
[root@server1 ~]# dig google.com


; <<>> DiG 9.9.4-RedHat-9.9.4-51.el7_4.2 <<>> google.com

;; global options: +cmd

;; Got answer:

;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 38879

;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
```

```
;; OPT PSEUDOSECTION:

; EDNS: version: 0, flags:; udp: 4096

;; QUESTION SECTION:

;google.com.                        IN        A


;; ANSWER SECTION:

google.com.               264       IN         A         216.58.207.14


;; Query time: 31 msec

;; SERVER: 192.168.8.1#53(192.168.8.1)

;; WHEN: Fri Aug 17 17:16:06 +03 2018

;; MSG SIZE  rcvd: 55

[root@server1 ~]#
```

In Example 3-57, the DNS server used for the name resolution is 192.168.8.1. The IP address of this DNS server is configured in the /etc/resolv.conf file, shown in Example 3-58. To configure other DNS servers, you list each server's IP address on a new line in this file.

Example 3-58 *List of DNS Servers in the /etc/resolv.conf File*

```
[root@server1 ~]# cat /etc/resolv.conf

# Generated by NetworkManager

nameserver 192.168.8.1

[root@server1 ~]#
```

Manual DNS entries are configured in the /etc/hosts file. If an entry for a domain name is found in that file, the DNS servers are not consulted for resolution. There is one caveat, though: The **dig** command still requests the name resolution from the DNS server configured in /etc/resolv.conf. However, the **ping** command and also the web browsers on the system use the hosts file, and, therefore, use the manual entry there. Try to add a manual entry for google.com in the hosts file, pointing to an IP address that is not reachable and then try to use **dig**, use **ping**, and browse to google.com and notice how each of these behave differently.

Summary

This chapter takes Linux administration a step further and covers storage, security, and networking. It discusses the following topics:

- Partitioning, formatting, and managing physical storage
- Creating physical volumes, volume groups, and logical volumes using LVM
- User and group security management
- File security management, including permission bits and ACLs
- Linux system security, including the Linux firewall
- Managing Linux networking by using the **ip** utility
- Managing Linux networking by using the NetworkManager CLI (**nmcli**)
- Managing Linux networking via network scripts and configuration files
- Network services such as DNS

Chapter 4, "Linux Scripting," builds on this chapter and covers Linux scripting, which is one big step towards automation.

Chapter 4. Linux Scripting

Recall from Chapter 1, "The Network Programmability and Automation Ecosystem," and Chapter 2, "Linux Fundamentals," that automation has been described as a process of breaking down a big task into smaller, *repeatable* tasks and then attempting to automate each of those repeatable tasks by doing the heavy lifting only once and having a tool repeat the task for you. This chapter gives you your first real taste of automation using programmability: writing scripts in the Bash programming language, more commonly referred to as the Bash scripting language. Scripts (whether in Bash or any other language) are written once and can, theoretically, be executed an infinite number of times.

A *shell* is a program that parses and interprets commands and then passes the (interpreted) commands to the kernel for execution. Commands are typically entered into the shell individually, through what is commonly known as *interactive mode*: You type a command, and you get the result instantly. Piping and redirection add to the sophistication of what can be achieved interactively with the shell. Scripting takes this a step further, by allowing you to type a sequence of commands in a file and then pass the file to the shell for execution, without having to manually enter the commands through the CLI one by one. With high-level programming languages like Python, Java, or Go, the resulting sequence of commands is commonly called a *program*. In case of Bash, such a sequence is commonly referred to as a *script* because Bash is not a full-fledged programming language. Bash is typically used to glue together different components into a script, where each of the components may be a program on its own (for example, a Linux utility).

Scripts can run system commands such as **pwd**, **ls**, and **cat**. Scripts typically use *variables* that hold values that can change throughout the execution of the script. They may also contain constructs for *conditional* execution of code (running a line or block of code only if a particular condition is true) or constructs for *looping* (running the same line or block of code more than once). Scripts can read *input* from the keyboard or a file, and they can *output* to the screen or to a file. These examples just scratch the surface of what a Bash script may contain or what it can do.

This chapter starts by covering three new utilities: **grep**, **awk**, and **sed**. These advanced but simple utilities are heavily used in scripting. Then, after introducing the general structure of a Bash script, the chapter covers the following topics:

• Comments

• Input and output

• Variables and arrays

• Expansion, operations, and comparisons

• System commands

• Conditional statements

• Loops

• Functions

Finally, this chapter touches on Expect, a programming language that is frequently used to extend the capabilities of Bash with respect to interacting with a user running a script. Expect comes in handy, for example, when logging in to a device requires a username and password to be entered by the user.

The *GNU Bash Manual* is published by the Free Software Foundation (FSF) at https://www.gnu.org/software/bash/manual/. As you read this chapter, consult the manual any time you need more detailed coverage of the subjects presented here.

Regular Expressions and the grep Utility

Regular expressions are briefly introduced in Chapter 3, "Linux Storage, Security and Networks." If you are a network engineer or a software developer, chances are that you have already had some exposure to regular expressions. If not, do not worry, as they are thoroughly covered in this section.

A *regular expression*, or *regex* for short, is basically a sequence of characters, commonly referred to as a *pattern*. The pattern is used to search through text to find matches. Each character in a pattern may be a literal or a metacharacter. A *literal* represents itself; for example, the regex **zz** would match any occurrence of the literal zz, such as **zz**, o**zz**, or bli**zz**ard. A *metacharacter* has a special meaning; for example, the plus symbol (**+**) means one or more occurrences of the character that precedes it. For example, the regex **12+** would match the number 1 followed by one or more occurrences of the number 2, such as **12**, **122**, or **1222**. The concept is very simple, yet extremely powerful and provides limitless use cases. This section covers how the **grep** utility can be used to search through a file for lines that match some criteria expressed by using regular expressions.

Note

A regex pattern may use square brackets (**[]**), curly braces (**{}**) or parenthesis (**()**)as part of the pattern itself, as you will see shortly in this chapter. For this reason, and to avoid confusion, the standard code conventions used in the rest of the book will not be used throughout this chapter. Instead, keywords that need to be typed literally as shown will appear in **bold** and placeholders for arguments for which you should supply actual values will be entered in *italic*. Any other symbol or character will be entered literally as shown.

The **grep** utility, originally created by Ken Thompson for UNIX, searches a file for text that matches one or more patterns and then prints out the lines containing matches to stdout. Although the general syntax of the command is **grep** *options patterns file_name*, the command is commonly coupled with redirection (piping) to search through the output of another command in order to display only the lines that have pattern matches.

The **grep** utility accepts several flavors of regular expressions, based on the options used in the command:

• **grep -G** *patterns*: Interprets the patterns in the command as GNU/standard regular expressions

• **grep -E** *patterns*: Interprets the patterns in the command as extended regular expressions

• **grep -F** *patterns*: Interprets the patterns in the command as fixed regular expressions

• **grep -P** *patterns*: Interprets the patterns in the command as Perl regular expressions

grep -G supports basic regular expressions, which can use any of the following metacharacters:

• **^**: Represents the beginning of a line

• **$**: Represents the end of a line

• **.**: Represents any single character except a newline character

• *****: Represents zero or more occurrences of the preceding character

• **[***literals***]**: Represents a single character that matches any one of the enclosed literals. For example **[abc]** is equivalent to "match either **a** or **b** or **c**"

• **[***first_literal***-***last_literal***]**: Represents a single character that matches any character in the range between the brackets (inclusive). For example **[a-f]** is equivalent to "match any letter from **a** to **f**" and **[1-5]** is equivalent to "match any number from **1** to **5**".

• **\<** or **\b**: Represents the beginning of a word

• **\>** or **\b**: Represents the end of a word

• \: Represents the escape character

In addition to all the options supported with the **-G** option, the **grep -E** command also supports the following metacharacters:

• **?**: Represents zero or one occurrences of the preceding character

• **+**: Represents one or more occurrences of the preceding character

• **{***N***}** or **{***N***,***M***}**: Represents strings with *N* repetitions or between *N* and *M* repetitions

• **|**: Represents the OR operator

• **( )**: Represents a capture group

Each of these metacharacters is explained in detail in this section. Because the **grep** command with the **-E** option supports both basic and extended regular expressions, the examples throughout this section use the **-E** option.

Note

grep -E may be replaced by the command **egrep**, and **grep -F** may be replaced by the command **fgrep**. However, the **egrep** and **fgrep** commands are deprecated and are mentioned here in case you run into a legacy system or code that uses either of them.

The **grep** utility searches text files for text that matches a particular regex pattern and then prints out (or, more formally, outputs to stdout) the whole line where a match is found. In fact, **grep** stands for *global regular expression print*. In Example 4-1, the file named data-file is used to showcase the use of **grep** and regular expressions.

Example 4-1 *Data File for Testing Regular Expressions with* ***grep***

```
[NetProg@server1 grep-scripts]$ cat data-file

KhaledAbuelenain 11 Anton 22 Cairo 33 jeff

Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333

Vinit 1111 Jain 2222 Palo Alto 3333 JeffDoyle 4444

11111 Khaled 22222 Ca:ro Dusseldorf 33333 Anton

Anton Karneliuk Ca9ro Jeff
```

Example 4-2 shows how to search for all lines containing the pattern **Jeff**. No metacharacters are used in this example.

Example 4-2 *Printing the Lines Containing Matches to the Pattern Jeff*

```
[NetProg@server1 grep-scripts]$ grep 'Jeff' data-file

Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333
```

```
Vinit 1111 Jain 2222 Palo Alto 3333 JeffDoyle 4444

Anton Karneliuk Ca9ro Jeff
```

Although **grep** prints the whole line where a match is found, on a monitor the *matching pattern* is typically colored differently, usually in a shade of red. However, this is configurable and implementation specific. In the examples in this section, the matches are highlighted in gray shading. As you can see in Example 4-2, running the **grep** command results in three matches, and three lines containing the pattern **Jeff** have been printed out. Notice two things here. First, it doesn't matter where in the line the pattern **Jeff** appears; regardless of whether it is at the beginning of the line, in the middle, or at the end, it will match. The second thing to notice is that any occurrence of the pattern will result in a match, whether it is a standalone word or a part of another word, such as JeffDoyle.

Now say that you need to find the same word but only when it is at the beginning of a line. Example 4-3 illustrates the use of the caret symbol (**^**) to match a pattern only if the line begins with it. Similarly, the dollar symbol (**$**) is used to find the lines that end with the pattern **Jeff**.

Example 4-3 *Using ^ and $ to Find Lines That Start and End with a Pattern*

```
[NetProg@server1 grep-scripts]$ grep -E ^Jeff data-file

Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333



[NetProg@server1 grep-scripts]$ grep -E Jeff$ data-file

Anton Karneliuk Ca9ro Jeff
```

Notice that the first line in the file ends with the name jeff. However, the name begins with a lowercase j, while the pattern used for matching has an uppercase J. **grep** is case sensitive. Example 4-4 provides two solutions that allow you to work around this behavior. The first solution is to use the option **-i**, which makes **grep** case insensitive. The other solution is to use a character class.

Example 4-4 *Using Either The -i Option Or a Character Class to Match on Uppercase and Lowercase Letters*

```
! Using the -i option to make grep case insensitive

[NetProg@server1 grep-scripts]$ grep -E -i Jeff data-file

KhaledAbuelenain 11 Anton 22 Cairo 33 jeff

Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333

Vinit 1111 Jain 2222 Palo Alto 3333 JeffDoyle 4444

Anton Karneliuk Ca9ro Jeff



! Using a character class

[NetProg@server1 grep-scripts]$ grep -E [Jj]eff data-file

KhaledAbuelenain 11 Anton 22 Cairo 33 jeff

Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333

Vinit 1111 Jain 2222 Palo Alto 3333 JeffDoyle 4444

Anton Karneliuk Ca9ro Jeff
```

A character class allows you to specify alternatives to match on for a *particular character position*. In Example 4-4, any word that begins with either **j** or **J** matches. Different options are entered side by side between brackets. For example, at a specific position in a string, in order to match one of the following letters: a, b, c, or d, you can use the character class **[abcd]** in that position in the pattern. In addition, a character class can provide ranges. To match any small character in the alphabet, you can use the character class **[a-z]**, and in order to match any alphanumeric character (character or number), you can use the character class **[a-zA-Z0-9]**.

Now what if you need to match the beginning of a word or the end of a word? The data file contains several occurrences of the name Khaled. Example 4-5 shows how to use the character sequences **\<** and **\>** to match the pattern **Khaled** only if it starts or ends a word, respectively. Both can be used together to match a pattern if the pattern is a word on its own, as also shown in the example.

Example 4-5 *Matching the Beginning and Ending of Words by Using \< and \>*

```
! Matching the pattern Khaled

[NetProg@server1 grep-scripts]$ grep -E Khaled data-file

KhaledAbuelenain 11 Anton 22 Cairo 33 jeff

Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333

11111 Khaled 22222 Ca:ro Dusseldorf 33333 Anton



! Matching the pattern Khaled if it is the beginning of a word only
```

```
[NetProg@server1 grep-scripts]$ grep -E '\<Khaled' data-file
KhaledAbuelenain 11 Anton 22 Cairo 33 jeff
11111 Khaled 22222 Ca:ro Dusseldorf 33333 Anton

! Matching the pattern Khaled at the end of a word only
[NetProg@server1 grep-scripts]$ grep -E 'Khaled\>' data-file
Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333
11111 Khaled 22222 Ca:ro Dusseldorf 33333 Anton

! Matching the pattern Khaled on its own
[NetProg@server1 grep-scripts]$ grep -E '\<Khaled\>' data-file
11111 Khaled 22222 Ca:ro Dusseldorf 33333 Anton
```

Notice that the regex pattern is enclosed in single quotation marks when **\<** and **\>** notations are used. This is because the backslash followed by a character has a special meaning, and for this special meaning to be applied, single quotation marks must be used. This is called *expansion*. Several expansion types exist. (You'll learn more about expansion later in this chapter.)

The dot (**.**) is used in a regex pattern to represent a single character. In Example 4-6, the dot is used in the pattern **Ca.ro**. As you can see, it matches **Cairo**, **Ca9ro**, and **Ca:ro**. It basically matches the letters **Ca**, followed by any single character, followed by the letters **ro**.

Example 4-6 *Using a Dot to Match Any Single Character*

```
[NetProg@server1 grep-scripts]$ grep -E Ca.ro data-file
KhaledAbuelenain 11 Anton 22 Cairo 33 jeff
11111 Khaled 22222 Ca:ro Dusseldorf 33333 Anton
Anton Karneliuk Ca9ro Jeff
```

Repetition metacharacters are characters that are used in a regex pattern to indicate a certain number of repetitions on the literal that immediately precedes the repetition character. For example, when a regex is composed of the letter A followed by a repetition character, that regex will match on zero or more repetitions of the letter A, depending on which repetition character is used. Four repetition notations are mentioned earlier in this chapter and are repeated here for convenience:

- *****: Represents zero or more occurrences of the preceding character
- **?**: Represents zero or one occurrences of the preceding character
- **+**: Represents one or more occurrences of the preceding character
- **{***N***}** or **{***N***,***M***}**: Represents words with *N* repetitions or between *N* and *M* repetitions

Example 4-7 uses some of these notations to illustrate the concept of repetition in patterns.

Example 4-7 *Using Repetition Characters*

```
! The full data file
[NetProg@server1 grep-scripts]$ cat data-file
KhaledAbuelenain 11 Anton 22 Cairo 33 jeff
Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333
Vinit 1111 Jain 2222 Palo Alto 3333 JeffDoyle 4444
11111 Khaled 22222 Ca:ro Dusseldorf 33333 Anton
Anton Karneliuk Ca9ro Jeff

! Using the asterisk symbol (*) to match zero or more occurrences of '2'
[NetProg@server1 grep-scripts]$ grep -E 2* data-file
KhaledAbuelenain 11 Anton 22 Cairo 33 jeff
Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333
Vinit 1111 Jain 2222 Palo Alto 3333 JeffDoyle 4444
11111 Khaled 22222 Ca:ro Dusseldorf 33333 Anton
```

```
Anton Karneliuk Ca9ro Jeff


! Using the asterisk symbol (*) to match a '2' followed by zero or more occurrences of '2'

[NetProg@server1 grep-scripts]$ grep -E 22* data-file

KhaledAbuelenain 11 Anton 22 Cairo 33 jeff

Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333

Vinit 1111 Jain 2222 Palo Alto 3333 JeffDoyle 4444

11111 Khaled 22222 Ca:ro Dusseldorf 33333 Anton


! Using the plus symbol (+) to match one or more occurrences of '2'

[NetProg@server1 grep-scripts]$ grep -E 2+ data-file

KhaledAbuelenain 11 Anton 22 Cairo 33 jeff

Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333

Vinit 1111 Jain 2222 Palo Alto 3333 JeffDoyle 4444

11111 Khaled 22222 Ca:ro Dusseldorf 33333 Anton
```

In the first section of Example 4-7, **grep** uses the regex **2***, which translates to zero or more occurrences of 2. This will basically match all lines because *no occurrences* of 2 will match *everything*. This is why the last line in the output is a match. In the second section, a tighter condition using the regex **22*** states that a word will match if it contains a 2 followed by zero or more occurrences of 2. This means that a single 2 or any number of consecutive 2s will result in a match. This is the same criterion specified in the last section in the example, but the last section uses a plus symbol (**+**) to match on any word that contains one or more occurrences of 2. As you can see, in this case, the regex **22*** is equivalent to the regex **2+**.

It is interesting to note that the dot (**.**) is frequently used with the asterisk (as in **.***) to indicate any number of any characters—that is, anything (or everything). This character sequence matches any word that has zero or more occurrences of any character. Of course, this is not very useful unless it is used as part of a larger regex in which, perhaps, it is used to indicate anything followed by, preceded by, or enclosed by a more meaningful pattern. In Example 4-8, this notation is used in the regex **Doyle.*3** to match the second line in the data file. The pattern simply matches on the string **Doyle** and the number **3**, on the same line, with anything in between.

Example 4-8 *Using the .* Notation to Match Anything/Everything*

```
[NetProg@server1 grep-scripts]$ grep -E Doyle.*3 data-file

Jeff 111 Doyle 222 Colorado AbuelenainKhaled 333
```

Understanding how the other repetition characters work may be slightly more challenging because **grep** outputs the whole line in which a match is found. Example 4-9 displays a new data file that contains a different number of consecutive 1s on each line.

Example 4-9 *Data File for Testing the Repetition Characters ? and + and the Notations {N} and {N,M}*

```
[NetProg@server1 grep-scripts]$ cat data-file-2

123

1123

11123

111123

1111123


23132

231132

2311132

23111132

231111132
```

First, the question mark (**?**) is used in Example 4-10 in the regex **11?23**. This pattern will match any sequence of characters that starts with the literal 1, followed by *zero or one* occurrence of 1, followed by the literal 23. Although this actually only matches 123 and 1123, the first five lines of the file all appear to match. Notice the highlighting in Example 4-10: The highlighted part of the line is the part that actually matches the regex (and is typically differently colored on a monitor). However, if the results are redirected to an output file and the matches appear all in the same font color, the results may be misleading. This would not be the case if the literal preceding the matching sequence were anything except the literal 1, as you will see in the next example.