The Textbook

THIRD EDITION

Syed Mansoor Sarwar
Robert M. Koretsky

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

UNIX
The Textbook
THIRD EDITION

UNIX
The Textbook
THIRD EDITION

Syed Mansoor Sarwar
Robert M. Koretsky


CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20160226

International Standard Book Number-13: 978-1-4822-3358-2 (Hardback)

Library of Congress Cataloging-in-Publication Data

Names: Sarwar, Syed Mansoor, author. | Koretsky, Robert, author.
Title: UNIX : the textbook / Syed Mansoor Sarwar and Robert M. Koretsky.
Description: Third edition. | Boca Raton : Taylor & Francis, CRC Press, 2016.
| Includes bibliographical references and index.
Identifiers: LCCN 2016009010 | ISBN 9781482233582 (alk. paper)
Subjects: LCSH: UNIX (Computer file) | Operating systems (Computers)
Classification: LCC QA76.774.U64 S37 2016 | DDC 005.4/32--dc23
LC record available at http://lccn.loc.gov/2016009010

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To my family

S.M.S.

To my family

R.M.K.

Contents

Preface to the Third Edition

This third edition of *UNIX: The Textbook* has many significant changes and additions incorporated into it, in terms of both the scope and content of the previous editions. It is a textbook on the modern, twenty-first-century UNIX operating system. It uses an introductory approach in style, very similar to the style of the previous editions. With the exception of four chapters on system programming, the book can be used very successfully by a complete novice, as well as by an experienced UNIX system user, in both an informal and formal learning environment.

The two UNIX systems that we deploy to illustrate everything in this edition are PC-BSD and Solaris. There are many things that make these two systems superior to, as well as very different from, any contemporary, nominally UNIX distribution, and also from other NIX-like operating systems, such as Linux and OS X. There are many topics covered in this book that older, more traditional textbook approaches to UNIX could not include, such as the Zettabyte File System (ZFS) and a highly developed KDE or Gnome GUI desktop environment. The traditional text-based command line interface, though, is still a very integral part of our presentation of UNIX.

CHANGES IN THE THIRD EDITION OF THIS BOOK

Because PC-BSD and Solaris UNIX have had many important functional additions made to the application user interface since the previous edition came out, and because UNIX is now an even more widely-dispersed system in the marketplace than previously, we felt that we needed to add instructional material to the book covering these additions, including:

- Showing desktop KDE PC-BSD and Gnome Solaris as base system implementations of UNIX.
- Adding methods for customizing vi, vim, and emacs.
- Adding a complete tutorial chapter on the Python programming language and its use in UNIX.
- Giving a complete tutorial on the `git` command, and using Github.
- Adding four new, complete chapters on UNIX system programming and the UNIX API.

- Revising the chapter on networking and internetworking to bring it in line with current standards.
- Complete covering system call interfaces, files, file-related data structures in the UNIX kernel, file I/O paradigms, and file manipulation API.
- Extensive coverage of UNIX processes and threads, process-related kernel data structures in the UNIX kernel, process management API, and signal handling.
- Comprehensively covering interprocess communication in UNIX using pipes, named pipes (FIFOs), and sockets.
- Comprehensively coverage of Internetworking with UNIX TCP/IP: the client–server software for the Internet services using sockets, including the design and implementation of concurrent servers using the `select` system call and the need for concurrent clients.
- Providing coverage of important practical considerations in the design and implementation of production-quality client–server software.
- Completely revising much of the tutorial section on the X Window System to now include writing Xlib and Xcb code.
- Adding a new, extensive chapter on UNIX system administration that details installation, maintenance, and updating/upgrading PC-BSD and Solaris systems on your own PC.
- Adding a complete reference chapter on ZFS, the default file system on PC-BSD and Solaris.
- Adding a complete chapter on virtualization methodologies that illustrate PC-BSD jails and iocage, Solaris zones, and installation of various guest operating systems in popular host systems using VirtualBox.
- Adding many new diagrams, tables, interactive shell sessions, in-chapter tutorials, in-chapter exercises, and end-of-chapter problems.
- Providing coverage of many new commands and enhancing coverage of existing commands.
- Providing up-to-date URLs for important Web resources on nearly everything in the book.
- Enhancing the usability of all shell scripts, Python and C programs, and other programming code shown in the printed book, by installing them at a Github repository for easy download to a local repository.
- Redesigning the text layout to provide a more usable active learner document.

As in the last editions, one very important fact to keep in mind when you look at what we have included in this edition, and for that matter in the sequencing and presentation

of all the material in the book, is the fact that we have almost 65 years of practical teaching experience at the college level. Our continuing concept for this book grew out of our unwillingness to use either the large, intractable UNIX reference sources or the short "nutshell" guides to teach meaningful, complete, and relevant introductory classes on the subject. We still feel very strongly that a textbook approach, with pedagogy incorporating in-chapter tutorials and exercises, as well as useful problem sets at the end of each chapter, allows us to present all the important UNIX topics for a classroom lecture–laboratory–homework presentation. We have continued to fine-tune this textbook presentation in a manner consistent with optimal learning outcomes (i.e., well-thought-out sequencing of old and new topics, well-developed and timely lessons, online laboratory problems, and homework exercises/problems synchronized with the sequencing of chapters in the book). As in the earlier editions, because of the greatly increased depth and breadth of coverage of the basic and advanced topics we present, anyone interested in furthering their professional knowledge of the subject matter will also find this textbook useful.

THE PURPOSES OF THIS BOOK IN THE THIRD EDITION

Our primary purpose remains a didactic description of the UNIX application user's interface (AUI), and we also try to do this in a way that gives the reader insight into the inner workings of the system, along with explaining some important UNIX concepts, data structures, and algorithms. Notable examples of our revealing the inner workings of the system are the in-depth descriptions of the UNIX file, process, and I/O redirection concepts.

Our secondary purpose is to extensively describe the UNIX application programmer's interface (API) in terms of C/C++ libraries and UNIX system calls. In writing this third edition, particularly for the system programming chapters, we do assume previous basic to intermediate knowledge of C/C++ programming on the part of the reader.

The tertiary purpose of this textbook is to describe some important UNIX software engineering tools for developers of C/C++ software and shell scripts.

THE PRESENTATION FORMAT

The didactic structure of each chapter in this new edition follows one of two similar formats: either the shell session format, or the tutorial format. In the shell session format (used in all chapters except 3, 16, 22–25), the following outline is used:

- Learning objectives
- Introduction
- Topic discussion and background organized in sections and subsections
- Illustrative commands or topic illustrations presented as *shell sessions*, where the user types in commands shown and results are displayed
- In-chapter exercises that reinforce what was discussed on a topic or done interactively in a shell session

- Summary
- End-of-chapter questions and problems keyed to topics presented

In the tutorial format (used in Chapters 3, 16, 22–25) the following outline is used:

- Learning objectives
- Introduction
- Topics discussions and background organized in sections and subsections
- One or several example sessions or practice session tutorials that illustrate the commands and topics of interest in any particular section or subsection
- Illustrative commands or topic illustrations presented as shell sessions, where the user types in commands shown and results are displayed
- In-chapter exercises that reinforce what was discussed in an example or practice session, on a topic, or done interactively in a shell session
- Summary
- End-of-chapter problems keyed to topics presented

This edition has had many new diagrams and tables added, and there are many new in-chapter tutorials, interactive shell sessions, in-chapter exercises, and end-of-chapter problems. We have added more syntax boxes whenever we introduce a new command or utility. These syntax boxes describe the exact syntax of the command (and any other pertinent variants of the basic syntax), its purpose, the output produced by the command, and its useful options and features. In addition, every chapter contains a summary of the material covered in the chapter. There is also a glossary of terms used in the book.

PATHWAYS THROUGH THE TEXT

If this book is to be used as the main text for an introductory course in UNIX, Chapters 1–15 and 22 should be covered. If the book is to be used as a companion to the main text in an operating systems concepts and principles course, the coverage of chapters would be dictated by the order in which the main topics of the course are covered but should include Chapters 4, 9, 10, 18–21, 24, and 25. For use in a C/C++ or shell programming course, Chapters 1, 4–17 and relevant sections of Chapter 3 would be a great help to students. The extent of coverage of Chapter 17 would depend on the nature of the course—partial coverage in an introductory and full coverage in an advanced course. For use in a course of UNIX system administration course, Chapters 4–11, 22, 23, 24, 25, and relevant sections of Chapters 12–15 should be used. In a system programming course, Chapters 12–16, 18–21, and relevant portions of Chapters 4–11 and Chapters 24–25 should be covered. Finally, in a course on UNIX network programming, Chapters 11, 18–21, and relevant portions of Chapters 12–16 should be used.

THE DESIGN OF FONTS

The following typefaces have been used in the book for various types of text items.

Font	Text Item
Minion Pro, italic	Key terms: • Whatever directory you are currently in is known as the *present working directory*.
Minion Pro, bold	Files/directories/symbolic constants/menu paths: • The directory **first** and the file **myfile2** are now removed. • Make the Options menu choice **Save Options** • Make the pull-down menu choice **File>Quit** • A socket with **AF_INET** address family is known as the *Internet domain socket*.

`Courier Std`	Commands, program code, output of commands and programs, and options:
	• Use the `man` and `whatis` commands to find information about the `passwd` command.
	• The output of the `date` command is `Thu Apr 7 13:53:30 PKT 2016`.
	• You can use the `-l` option to display the long listing.
	• The following session shows the Bourne shell script in the **for_demo1** file
	`$ cat for_demo1`
	`#!/bin/sh`
	`for people in Debbie Jamie John Kitty Kuhn Shah`
	`do`
	`  echo "people"`
	`done`
	`exit 0`
	`$`
	Keystrokes:
	• `<Enter>`, `<Alt+V>`, `<F1>`, `a`
	Prompts, messages, dialogs, windows:
	• A user who runs a `write` or `talk` command sees the message `Permission denied`.
	• The system then displays the `login:` prompt.
	• In the `Find file:` dialog box that opens…
	• Click the OK button in the `Save` window.

`Courier Std, bold`	User input: • `[bob@pcbsd-923] ~%` **`ssh 192.168.0.8`** • `Password for bob@pcbsd-2467:` **`XXX`**

SUPPLEMENTS

A variety of supplemental materials are available for all users of this textbook and additional material only to qualified instructors.

Materials Available to all Users of this Textbook

1. You can use your web browser and retrieve the materials from the following Github repository:

   https://github.com/bobk48/unixthetextbook3

2. Or you can use the steps of the following example in the book to prepare and download these materials:

   Example 17.5: Pulling from a GitHub Repository

3. To access these materials as shown in Example 17.5, pull from the repository using this **`git`** command:

   ```
   % git pull https://github.com/bobk48/unixthetextbook3 master
   ```

4. In either case, you will find the following in your new local repository:

- Answers to in-chapter exercises.
- Source code for C/C++ programs, Python code, and long shell scripts, arranged by chapter.
- Updated links to other UNIX resources on the Web.
  - Author-maintained Web Resources listing for chapters which do not contain this in the printed book.
  - Updates of version-specific content of PC-BSD and Solaris that severely impact our printed-book presentations.
  - Errata.

Additional material is available from the CRC website at http://www.crcpress.com/product/isbn/97814822335832.

Resources Available to Qualified Instructors Only

Contact your CRC Press representative to gain access to this material.

Solutions to problems at the end of each chapter.

We take full responsibility for any errors in the book. You can send your error reports and comments to us at the above listed Github site. We will incorporate your feedback and fix any errors in subsequent printings.

Acknowledgments for the Third Edition

We started writing this third edition during the fall of 2013 and finished it in the spring of 2016. Completing such a large revision would not have been possible without the help of many. First and foremost, the authors sincerely thank the editor of the book, Randi Cohen, for her support, guidance, reassurance, professionalism, and understanding throughout this project. She is the top of the line. Also, we would like to extend a warm and gracious acknowledgment to Amber Donley and Joette Lynch at CRC Press and Michelle van Kampen at Deanta Publishing Services for all of their professional diligence and extremely thorough work on the production of this book.

We convey our sincere thanks to the following reviewers of this edition who gave valuable feedback, and numerous accurate and insightful comments. We are particularly grateful to Professor Richard Fox, who meticulously read the manuscript and gave many useful suggestions that greatly enhanced the final product.

- Hussein Abdel-Wahab — Old Dominion University
- Gregory B. Newby — Compute Canada
- Richard Fox — Northern Kentucky University

Acknowledgments for the Second and First Editions

Thanks are extended to Marilyn Lloyd of AWL for her many excellent suggestions and outstanding work in the production phase of this book. We are also grateful to Joyce Wells for designing a beautiful cover for the book. Thanks also to Katherine Harutunian at AWL for her many valuable suggestions and support. The work of the whole Addison Wesley team was excellent! Special thanks to Keith Henry and Alisa Andreola at Dartmouth Publishing for their excellent copyediting and interior design.

We also thank the following reviewers for reviewing the second edition of the book.

- Robert Albright — University of Delaware
- Hussein Abdel-Wahab — Old Dominion University
- Dunren Che — Southern Illinois University
- C. Michael Costanzo — University of California, Santa Barbara
- Robert M. Cubert — University of Florida
- James P. Durbano — University of Delaware
- Nisar Hundewale — Georgia State University
- Mark Hutchenreuther — California Polytechnic State University
- Stephen P. Leach — Florida State University
- Susan Lincke-Salecker — University of Wisconsin Parkside
- Mike Qualls — Grossmont Community College
- Daniel Tomasevich — San Francisco State University
- Paul Tymann — Rochester Institute of Technology
- Troy Vasiga — University of Waterloo

We would like to acknowledge Ronald E. Bass, Thomas A. Burns, Chuck Lesko, Toshimi Minoura, Selmer Moen, Gregory B. Newby, Dr. Marianne Vakalis, and Dr. G. Jan Wilms for their insightful comments in reviewing this book's first edition.

Personal Acknowledgments

Syed Mansoor Sarwar I thank my parents, wife, children, and siblings for their love, support, and trust. They have all been a positive influence in my life, have helped me in many ways, and remain my biggest supporters. I can never pay back the grace that they have extended to me over the years. My mother, a class act in motherhood, has been fighting serious illnesses for the past several years. I pray for her good health and peace of mind. My father, an avid reader even at the age of 94, is an icon of wisdom, courage, care, intellect, logical reasoning, and mental toughness, and remains my inspiration for the quest of knowledge discovery, rational thinking, and service to fellow humans. Without his and my wife's continuous encouragement and support, writing this book would not have been possible.

My wife, Robina, and children Maham, Ibraheem, and Hassan have been extremely patient and supportive during the course of this project. Thank you for your understanding and support, guys! I could not have done it without you. At the time of writing the second edition of the book, Hassan and Maham were in middle school, and the then "snug bug" Ibraheem in prenursery. Now, Hassan is the CEO of *infinione* (infinione.com), a Los Angeles-based technology company that he founded when he was a junior at the University of Southern California (USC). Maham is a merit scholarship holder senior, majoring in textile design with the Best Designer award under her belt in a national competition. Several of her designs have already been marketed through well-known fashion women's wear brands such as Beechtree. Ibraheem is now a young 6′2″ teenager, with a love for mathematics and science, and has the goal to become a top-notch scientist as well as a professional basketball player.

My special thanks to my sisters Rizwana and Farhana, and brothers Aqeel, Nadeem, Masood, and Nabeel for their friendship and care. I convey my sincere gratitude to them, my sisters-in-law Maimoona, Sadaf, and Farzana, and brother-in-law Hamid, for taking care of our father and ailing mother during the hours of their need. Folks, I will forever remain indebted to you for performing my share of service toward our parents. I hope to be able to offer a payback in some form someday.

I thank the teachers who taught me the use, administration, internals, and programming of UNIX. They are James Davis, Doug Jacobson, and Arthur Oldehoeft at Iowa State University, and Jim Binkley at Oregon Graduate Institue and Portland State University. I wrote this book under the most trying professional circumstances of my career. I thank my colleagues at the Punjab University College of Information Technology (PUCIT), the

GIS Center, the Institute of Business and Information Technology (IBIT), Institute of Chemistry, and the Department of Space Science who supported me through thick and thin. I wish them all peaceful lives and outstanding professional careers. Fellows, you are the best colleagues one could ever wish for. Thank you for being there!

I also thank my friends, colleagues, teachers, students, and professionals who have played a positive role in my life. They are in no particular order: R. A. Khan, Mohammad Ishaq, the late Ghulam us Saqlain Naqvi, Shahid H. Bokhari, Mohammad Ali Maud, Haroon Babri, Wasif Khan, Norman Scott, Janice Jenkins, Art Pohm, James Davis, Arthur Oldehoeft, Thomas Piatkowski, Douglas Jacobson, the late James "Jolly" Triska, the late Charlie Wright, David Schmidt, Terry Smay, Zartash Uzmi, Tariq Jadoon, Sohaib Khan, Aamer Mahmood, Asim Loan, the late Masood Ahmad, Sohail Aftab, Jahangir Ikram, Tariq Butt, Asim Rasul, Hamid Ch, Ejaz Ashraf, Shahzad Sarwar, Murtaza Yousaf, Waqar Jaffrey, Kamran Malik, Sadeeqa Khan, Shahid Farid, Hassan Khan, Sidra Faisal, Zobia Sohail, Laeeq Aslam, Faisal Bukhari, Waheed Iqbal, Faisal Aslam, Fawaz Bokhari, Nastaeen Fatima, Muddassira Arshad, Esha Aftab, Fakhra Jabeen, Aisha Khan, Amina Mustanser, Tayyaba Tariq, Saima Ali, Umair Babar, Imran Farid, Mehvish Poshni, Khurram Shahzad, Zubair Nawaz, Adeel Nisar, Ahmad Ghazali, Asif Sohail, Muhammad Farooq, Kashif Murtaza, Nazar Khan, Aasim Ali, Qudsia Hamid, Muhammad Haris, Nida Samad, Zia Afzal, Syed Muhammad Ali, Samreen Jawed, Anzar Ahmad, Rukhsana Kauser, Madiha Khalid, Mian Mubasher, Sanam Ahmad, Sarah Riaz, Asim Siddique, Tariq Saeed, Mehwish Khurshid, Adnan Asif, Saleem Akhtar, Aamir Raza, Mohsin Omer, Saeed Abbasi, Muhammad Shafiq, Arif Kareem, Muhammad Ajmal, Athar Pasha, Anisul Haq, Shamaila Gull, Javed Sami, Ahsan Sharif, Shaukat Ali, Aurangzeb Alamgir, the late Saeed Nagra, Shahid Ghazi, Asghar Iqbal, Fauzia Adeeba, Maryam Nawaz, Khawaja Imran Nazir, Shahzad Shaukat, Syed Mazahir Ali Akbar Naqvi, Saad Rasool, Anum Azhar, Aitzaz Chaudhary, Aziz Inan, Zia Yamayee, Shahid Ghazi, Asghar Iqbal, Fauzia Adeeba, Maryam Nawaz, Kitty Tilton, Jamie Strohecker, Debbie Spear, Lorraine Yoder, Mathew Kuhn, Tanya Crenshaw, Peter Osterberg, Robert Albright and, last but not least, my mentor at the University of Portland, the late Tom Nelson.

I thank the many students at the Iowa State University, University of Portland, Portland Community College, LUMS, and PUCIT who helped me in the development of the material for the book. Thanks also to my colleagues at the University of Portland, LUMS, and PUCIT who helped me with the installation and use of the UNIX systems for the three editions of the book. They are Kent Thompson, Dale Frakes, Abid Latif "Pasoorri", Arif Butt, and Rizwan Malik. My office staff Zubair Siddique, Muhammad Azhar, and Humayun Ejaz helped me with all the diagrams in this edition of the book. Thank you, gentlemen!

Last but not least, I also thank my coauthor, Bob Koretsky, for his help, encouragement, and trust. Bob, you were a great support throughout the project. Completing this task was not possible without you! I look forward to working with you on future projects, including the second edition of *Linux: The Textbook*.

Robert M. Koretsky I thank my family, for all of the love and continued support through the years. Special thanks to my grandsons Victor and Garvey. I also thank Mansoor Sarwar, for his friendship, intelligence, patience, and continued support and inspiration.

CHAPTER 1

Overview of Operating Systems

Objectives

- To explain what an operating system is
- To describe briefly operating system services
- To describe character and graphical user interfaces
- To discuss different types of operating systems
- To describe briefly the UNIX operating system
- To give an overview of the structure of a contemporary system
- To describe briefly the structure of the UNIX operating system
- To detail some important system setups
- To describe briefly the history of the UNIX operating system
- To provide an overview of the different types of UNIX systems

1.1 INTRODUCTION

Many operating systems are available today, some general enough to run on any type of computer (from a personal computer, or PC, to a mainframe), and some specifically designed to run on a particular type of computer system, including real-time computer systems used to control the movement of mechanical devices such as robots, tablet computers, and cell phones. In this chapter, we describe the purpose of an operating system and the different classes of operating systems. Before describing different types of operating systems and where UNIX fits in this categorization, we present a layered diagram of a contemporary computer system and discuss the basic purpose of an operating system.

We then describe different types of operating systems and the parameters used to classify them. Then, we identify the class that UNIX belongs to and briefly discuss the different members of the UNIX family.

The people who use UNIX comprise application developers, systems analysts, programmers, administrators, business managers, academicians, and people who just wish to read their e-mail. From its earliest inception in 1969 as a laboratory research tool, it was further developed in the academic community, and then endorsed for commercial uses. In its version today, UNIX has an underlying functionality that is complex but easy to learn, and extensible yet easily customized to suit a user's style of computing. One key to understanding its longevity and its heterogeneous appeal is to study the history of its evolution.

1.2 WHAT IS AN OPERATING SYSTEM?

A computer system consists of various hardware and software resources, as shown in a layered fashion in Figure 1.1. The primary purpose of an operating system is to facilitate easy, efficient, fair, orderly, and secure use of these resources. It allows the users to employ application software—spreadsheets, word processors, Web browsers, e-mail software, and other programs. Programmers use language libraries, system calls, and program generation tools (e.g., text editors, compilers, and version control systems) to develop software. Fairness is obviously not an issue if only one user at a time is allowed to use the computer system, including single-user desktop systems, laptops, tablet computers, and cell phones. However, if multiple users are allowed to use the computer system, fairness and security are two main issues to be tackled by the operating system designers.

Hardware resources include keyboards, touch pads, display screens (may also be touch screens), main memory (commonly known as *random access memory* or RAM), disk drives, network interface cards (NICs), and central processing units (CPUs). Software resources include applications such as word processors, spreadsheets, games, graphing

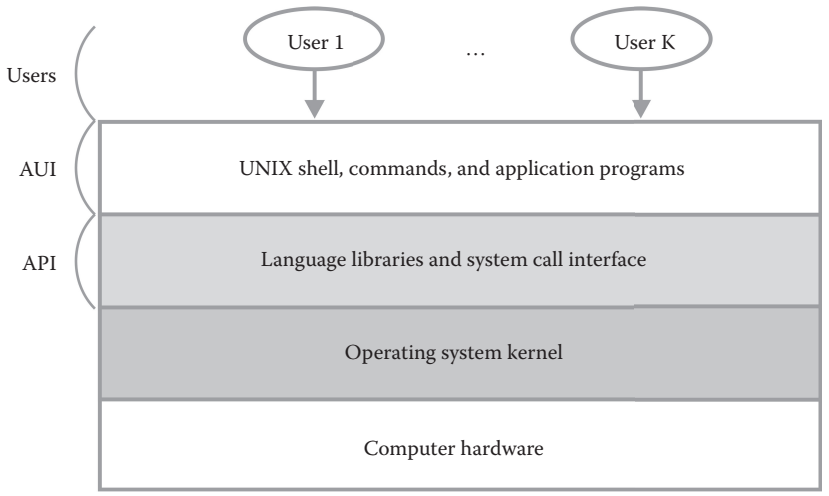

FIGURE 1.1 A layered view of a contemporary computer system.

tools, picture- and video-processing tools, and Internet-related tools such as Web browsers. These applications, which reside at the topmost layer in the diagram, form the application user interface (AUI). The AUI is glued to the operating system kernel via the language libraries and the system call interface. The system call interface comprises a set of functions that can be used by the applications and library routines to execute the kernel code for a particular service, such as reading a file. The language libraries and the system call interface comprise what is commonly known as the *application programmer interface* (API). The kernel is the core of an operating system, where issues like CPU scheduling, memory management, disk scheduling, and interprocess communication are handled. The layers in the diagram are shown in an expanded form for the UNIX operating system in Figure 1.2, where we also describe them briefly.

There are two ways to view an operating system: top down and bottom up. In the bottom-up view, an operating system can be viewed as a software that allocates and deallocates system resources (hardware and software) in an efficient, fair, orderly, and secure manner. For example, the operating system decides how much RAM space is to be allocated to a program before it is loaded and executed. The operating system ensures that only one file is printed on a particular printer at a time, prevents an existing file on the disk

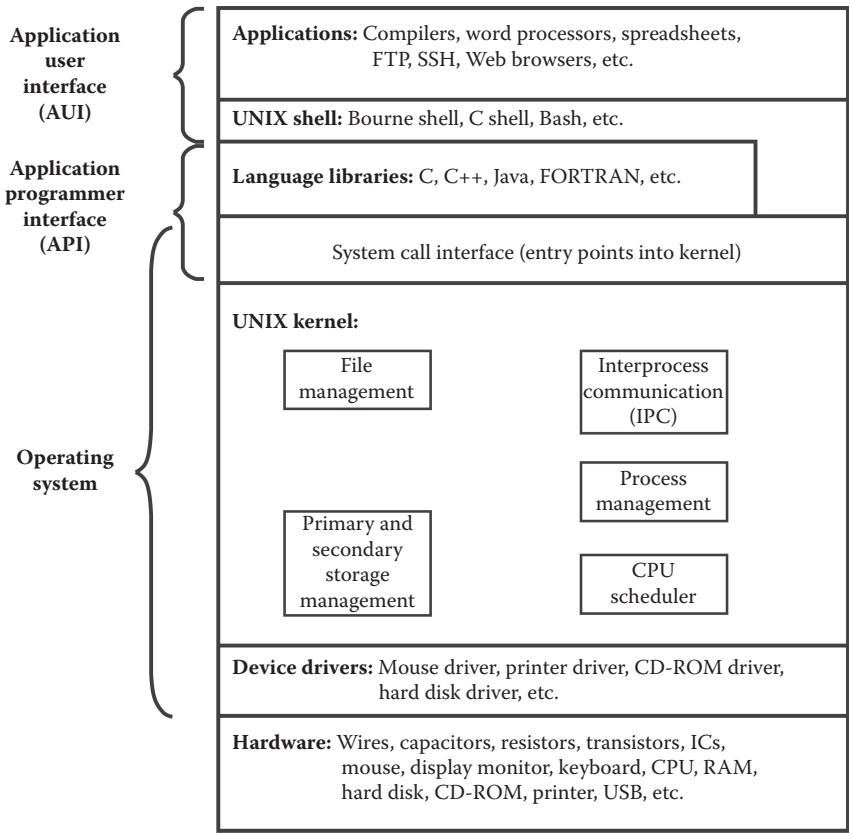

FIGURE 1.2 Software architecture of the UNIX operating system.

from being accidentally overwritten by another file, and further guarantees that, when the execution of a program given to the CPU for processing has been completed, the program relinquishes the CPU so that other programs can be executed. Thus the operating system can be viewed as a resource manager.

In the top-down view, which we espouse in this textbook, an operating system can be viewed as a piece of software that isolates you from the complications of hardware resources. You therefore do not have to deal with the extremely difficult (and sometimes impossible for most users) task of interacting with these resources. For example, as a user of a computer system, you don't have to write the code that allows you to save your work as a file on a hard disk, use a mouse as a point-and-click device, use a touch screen or touch pad, or print on a particular printer. Also, you do not have to write new software for a new device (e.g., mouse, disk drive, or DVD) that you buy and install in your system. The operating system performs the task of dealing with complicated hardware resources and gives you a comprehensive machine with a simple, ready-to-use interface. This machine allows you to use simple commands to retrieve and save files on a disk, print files on a printer, and play movies from a DVD. In a sense, the operating system provides a virtual machine that is much easier to deal with than the physical machine. You can, for example, use a command such as `cp memo letter` to copy the memo file to the letter file on the hard disk in your computer without having to worry about the location of the memo and letter files on the disk, the structure and size of the disk, the brand of the disk drive, and the number or name of the various drives (floppy, CD-ROM, and one or more hard drives) on your system.

1.3 OPERATING SYSTEM SERVICES

An operating system provides many ready-made services for users. Most of these services are designed to allow you to execute your software, both application programs and program development tools, efficiently and securely. Some services are designed for housekeeping tasks, such as keeping track of the amount of time that you have used the system. The major operating system services therefore provide mechanisms for following secure and efficient operations and processes:

- Execution of a program
- Input and output operations performed by programs
- Communication between processes
- Error detection and reporting
- Manipulation of all types of files
- Management of users and security

A detailed discussion of these services is outside the scope of this textbook, but we discuss them briefly when they are relevant to the topic being presented.

1.4 CHARACTER (COMMAND LINE) VERSUS GRAPHICAL USER INTERFACES

In order to use a computer system, you have to give commands to its operating system. An input device, such as a keyboard, is used to issue a command. If you use the keyboard to issue commands to the operating system, the operating system has a character user interface (CUI), commonly known as the *command line interface*. If the primary input device for issuing commands to the operating system is a point-and-click device, such as a mouse, a touch screen, or a touch pad, the operating system has a graphical user interface (GUI). Most, if not all, operating systems have both character and graphical user interfaces, and you can use either. Some have a command line as their primary interface but allow you to run software that provides a GUI. Operating systems such as DOS and UNIX have CUIs, whereas Mac OS, OS/2, and Microsoft Windows primarily offer GUIs but have the capability to allow a user to enter a DOS- or UNIX-like terminal screen. Although UNIX comes with a CUI as its basic interface, it can run software based on the X Window System (Project Athena, MIT) that provides a GUI interface. Moreover, most UNIX systems now have a state-of-the-art X-based GUI. Mac OS X (Darwin), running on Apple products, is the most well-known GUI-based UNIX system. We discuss the UNIX GUI in Chapter 23.

Although a GUI makes a computer easier to use, it gives you an automated setup with reduced flexibility. A GUI also presents an extra layer of software between you and the task that you want to perform on the computer, thereby making the task slower. In contrast, a CUI gives you ultimate control of your computer system and allows you to run application programs any way you want. A CUI is also more efficient because a minimal layer of software is needed between you and your task on the computer, thereby enabling you to complete the task faster. It is also malleable and gives the user more control. Because many people are accustomed to the graphical interfaces of popular gizmos and applications such as Nintendo and Web browsers, the character interface presents an unfamiliar and sometimes difficult style of communicating commands to the computer system. However, computer science students are usually able to meet this challenge after a few hands-on sessions.

1.5 TYPES OF OPERATING SYSTEMS

Operating systems can be categorized according to the number of users who can use the system at the same time and the number of processes (executing programs) that the system can run simultaneously. These criteria lead to three types of operating systems:

- *Single-user, single-process system*: These operating systems allow only one user at a time to use the computer system, and the user can run only one process at a time. Such operating systems are commonly used for PCs. Examples of these operating systems are earlier versions of Mac OS, DOS, and many of Microsoft's Windows operating systems.
- *Single-user, multiprocess system*: As the name indicates, these operating systems allow only a single user to use the computer system, but the user can run multiple processes simultaneously. These operating systems are also used on PCs. Examples

of such operating systems are OS/2, Windows XP Workstation, and batch operating systems. Batch processing is still commonly used in mainframe computers, and most modern operating systems including UNIX, Microsoft Windows, Linux, and Mac OS perform some tasks in batch mode. Even smartphone operating systems including Android and iOS perform tasks in batch mode.

- *Multiuser, multiprocess system*: These operating systems allow multiple users to use a computer system simultaneously, and every user can run multiple processes at the same time. These operating systems are commonly used on computers that support multiple users in organizations such as universities and large businesses. Examples of these operating systems are UNIX, Linux, Windows NT Server, MVS, and VM/CMS.

Multiuser, multiprocess systems are used to increase resource utilization in the computer system by multiplexing expensive resources such as the CPU. This capability leads to increased system throughput (the number of processes finished in unit time). Resource utilization increases because, in a system with several processes, when one process is performing input or output (e.g., reading input from the keyboard, capturing a mouse click, or writing to file on the hard disk), the CPU can be taken away from this process and given to another process—effectively running both processes simultaneously by allowing them both to make progress (one is performing input/output [I/O] and the other is using the CPU). The mechanism of assigning the CPU to another process when the current process is performing I/O is known as *multiprogramming*. Multiprogramming is the key to all contemporary multiuser, multiprocess operating systems. In a single-process system, when the process using the CPU performs I/O, the CPU sits idle because there is no other process that can use the CPU at the same time.

Operating systems that allow users to interact with their executing programs are known as *interactive operating systems*, and the ones that do not are called batch operating systems. Batch systems are useful when programs are run without the need for human intervention, such as systems that run payroll programs. The VMS operating system has both interactive and batch interfaces. Almost all well-known contemporary operating systems (UNIX, Linux, DOS, Windows, etc.) are interactive. UNIX and Linux also allow programs to be executed in batch mode, with programs running in the background (see Chapter 10 for details of "background process execution" in UNIX). Multiuser, multiprocess, and interactive operating systems are known as *time-sharing systems*. In time-sharing systems, the CPU is switched from one process to another in quick succession. This method of operation allows all the processes in the system to make progress, giving each user the impression of sole use of the system. Examples of time-sharing operating systems are UNIX, Linux, and Windows NT Server.

1.6 THE UNIX FAMILY

Years ago the name UNIX referred to a single operating system, but it is now used to refer to a family of operating systems that are offshoots of the original in terms of their user interfaces. Éric Lévénez (www.levenez.com/unix) lists names of over 270 UNIX flavors at the time of writing this book. Some of the members of this family are AIX, BSD, DYNIX,

FreeBSD, HP-UX, Linux, MINIX, NetBSD, SCO, Solaris, OpenSolaris, SunOS, System V, XENIX, PC-BSD, OpenBSD, Mac OS X (Darwin), and XINU. In Section 1.8, we give a brief history of some of the most popular and developmentally influential UNIX systems.

1.7 UNIX SOFTWARE ARCHITECTURE

Figure 1.2 shows a layered diagram for a UNIX-based computer system, identifying the system's software components and their logical proximity to the user and hardware. We briefly describe each software layer from the bottom up.

1.7.1 Device Driver Layer

The purpose of the device driver layer is to interact with various hardware devices. It contains a separate program for interacting with each device, including the hard disk driver, floppy disk driver, CD-ROM driver, keyboard driver, mouse driver, touch pad driver, and display driver. These programs execute on behalf of the UNIX kernel when a user command or application needs to perform a hardware-related operation such as a file read that translates to one or more disk reads. The user doesn't have direct access to these programs and therefore can't execute them as commands.

1.7.2 UNIX Kernel

The UNIX kernel layer contains the actual operating system. Some of the main functions of the UNIX kernel, listed in Figure 1.2, are described in this section. In addition, the kernel performs several other tasks for fair, orderly, and safe use of the computer system. These tasks include managing the CPU, printers, and other I/O devices. The kernel ensures that no user process takes over the CPU forever, that multiple files are not printed on a printer simultaneously, and that a user cannot terminate another user's process.

1.7.2.1 Process Management

This part of the kernel manages processes in terms of creating, suspending, resuming, and terminating them, and maintaining their states. It also provides various mechanisms for processes to communicate with each other and schedules the CPU to execute multiple processes simultaneously in a time-sharing system. *Interprocess communication* (IPC) is the key to the client–server-based software that is the foundation for Internet applications, including Web browsing (HTTP), file transfer (FTP), and remote login (SSH). The UNIX system provides three primary IPC mechanisms/channels:

- *Pipe*: Two or more related processes running on the same computer can use a pipe as an IPC channel. Typically, these processes have a parent–child or sibling relationship. A pipe is a temporary channel that resides in the main memory and is created by the kernel, usually on behalf of the parent process.
- *Named pipe*: A named pipe, also known as a FIFO, is a permanent communication channel that resides on the disk and can be used for IPC by two or more related or unrelated processes running on the same computer.

- *BSD socket*: A BSD socket is also a temporary channel that allows two or more processes in a network (or on the Internet) to communicate, although processes on the same computer can also use them. Sockets were originally a part of the BSD UNIX only, but they are now available on almost every UNIX system. Internet software such as Web browsers, File Transfer Protocol (FTP), Secure Shell (SSH), and electronic mailers are implemented by using sockets. AT&T UNIX has a similar mechanism called the Transport Layer Interface (TLI).

We discuss these mechanisms of IPC in detail under "UNIX System Programming" in Chapters 18 through 21.

1.7.2.2 File Management

This part of the kernel manages files and directories, also known as folders. It performs all file-related tasks, including file creation and removal, directory creation and removal, setting access privileges on files and directories, and maintaining their attributes, such as file size. A file operation usually requires manipulation of a disk. In a multiuser system, a user must never be allowed to manipulate a disk directly because it contains files belonging to other users, and user access to a disk poses a security threat. Only the kernel must perform all file-related operations, such as file removal. Also, only the kernel must decide where and how much space to allocate to a file.

1.7.2.3 Main Memory Management

This part of the kernel allocates and deallocates RAM in an orderly manner so that each process has enough space to execute properly. It also ensures that part or all of the space allocated to a process does not belong to some other process. The space allocated to a process in the memory for its execution is known as its *address space*. The kernel ensures that no process accesses an area of memory that does not belong to its address space. The kernel maintains areas in the main memory that are free to be allocated to processes. The kernel code that performs this task is called the *free space manager*. When a program is to be loaded in the main memory, the free space manager allocates adequate space for it and the *loader* loads the program into this space. The kernel also records where all the processes reside in the memory so that, when a process tries to access main memory space that does not belong to it, the kernel can terminate the process and give a meaningful message to the user. When a process terminates, the kernel deallocates the space allocated to the process and puts it back in the free space pool so that it can be reused.

1.7.2.4 Disk Management

The kernel is also responsible for maintaining free and used disk space and for the orderly and fair allocation and deallocation of disk space. It decides where and how much space to allocate to a newly created file. The kernel code that performs this task is known as the *disk storage manager*. Also, the kernel performs *disk scheduling*, deciding which request to serve next when multiple requests for file read, write, and so on, arrive for the same disk.

1.7.3 System Call Interface

The system call interface layer contains entry points into the kernel code. Because the kernel manages all system resources, any user or application request that involves access to any system resource must be handled by the kernel code. But user processes must not be given open access to the kernel code for security reasons. So that user processes can invoke the execution of kernel code, UNIX provides several openings, or function calls, into the kernel, known as *system calls*. There are numerous system calls that deal with the manipulation of processes, files, and other system resources. These calls are well tested, and most of them have been used for several years, so their use poses much less of a security risk than if any user code were allowed to perform the task.

1.7.4 Language Libraries

A *language library* is a set of prewritten and pretested functions in a programming language available to programmers for use with the software that they develop. The availability and use of libraries saves time because programmers do not have to write these functions from scratch. This layer contains libraries for several languages, including C, C++, C#, Java, Perl, and Python. For the C language, for example, there are several libraries, including a string library (which contains functions for processing strings, such as a function for comparing two strings) and a math library (which contains functions for mathematical operations, such as finding the cosine of an angle).

As we stated earlier in this chapter, the libraries and system call interface form what is commonly known as the API. In other words, programmers who write software in a language such as C can use in their code the prewritten functions available in the various C libraries and system calls.

1.7.5 UNIX Shell

The *UNIX shell* is a program that starts running when you log on and interprets the commands that you enter. The most popular shells are the Bourne shell (`sh`), Bourne Again shell (`bash`), C shell (`csh`), TC shell (`tcsh`), and Korn shell (`ksh`). We show the usage of shell commands and shell scripts (see Chapters 12 through 15) in Bourne and C shells.

1.7.6 Applications

The applications layer contains all the applications (tools, commands, and utilities) that are available for your use. A typical UNIX system contains hundreds of applications; we discuss the most useful and commonly used applications throughout this textbook. When an application that you're using needs to manipulate a system resource (e.g., reading a file), it needs to invoke some kernel code that performs the task. An application can find the appropriate kernel code to execute in one of two ways: (1) by using a proper library function and (2) by using a system call. Library calls constitute a higher-level interface to the kernel than system calls, which makes library calls a bit easier to use. However, all library calls eventually use system calls to begin execution of the appropriate kernel code. Therefore, the use of library calls in software results in slightly slower execution. A detailed discussion of language libraries and system calls is, generally, beyond the scope of this

textbook. However, we discuss and show the use of several library calls and system calls in Chapter 16 on Python and Chapters 18 through 21 on UNIX system programming.

The user can use any command or application that is available on the system. As we mentioned earlier in this chapter, this layer is commonly known as the AUI.

1.8 DEVELOPMENT OF THE UNIX OPERATING SYSTEM

How has UNIX achieved the status and position in the marketplace it has now? Because it is a multiuser, multiprocess operating system that can run on the X86 architectures of very small- to very large-scale hardware used by most computer users in the world. That means it is used by casual, individual users on their home computers, all the way up to 60 processor servers in a cloud configuration at a commercial facility. It can link your home computer to the Internet with a standard browser like Opera or Firefox. Also, the vast majority of Internet servers run on UNIX or Linux machines.

What really are the differences between the three major families of UNIX? Basically, the kernels and their APIs, the file systems, the device driver bases, and to some extent the desktop management systems they use. And accordingly, in reality there are only two true UNIX operating system families now: the BSD family, exemplified in our book by the FreeBSD offshoot named PC-BSD, and the Solaris family. Both families have the distinguishing feature of booting from and including the Z File system (ZFS) in their kernel. At the time of writing this book, none of the other NIX-like systems (Linux, OS X) have this feature.

The people who use the UNIX operating system are application developers, systems analysts, web programmers, system administrators, business managers, academicians, and people who just want to read their e-mail. From its earliest inception in 1969 as a laboratory research tool, it was further developed in the academic community and then endorsed for commercial uses. Today's UNIX has an underlying functionality that is complex but easy to learn because of its GUI, and extensible yet easily customized to suit a user's style of computing. The GUI is comparable to those on Windows or OS X machines. One of the primary keys to understanding its longevity, and its heterogeneous appeal, is to study the history of its evolution, which follows.

1.8.1 Beginnings

Before we describe the evolution of UNIX, first we have to ask, why is this operating system so "friendly" and accommodating? Part of the answer is that this ever-evolving operating system, which is accepted and used throughout the world, was developed in response to the needs and activities of a very heterogeneous community of computer users. It grew, changed, and improved because of the work and cooperation of many diverse, and sometimes opposing, individuals and groups.

UNIX continuously grew, changed, and improved alongside the development of computer hardware, software applications, networking, and other components of the "computer revolution." The UNIX project started as a personal and subjective endeavor but exploded into a universal and generic technical tool. Thus its various audiences must have found some basic advantages in this tool—particularly the largest audience, common users. Separating the influences of these various user groups in the development of UNIX

is difficult. Moreover, because the system is fundamentally an open software system—that is, the source code is freely distributed among the community of users—its evolution has been shaped to some extent by a populist mindset. For example, development resource and source code repositories such as GitHub expedite this development model in the twenty-first century. It will continue to be shaped as such in the future, thanks to organizations like the Open Software Foundation, and because of the pervasive use of the Internet in social life, academic settings, and in business and professional settings.

It is the underlying core functionality of UNIX that brings together its diverse audiences into a community—not so much in the sociological sense, but more in an independent, DIY, intellectual sense. As you delve into the subject matter of this textbook, you might wonder where you fit into the UNIX community and how its functionality might be adapted for your uses. Essentially, it is the style of your interaction with the computer that will be the most important, invigorating, and critical aspect of your work with the UNIX operating system.

The development of other contemporary operating systems is motivated and informed by completely different forces and bases (primarily commercialization) than those that motivated the inception and development of UNIX (primarily a user-friendly, text-based operating system). The history of UNIX is a record of how a system should be developed, regardless of how you believe that system should be structured, how you think it should function (whatever your user perspective), and who you believe should control that development.

Figure 1.3 describes the three main branches of UNIX systems as they were developed from 1969 to the present. The approximate dates of the development of milestone versions in each of the branches are shown on the left.

The UNIX Support Group (USG), UNIX System Development Laboratory (USDL), and UNIX System Laboratories (USL) were commercial spin-offs of AT&T. The UNIX Programmer's Work Bench (PWB) was distributed initially through the USG.

In the mid-1960s, Bell Laboratories began a collaborative effort to develop a multiuser operating system known as MULTICS. One of the biggest drawbacks inherent in the functionality of this new operating system was the complexity of the software and hardware required to accomplish simple tasks for multiple users. Following the failure of the MULTICS project, Ken Thompson, Dennis Ritchie, and others at Bell Laboratories developed a multiuser operating system called UNIX, which first ran on a DEC PDP-7 computer and later was ported to a PDP-11 computer. One of the features of UNIX that distinguished it from MULTICS was that it allowed processes to be created easily by a single user.

The most important historical development in the early 1970s was the recoding of most of the operating system in the high-level programming language, C. At that time, most operating system programs were written in a low-level programming language, known as an *assembly language*, which was specifically tailored to the architecture of the processor that a particular make of computer used. Thus, an operating system written in a low-level language was not portable between computers with different processors made by different manufacturers. Written in C, UNIX was very portable. Also, C, as well as other high-level languages, is much easier to program with than assembly language, which is characteristically difficult.

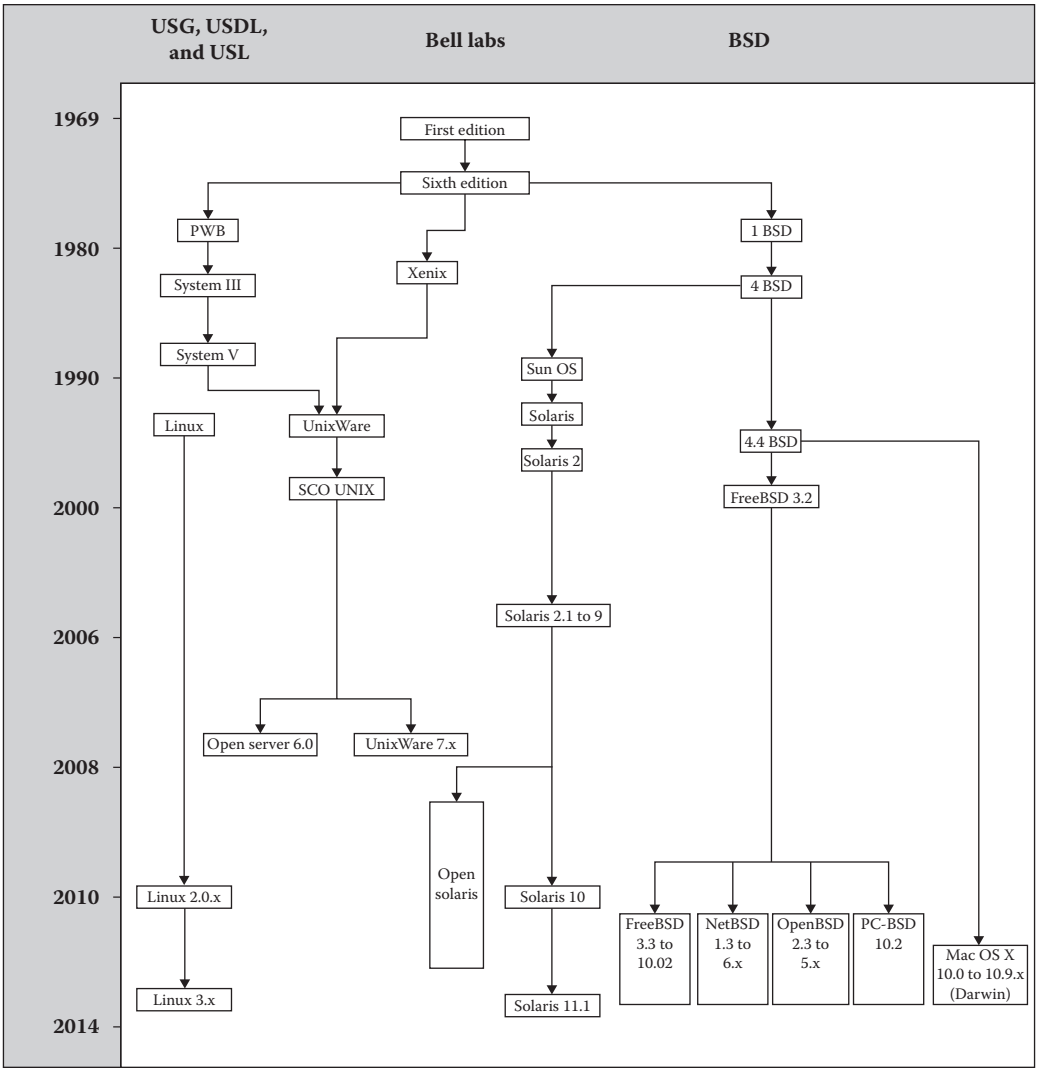

FIGURE 1.3 Schematic UNIX time line.

1.8.2 Research Operating System

Bell Laboratories controlled the research systems versions of UNIX, known as versions 1 through 6. These versions had three important characteristics:

1. The UNIX system was continually developed and written in C, with only a small subset of the code tailored to a target processor.
2. Releases were distributed as C source code, which could be easily modified and improved upon to add functionality by those who obtained any of the research versions of the system.

3. The design of the system allowed users to run multiple processes concurrently and to connect these processes with IPC channels, including pipes, FIFOs, and sockets, as discussed in Section 1.7.2. We present the implementation of this design aspect in Chapter 9.

1.8.3 AT&T System V

In response to the changing business environment in the early 1980s, Bell Laboratories/AT&T licensed further releases of UNIX as System III and finally as System V, starting in 1983. This main branch of UNIX continued to be developed, as shown in Figure 1.3, through System V, Release 4 (SVR4), when it again diverged and evolved to survive as SCO UNIX in the mid- to late 1990s. Currently, there are four major System V–based UNIX systems: AIX 7.x, OpenServer 6.x, UnixWare 7.x, Solaris 11.x, OpenSolaris and its variants, and HP-UX 11i v3.

1.8.4 Berkeley Software Distributions

The University of California at Berkeley initiated and maintained the development of UNIX along its second main branch throughout the 1980s and into the 1990s. Contractual agreements made the operating system freely available to universities, so these releases contributed in large part to the popularization of UNIX. These versions were released as Berkeley Software Distributions (BSD), 3BSD, and 4BSD–4.4BSD. Most recently, BSD UNIX survives as FreeBSD, and its offshoot PC-BSD, OpenBSD, and NetBSD. Today, the most popular variants of BSD UNIX are FreeBSD 10.x, OpenBSD 5.x, NetBSD 6.x, and Mac OS X 10.9.x (Darwin).

In this book, we show example commands primarily under PC-BSD. Where appropriate, we also discuss features of Solaris.

1.8.5 History of Shells

The development of the shell as a UNIX utility parallels the development of the system itself. Steven R. Bourne wrote the first commercially available shell, the Bourne shell. Available in the seventh edition in 1979, it is the default shell on many System V versions. The C shell, written in the late 1970s primarily by Bill Joy, was made available soon after in 2BSD. When introduced, it provided a C program–like programming interface for writing shell scripts. Following the development of the C shell, the Korn shell was introduced officially in SVR4 in 1986. Written by David Korn of Bell Laboratories, it included a superset of Bourne shell commands but had more functionality. It also included some useful features of the C shell. We discuss the development history of the UNIX shell in a bit more detail in Chapter 2.

The three major shells have slightly different features and command sets. In this textbook, we discuss common features and command sets for all UNIX shells and versions. Whenever we discuss a feature or command that is particular to a shell or version, we state that specifically.

1.8.6 Current and Future Developments

Probably the most exciting and challenging current UNIX development for all other flavors of UNIX (besides Solaris and FreeBSD) focuses on the incorporation of the ZFS into

the kernel, and having the boot disk use ZFS. ZFS was developed by Sun Microsystems in the years prior to its incorporation into the Solaris family in 2006. It is now the standard file system in only Solaris (and its noncommercial equivalent, OpenIndiana) and FreeBSD (and its offshoot PC-BSD). Since the purchase of Sun by Oracle, the source development of ZFS has proceeded basically along two branches: the Oracle Solaris branch and the open branch.

Another major challenge on the horizon for UNIX systems is the incorporation of systemd into the kernel. Currently, systemd is a suite of system management daemons, libraries, and utilities designed as a central management and configuration platform for Linux. systemd is used on a majority of the current implementations and official releases of the Linux kernel. It is a Linux init system (the process called on by the Linux kernel to initialize the user space during the Linux startup process and manage all processes afterwards), thus replacing the UNIX System V and BSD-style `init` daemon. The name systemd adheres to the convention of making daemons easier to distinguish by having the letter d as the last letter of the filename. Whether or not systemd will be incorporated into the UNIX kernel remains to be seen.

Finally, the replacement of the X Window System protocol by various other software systems promises to yield a smaller, more effective GUI system. Wayland is a protocol that specifies the communication between a display server (called a Wayland compositor) and its clients, as well as a reference implementation of the protocol in C.

1.9 VARIATIONS IN UNIX SYSTEMS

As shown in Figure 1.3, the development of the UNIX systems proceeded along three main branches from a single core. Many of the branches' divergences and similarities were caused by the Bell Laboratories and AT&T legal licensing arrangements during the 1970s and 1980s. The primary advantage of the divergences was a command- and function-rich operating system in each of the branches. The early Bell Labs releases were copied and distributed freely as source code, which academic and commercial users could easily modify to suit their hardware and software. Such adaptations led to a proliferation of ways in which various aspects of the operating system evolved. Even the later releases of System V and BSD could be modified easily via accommodations provided by the vendor of the operating system version, even if the source code was not available. Many of the later releases were compatibility releases meant to provide uniformity between any particular implementation and its perceived competitors. The important contribution of these compatibility releases and their offshoots is a helpful amount of homogeneity, regardless of whether you use a modern derivative of SVR4, Solaris, 4.4BSD, Linux, or Apple OS X.

Divergence has the drawback that programs and even commands that work on one version fail to work on another, thus defeating the inherent strength of user-friendliness of the system itself. Attempts have been made to standardize UNIX—for example, via the IEEE Portable Operating System Interface (POSIX). This software standard not only covers UNIX, but also in particular specifies program operation and user interfaces, leaving their implementations to the developer. Several standards have been adopted, and more

have been proposed. For example, adopted POSIX standards specify the shell and utility standardization.

By far the most inclusive and wide-ranging standardization mechanism is the Single UNIX Specification (SUS). This is a family of standards for computer operating systems, compliance with which is required to qualify for the name UNIX. The core specifications of the SUS are developed and maintained by the Austin Group, which is a joint working group of IEEE, ISO/IEC JTC 1/SC 22, and the Open Group.

SUMMARY

An operating system is software that runs on the hardware of a computer system to manage its hardware and software resources. It also gives the user of the computer system a simple, virtual machine that is easy to use. The basic services provided by an operating system offer efficient and secure program execution, I/O operations, communication between processes, error detection and reporting, and file manipulation.

Operating systems are categorized by the number of users that can use a system at the same time and the number of processes that can execute on a system simultaneously: single-user single-process, single-user multiprocess, and multiuser multiprocess operating systems. Furthermore, operating systems that allow users to interact with their executing programs (processes) are known as *interactive systems*, and those that do not are called batch systems. Multiuser, multiprocess interactive systems are known as *time-sharing systems*, of which UNIX is a prime example. The purpose of multiuser, multiprocess systems is to increase the utilization of system resources by switching them among concurrently executing processes. This capability leads to higher system throughput, or the number of processes finishing in unit time.

In order to use a computer system, the user issues commands to the operating system. If an operating system accepts commands via the keyboard, it has a CUI. If an operating system allows users to issue commands via a point-and-click device such as a mouse, it has a GUI. Although UNIX comes with a CUI as its basic interface, it can run software based on the X Window System (Project Athena, MIT) that provides a GUI. Most UNIX systems now have both interfaces. Mac OS X (Darwin), running on Apple products, is the most well-known GUI-based UNIX system.

A computer system consists of several hardware and software components. The software components of a typical UNIX system consist of several layers: applications, shell, language libraries, system call interface, UNIX kernel, and device drivers. The kernel is the main part of the UNIX operating system and performs all the tasks that deal with allocation and deallocation of system resources. The shell and applications layers contain what is commonly known as the AUI. The language libraries and the system call interface contain the API.

The historical development of UNIX is characterized by an open systems approach, whereby the source code was freely distributed among users. Development of many versions of UNIX progressed along three main branches. Two of these branches, Oracle Solaris and FreeBSD, can best be characterized as commercial and academic. Compatibility releases of various versions have been aimed at standardizing the system. The POSIX and the SUS are related standardization efforts. Two exciting and challenging new developments in the

future of true UNIX systems are incorporation of ZFS into the kernel, and the replacement of the X Window System protocol with systems such as Wayland.

QUESTIONS AND PROBLEMS

1. What is an operating system?
2. What are the three types of operating systems? How do they differ from each other?
3. What is a time-sharing system? Be precise.
4. What are the main services provided by a typical contemporary operating system? What is the basic purpose of these services?
5. List one advantage and one disadvantage each for the CUI and the GUI.
6. What is the difference between a CUIs and GUIs? What is the most popular GUI for UNIX systems? Where was it developed?
7. What comprises the API and the AUI?
8. What is an operating system kernel? What are the primary tasks performed by the UNIX kernel?
9. What is a system call? What is the purpose of the system call interface?
10. If you access a UNIX system with the `ssh` command, write down the exact step-by-step procedure you go through to log on and log off. Include as many descriptive details as possible in this procedure so that if you forget how to log on, you can always refer back to this written procedure.
11. What is a shell? Name the most popular UNIX shells. Log on to your UNIX computer system and note the shell prompt being used.
12. How can you tell which variant from the main branches of UNIX (see Figure 1.3) is being used on the computer system that you log on to?
13. If you were designing a POSIX standard, what would you include in it? You might want to research the already adopted and proposed standards before answering this question.
14. If you were designing an SUS standard, what would you include in it? You might want to research the already adopted SUS standards, presented briefly in the chapter and online, before answering this question.
15. What system was the immediate predecessor of UNIX? Where was this predecessor and UNIX itself initially developed, and by whom?
16. Name the major versions and the three main branches of UNIX development. Which was the commercial branch? Which was the academic branch?

17. What three important characteristics of UNIX during its early development helped popularize it? Explain how these characteristics apply to you as a UNIX user, whatever your perspective.

18. Name the two most popular UNIX systems that are the basis of most UNIX systems. Where were they developed?

19. Trace the history of UNIX by browsing the Web. How many UNIX systems have been developed so far? How many non-UNIX systems have been developed? What is the most popular UNIX system for PCs? Why do you think it is so popular?

20. Name five popular members of the UNIX family. What is the name of your UNIX system?

21. In the late 1960s and early 1970s, the Digital Equipment Corporation (DEC) was a key player in the development of time-sharing systems. Browse the Web and find an article on RSTS, an operating system developed at DEC. What was its full name? What machines did it run on? What were its key features?

CHAPTER 2

A "Quick Start" into the UNIX Operating System

Objectives

- To introduce the UNIX character user interface (CUI) and show the generic structure of UNIX commands
- To describe how to connect and log on to a computer running the UNIX operating system, particularly PC-BSD and Solaris
- To explain how to manage and maintain files and directories
- To show where to get online help for UNIX commands
- To demonstrate the use of a beginner's set of utility commands
- To describe what a UNIX shell is
- To describe briefly some commonly used shells
- To cover the basic commands and operators

  `alias, biff, cal, cat, cd, chsh, cp, csh, echo, exit, hostname, login, logout, lp, lpr, ls, man, mesg, mkdir, more, mv, passwd, PATH, pg, pwd, rm, rmdir, set, ssh, su, sudo, talk, telnet, unalias, uname, whatis, whereis, who, whoami, write`

2.1 INTRODUCTION

To start working productively in UNIX, the beginner needs to know eight sequential topics, in the order presented as follows:

1. How to type a syntactically correct command on the UNIX command line. One of the most useful modes of interaction with the UNIX system uses text-based, typed commands.

2. How to log in and log out of a computer running UNIX, using one of the standard methods we show. UNIX allows users to enter the operating system autonomously, do a combination of text- and graphics-oriented operations, and exit gracefully.
3. How to maintain and organize files in the file structure. Creating a tree-like structure of folders (also called directories), and storing files in a logical fashion in these folders, is critical to working efficiently in UNIX.
4. How to get help on commands and their usage. In the command-based CUI environment, being able to find out, in a quick and easy way, how to use a command correctly is imperative to working efficiently.
5. How to execute a small set of essential utility commands to set up or customize your working environment. Once a beginner is familiar with the right way to construct file maintenance commands, adding a set of utility commands makes each session more productive.
6. The essential ways to work with UNIX shells, what they are, and how to find out what shell is running when you log in.
7. Ways to change your shell, and what shell environmental variables are.
8. What shell metacharacters are.

To use this chapter successfully as a springboard into the remainder of the book, you should read and follow the instructions, in the order presented. Each chapter builds on the information that precedes it, and will give you the concepts, command tools, and methods to program in the UNIX operating system. In this chapter, the major commands are defined with an abbreviated syntax description, which will clarify general components for the remainder of the textbook as follows:

SYNTAX

The exact syntax of how a command, its options, and its arguments are typed on the command line

Purpose: The specific purpose of the command
Output: A short description of the results of executing the command
Commonly used options/features: A listing of the most popular and useful options and option arguments

2.2 THE STRUCTURE OF A UNIX COMMAND

Because UNIX is reliant on both a graphical and a text-based CUI, correctly typed syntax is critical to ensure subsequent correct execution of commands.

After a user successfully logs on to a UNIX computer, a shell prompt, such as the `$` character, appears on the screen. The shell prompt is simply a message from the

computer system to say that it is ready to accept keystrokes on the command line that directly follows the prompt. The general syntax, or structure of a *single* command (as opposed to a command line that may have *multiple* commands typed on the same line, separated with input and output redirection characters) as it is typed on the command line is as follows:

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

where:

`$` is the command line or shell prompt from the computer;
anything enclosed in **`[ ]`** is not always needed;
`command` is the name of the valid UNIX command for that shell in lowercase letters;
`[-option(s)]` is one or more modifiers that change the behavior of command;
`[option argument(s)]` is one or more modifiers that change the behavior of **`[-option(s)]`**; and
`[command argument(s)]` is one or more objects that are affected by **`command`**.

Note the following seven essentials:

1. A space separates command, options, option arguments, and command arguments, but no space is necessary between multiple option(s) or multiple option arguments.
2. The order of multiple options or option arguments is irrelevant.
3. A space character is optional between the option and the option argument.
4. Always press the `<Enter>` key to submit the command for interpretation and execution.
5. Options may be preceded by a single hyphen `-` or two hyphens, `--`. No space character between hyphen(s) and option(s).
6. A small percentage of commands (like `whoami`) take no options, option arguments, or command arguments.
7. Everything on the command line is case sensitive!

The following are examples of commands typed on the UNIX command line after the `$` prompt, and illustrate some of the variations of the correct syntax for a single command that may have options and arguments:

```
$ ls
$ ls -la
$ ls -la m*
$ lpr -Pspr -n 3 proposal.txt
```

The first example contains only the command. The second contains the command `ls` and two options, `l` and `a`. The third contains the command `ls`, two options, `l` and `a`, and

a command argument, `m*`. The fourth contains the command `lpr`, two options, `P` and `n`, two option arguments, `spr` and `3`, and a command argument, `proposal.txt`.

You must also use the following rule of thumb: If the command executes properly, then you are returned to the shell prompt; if it does not execute properly, then you get an error message displayed on the command line, and then you are returned to the shell prompt. For example, if you type `xy` on the command line and then press `<Enter>`, usually you will get an error message saying that no such command can be found, and you are returned to the shell prompt so that you can keystroke a valid command.

This rule of thumb does not ensure that what you wanted to achieve by typing the syntactically correct command on the command line will be achieved. That is, you could execute a command and get no error messages. But the command may not have done the things you wanted it to do, simply because you used it with the wrong options or command arguments.

2.3 LOGGING ON AND LOGGING OFF

How can you log on to a UNIX computer and then gracefully leave?

Using one of these general ways, or a hybrid version of them:

- *Stand-alone*: Use a stand-alone system, such as the PC-BSD or Solaris systems we use throughout this book, where UNIX is the only operating system on the hardware.
- *Remote*: Connect to a remote computer running UNIX from a computer running UNIX or another operating system.
- *Virtual*: Start UNIX as a guest operating system in a virtual environment, such as VirtualBox or VMware, while another operating system is the host system.

These general ways are the first step a user takes in a typical UNIX session: gaining access to a UNIX system properly in an autonomous way.

A more detailed description of these ways follows:

1. *Stand-alone*: This way, the most common case and the methodology we deploy in the rest of this book with PC-BSD and Solaris, involves sitting at a computer that can function completely on its own. This does not mean that the stand-alone computer is not hooked up to a local area network (LAN), intranet, or the Internet.

   Rather, the users' connection to UNIX is dedicated to a single user at a time (or possibly many autonomous users that log on to the same system individually at different times) sitting at the computer and logging on to use UNIX on that hardware platform exclusively.

2. *Remote*: There are several variations of using this way. Here are just two possible scenarios:

   a. You sit at a computer that acts like the traditional *terminal* connected to a mainframe computer. This could also be a *thin client* (a minimally configured and

capable device) connected to a server. It is connected by a high-speed communications link to another single computer or multiple computers that are all interconnected with a LAN or the Internet. At the terminal, and the console or command window that appears on its screen, your interface with the operating system runs on a single, or even multiple, other computer(s). This is a shared resource method, where several users on many different terminals can share a single UNIX system.

b. You sit at a stand-alone computer, and via software such as PuTTY, Secure Shell (SSH), or SSH X Windows forwarding (a variant of TCP port forwarding), you connect to another system over a high-speed telecommunications link. The PuTTY or SSH software then becomes your *graphical connection*, allowing you to log on and use a remote computer or system that is running UNIX. This is usually a shared resource method, where several users on many different remote computers can share a single UNIX system.

3. *Virtual*: You have a UNIX or NIX-like operating system such as LINUX, OS X, or another operating system installed and running the computer you are sitting in front of, and you have installed a virtual environment such as VirtualBox or VMware on that computer. Then, when you want to use a UNIX system, you simply switch environments so that the UNIX system in the virtual environment is what you are using to interface with the computer hardware.

We do not cover virtual connections in this chapter, but in Chapter 25, "Virtualization Methodologies," we cover virtual environments such as VirtualBox.

In the following subsections, we present three practical, useful, easy, and popular ways of connecting and logging on and off a computer running the UNIX system, as outlined in Section 2.3.

The three ways we show are:

1. Stand-alone login and logout for PC-BSD and Solaris.
2. Remote login via the PuTTY program from a computer running Microsoft Windows to a UNIX computer running PC-BSD.
3. Remote login via an SSH client from a UNIX client computer running PC-BSD UNIX to another remote UNIX host computer.

What is common to all three of these ways is that your first task is to identify yourself correctly as a valid and autonomous user to the UNIX system. Doing so involves typing in a valid username, or login name, consisting of a string of valid characters. You then have to type in a valid password for that username.

Before proceeding with the remainder of this chapter, you should determine which one of the preceding three ways you will use to log in to a UNIX system, and then select from the three following sections that give details on how to use that way correctly. If you cannot

determine this on your own, get help from your instructor or the system administrator at your site. Be aware that you may have to use a hybrid way of logging in and out.

2.3.1 Stand-Alone Login Connection to PC-BSD and Solaris

The login and logout procedures shown in this section are standard and vary only slightly between all UNIX and UNIX-like systems. This way assumes that someone has either logged out gracefully or rebooted the computer before you got to it, but has not shutdown the system.

In this section, it is assumed that you are logging on to an already-running computer with PC-BSD UNIX or Solaris as the operating system. As previously stated, when you log in, identifying yourself to the UNIX system is your first task. Doing so involves typing in a valid username, or login name, consisting of a string of valid characters. Then you type in a valid password for that username.

Be aware that when typing on the command line, UNIX is case sensitive!

2.3.1.1 PC-BSD Login and Logout

The login window to PC-BSD is shown in Figure 2.1.

In the login window, you should keystroke your username in the username field, and then by default keep the desktop manager as KDE. If other window or desktop management systems were previously installed on the computer, you can choose one of those in the login window.

In the password field of the login window, type in your password. Finally, click on the right-facing arrow as seen in Figure 2.1, and you will be logged in. On PC-BSD, the KDE desktop management system appears on screen by default.

To gracefully terminate your connection with the computer running PC-BSD, make the Kickoff Application Launcher menu choices **Leave>Log Out**.

2.3.1.2 Solaris Login and Logout

The login windows to Solaris are shown in Figures 2.2 and 2.3.

Two sequential login windows appear, allowing you to type in your username in the first and password in the second. On Solaris, after you log in, the Gnome desktop management interface appears on screen by default.

To log out, make the pull-down **System>Logout** menu choice.

FIGURE 2.1 Stand-alone login window on PC-BSD.

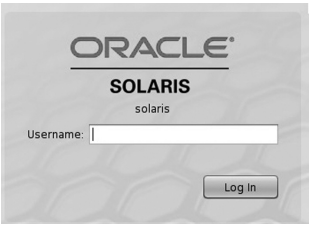

FIGURE 2.2 Stand-alone login window #1 on Solaris.

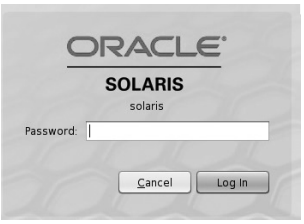

FIGURE 2.3 Stand-alone login window #2 on Solaris.

2.3.2 Connecting via PuTTY from a Microsoft Windows Computer

In this section, we make these basic assumptions:

1. That you are sitting at a computer running Microsoft Windows, and trying to connect and log on to a computer running the UNIX operating system.
2. On your Microsoft Windows computer, you are connected to the Internet, or an intranet where you know the Internet Protocol (IP) address of the UNIX computer you want to log on to.
3. You have downloaded and installed the PuTTY program on your Microsoft Windows computer or the system administrator has done so for you. The details of downloading this software and installing it are not given here. At the time of writing, the most current download site for the PuTTY program was:
   `http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`.
4. You are using PuTTY to make an SSH connection to a UNIX computer.
5. You know a valid username and password pair that will allow you to log in to the UNIX computer.

Once you execute the PuTTY program, you use the valid username/password pair, and then you can type commands into a console window or terminal screen. What you type in is shown as follows in **bold** text and is always followed by pressing the `<Enter>` key on the keyboard.

To begin, on the Microsoft Windows computer, double click on the PuTTY program icon, or from the **Start Menu>Programs** submenu, and choose PuTTY. When the PuTTY program first launches, the PuTTY configuration dialog window opens on screen, similar to Figure 2.4.

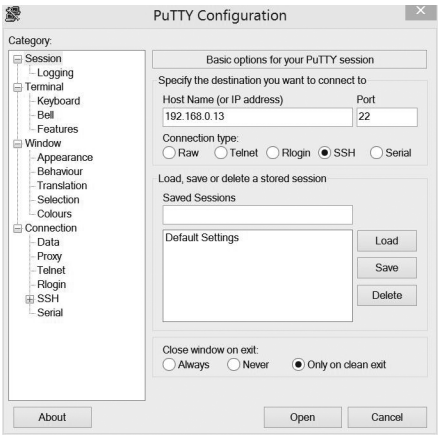

FIGURE 2.4 PuTTY configuration dialog window.

From the PuTTY configuration window, you can modify several of the parameters that control your interactive session with a UNIX system. Almost all of these parameters can be left at their defaults. The only two things that most users will need to do in this configuration window is type the host name (or IP address) of the UNIX computer they are trying to connect and log in to, and click the protocol button for SSH, as seen in Figure 2.4. The port number is automatically set at 22 if you click on the `SSH` button for "Connection type." You need to know what the host name or IP address of the UNIX computer you want to log on to is. Then click on the `Open` button and a console window will open on screen, as seen in Figure 2.5, thus allowing you to log in to the UNIX computer.

As previously stated, in the process of logging in, identifying yourself to the UNIX system is your first task. Doing so involves typing in a valid *username*, or *login name*, consisting of a string of valid characters. You then type a valid *password* for that username. There are both valid and invalid characters that you can use in both your username and password.

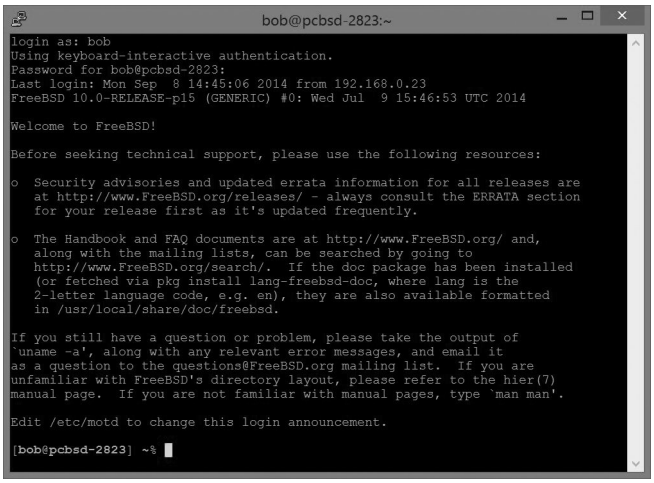

FIGURE 2.5 PuTTY login window.

See your system administrator or instructor to find out what these characters are on the UNIX system you want to log in to, if they have not already told you what they are.

As shown in Figure 2.5, in response to the `login:` prompt, you type in your username on the UNIX system, and then press `<Enter>` on the keyboard. In our case the username is **bob**, as seen in Figure 2.5. Remember that UNIX is case sensitive. When the `Password:` prompt appears, type your password on the UNIX system and then press `<Enter>` on the keyboard. Finally, the command line prompt appears on screen, as seen in Figure 2.5.

To terminate your connection type `logout` at the command line prompt and then press `<Enter>` on the keyboard or on a blank line press `<Ctrl+D>`—that is, hold down the `<Ctrl>` and `D` keys on the keyboard at the same time. Logging out is somewhat system dependent, as well as being an operation that can be tailored to a specific installation of UNIX by the local system administrator. In the C shell, the `logout` command is the default way of leaving the system gracefully.

If you use the Bourne shell or Korn shell, holding down `<Ctrl+D>` or typing `exit` will accomplish the same thing. You will then be logged off the system, the current PuTTY session will end, and all PuTTY windows will close.

If you started a new shell during your session and didn't exit that shell before logging off, UNIX will prompt `Not login shell`, and you will not be able to log off immediately. In this case, press `<Ctrl+D>` and the new shell will terminate. Also, if you started more than one shell and haven't exited from those shells before you log off, you will have to use `<Ctrl+D>` to terminate each shell individually. On some systems, you type `exit` on the command line to terminate a shell process. In either case, you will then be able to use the logout procedure previously described to leave the system.

2.3.3 Connecting via an SSH Client between UNIX Machines

This way allows a user on one UNIX computer to remote log in and log out of another UNIX computer using the SSH protocol. As detailed in Chapter 11, SSH is an encrypted channel of communication between computers on a LAN or on the Internet.

Before this way can be used, both systems must be able to talk to each other over the SSH channel, which we show how to do in Chapter 11. Also, as previously stated, the user must know a valid username/password pair to be able to log in to the remote system!

We show three possible ways this can happen. First, if the user has already logged into the host successfully from the client before and the authentication keys have not changed. Second, if the user has never logged into the host successfully before from the client. And third, if the user has logged into the host before but the authentication key on the host has changed since the last successful login. These are practical situations one might encounter any time you use this remote login method.

What the user types in is shown in **`bold`** text:

1. Having logged in before successfully:

```
[bob@pcbsd-923] ~% ssh 192.168.0.8
Password for bob@pcbsd-2467: XXX
```

```
Last login: Mon Sep 21 17:20:51 2015 from 192.168.0.13
FreeBSD 10.2-RELEASE-p4 (GENERIC) #0: Tue Aug 18 15:15:36 UTC
    2015
Output truncated...
[bob@pcbsd-2467] ~% Execute command line UNIX commands…
[bob@pcbsd-2467] ~% logout
Connection to 192.168.0.8 closed.
[bob@pcbsd-923] ~%
```

2. Having never logged in before:

```
[bob@pcbsd-923] ~% ssh 192.168.0.6
The authenticity of host '192.168.0.6 (192.168.0.6)' can't be
    established.
RSA key fingerprint is 47:62:a2:9b:24:9e:5e:51:49:3b:80:aa:91:
    a3:fd:de.
No matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.6' (RSA) to the list of
    known hosts.
Password: XXX
Last login: Mon Sep 21 17:06:59 2015 from 192.168.0.13
Oracle Corporation SunOS 5.11 11.2 June 2014
bob@solaris:~$ Execute command line UNIX commands…
bob@solaris:~$ logout
Connection to 192.168.0.6 closed.
[bob@pcbsd-923] ~%
```

3. Logged in before but host key has changed:

```
[bob@pcbsd-923] ~% ssh 192.168.0.8
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED! @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-
    middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
43:e8:cf:33:d5:ed:dd:05:d9:e9:a5:9d:d3:18:1d:2b.
Please contact your system administrator.
Add correct host key in /usr/home/bob/.ssh/known_hosts to get
    rid of this message.
Offending ECDSA key in /usr/home/bob/.ssh/known_hosts:2
ECDSA host key for 192.168.0.8 has changed and you have
    requested strict checking.
Host key verification failed.
[bob@pcbsd-923] ~% cd /usr/home/bob/.ssh
```

```
[bob@pcbsd-923] ~/.ssh% rm known_hosts
[bob@pcbsd-923] ~/.ssh% cd
[bob@pcbsd-923] ~% ssh 192.168.0.8
The authenticity of host '192.168.0.8 (192.168.0.8)' can't be
    established.
ECDSA key fingerprint is 43:e8:cf:33:d5:ed:dd:05:d9:e9:a5:9d:d
    3:18:1d:2b.
No matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.8' (ECDSA) to the list
    of known hosts.
Password for bob@pcbsd-2467: XXX
Last login: Sat Sep 19 11:24:47 2015 from 192.168.0.13
FreeBSD 10.2-RELEASE-p4 (GENERIC) #0: Tue Aug 18 15:15:36 UTC
    2015
Output truncated...
[bob@pcbsd-2467] ~% Execute command line UNIX commands...
[bob@pcbsd-2467] ~% logout
Connection to 192.168.0.8 closed.
[bob@pcbsd-923] ~%
```

In all of these scenarios, the user is assumed to have an account with the same username, and possibly password, on both client and host systems.

In 2., the keys are generated on host and client after you type in `yes` and press `<Enter>`.

In 3., after the first failed attempt to establish an SSH connection, the error message indicates that the authentication key has changed on the host. Therefore, a removal of the offending key in the file **/usr/home/bob/.ssh/known_hosts** on the client machine **[bob@pcbsd-923]** is done by deleting that file. Then a new key is generated, an exchange can take place, and the login can proceed.

The line in these sessions that reads *Execute command line UNIX commands…* is where the user types in any of the valid UNIX commands we show in this chapter and throughout the rest of this book. Finally, after typing `logout`, the user cuts the SSH channel connection, and is returned to the command line prompt of the local client system.

2.4 FILE MAINTENANCE COMMANDS AND HELP ON UNIX COMMAND USAGE

After your first-time login to a new UNIX system using one of the three ways we described, your first action is to construct and organize your workspace and the files contained in it. The operation of organizing your files according to some logical scheme is known as *file maintenance*. A logical scheme used to organize your files might consist of creating *bins* for storing files according the subject matter of the contents of the files, or according to the dates of their creation. In the following sections, you will type file creation and maintenance commands that produce a structure as shown in Figure 2.6. Complete the operations shown in the following sections in the order they are presented, to get a better overview of what file maintenance really is. Also, it is critical that you review what was

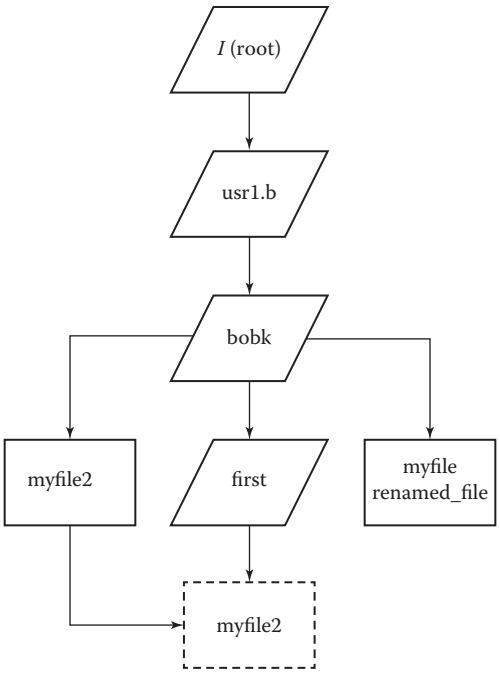

FIGURE 2.6 Example of a file and directory structure.

presented in Section 2.2 regarding the structure of a UNIX command, so that when you begin to type commands for file maintenance, you understand how the syntax of what you are typing conforms to the general syntax of any UNIX command.

2.4.1 File and Directory Structure

When you first log in, you are working in the *home directory*, or folder, of the autonomous user associated with the username and password you used to log in. Whatever directory you are presently in is known as the *current working directory*, and there is only one current working directory active at any given time. It is helpful to visualize the structure of your files and directories using a diagram. Figure 2.6 is an example of a home directory and file structure for a user named **bobk**. In this figure, directories are represented as parallelograms and plain files (e.g., files that contain text or binary instructions) are represented as rectangles. A *pathname*, or path, is simply a textual way of designating the location of a directory or file in the complete file structure of the UNIX system you are working on. For example, the path to the file **myfile2** in Figure 2.6 is **/usr1.b/bobk/myfile2**. The designation of the path begins at the root (/) of the entire file system, descends to the folder named **usr1.b**, and then descends again to the home directory named **bobk**.

As shown in Figure 2.6, the files named **myfile**, **myfile2**, and **renamed_file** are stored under or in the directory **bobk**. Beneath **bobk** is a *subdirectory* named **first**. In the next sections, you will create these files, and the subdirectory structure, in the home directory of the username that you have logged into your UNIX system.

2.4.2 Viewing the Contents of Files

To begin working with files, you create a new file by using the `cat` command. The syntax of the `cat` command is as follows:

SYNTAX

```
cat [options] [file-list]
```

Purpose: Join one or more files sequentially or display them in the console window
Output: Contents of the files in **file-list** displayed on the screen, one file at a time
Commonly used options/features:

+E	Display $ at the end of each line
-n	Put line numbers on the displayed lines
-- help	Display the purpose of the command and a brief explanation of each option

The `cat` command, short for concatenate, allows you to join files. In the example you will join what you type on the keyboard to a new file being created in the current working directory. This is achieved by the redirect character >, which takes what you type at the standard input (in this case the keyboard) and directs it into the file named **myfile**. As stated in Section 2.2, this usage involves the command `cat` but no options, option arguments, or command arguments. It simply uses the command, a redirect character, and a target, or destination, named **myfile**, where the redirection will go.

This is the very simplest example of a *multiple command* typed on the command line, as opposed to a single command, as shown in Section 2.2. In a multiple command, you can string together single UNIX commands in a chain with connecting operators, such as the redirect character shown here.

```
$ cat > myfile
This is an example of how to use the cat command to add plain text
to a file
<Ctrl+D>
$
```

You can type as many lines of text, pressing <Enter> on the keyboard to distinguish between lines in the file, as you want. Then, on a new line, when you hold down <Ctrl+D>, the file is created in the current working directory, using the command you typed. You can view the contents of this file, since it is a plain text file that was created using the keyboard, by doing the following:

This is a simple example of a single UNIX command.

```
$ more myfile
This is an example of how to use the cat command to add plain text
to a file
$
```

The general syntax of the more command is as follows:

SYNTAX

```
more [options] [file-list]
```

Purpose: Concatenate/display the files in **file-list** on the screen, one screen at a time
Output: Contents of the files in **file-list** displayed on the screen, one page at a time
Commonly used options/features:

+E/str	Start two lines before the first line containing `str`
-nN	Display N lines per screen/page
+N	Start displaying the contents of the file at line number `N`

The `more` command shows one screen of a file at a time. If the file is several pages long, you can proceed to view subsequent pages by pressing the `<Space>` key on the keyboard, or by pressing the `Q` key to quit. Solaris has a command named `pg` that accomplishes the same thing as the `more` command.

2.4.3 Creating, Deleting, and Managing Files

To copy the contents of one file into another file, use the `cp` command. The general syntax of the `cp` command is as follows:

SYNTAX

```
cp [options] file1 file2
```

Purpose: Copy **file1** to **file2**; if **file2** is a directory, make a copy of **file1** in this directory
Output: Copied files
Commonly used options/features:

-i	If destination exists, prompt before overwriting
-p	Preserve file access modes and modification times on copied files
-r	Recursively copy files and subdirectories

For example, to make an exact duplicate of the file named **myfile**, with the new name **myfile2**, type the following:

```
$ cp myfile myfile2
$
```

This usage of the `cp` command has two required command arguments. The first argument is the source file that already exists and which you want to copy. The second argument is the destination file or the name of the file that will be the copy. Be aware that many UNIX commands can take plain, ordinary, or regular files as arguments, or can take directory files as arguments. This can change the basic task accomplished by the command. It is also worth noting that not only can file names be arguments, but pathnames as well. This changes the site or location, in the path structure of the file system, of operation of the command.

In order to change the name of a file or directory, you can use the `mv` command. The general syntax of the `mv` command is as follows:

SYNTAX

```
mv [options] file1 file2
mv [options] file-list directory
```

Purpose: First syntax: Rename file1 to file2
Second syntax: Move all the files in file-list to directory
Output: Renamed or relocated files
Commonly used options/features:
-f Force the move regardless of the file access modes of the destination file
-i Prompt the user before overwriting the destination

In the following usage, the first argument to the `mv` command is the source file name, and the second argument is the destination name.

```
$ mv myfile2 renamed_file
$
```

It is important at this point to notice the use of spaces in UNIX commands. What if you obtain a file from a Windows 10 system that has one or more spaces in one of the file names? How can you work with this file in UNIX? The answer is simple. Whenever you need to use that file name in a command as an argument, enclose the file name in double quotes ("). For example, you might obtain a file that you have detached from an e-mail message from someone on a Windows 10 system, such as **latest revisions october.txt**.

In order to work with this file on a UNIX system—that is, to use the file name as an argument in a UNIX command—enclose the whole name in double quotes. The correct command to rename that file to something shorter would be:

```
$ mv "latest revisions october.txt" laterevs.txt
$
```

In order to delete a file, you can use the `rm` command. The general syntax of the `rm` command is as follows:

SYNTAX

```
rm [options] file-list
```

Purpose: Removes files in **file-list** from the file structure (and disk)
Output: Deleted files
Commonly used options/features:
-f Remove regardless of the file access modes of **file-list**
-i Prompt the user before removing files in **file-list**
-r Recursively remove the files in **file-list** if **file-list** is a directory; use with caution!

To delete the file **renamed_file** from the current working directory, type:

```
$ rm renamed_file
$
```

The most important command you will execute to do file maintenance is the `ls` command. The general syntax for the `ls` command is as follows:

SYNTAX

```
ls [options] [pathname-list]
```

Purpose: Sends the names of the files and directories in the directory specified by **pathname-list** to the display screen
Output: Names of the files and directories in the directory specified by **pathname-list**, or the names only if **pathname-list** contains file names only
Commonly used options/features:
- **-F** Display a slash character (/) after directory names, an asterisk (*) after binary executables, and an "at" character (@) after symbolic links
- **-a** Display names of all the files, including hidden files
- **-i** Display inode numbers
- **-l** Display long list that includes file access modes, link count, owner, group, file size (in bytes), and modification time

The `ls` command will list the names of files or folders in your current working directory or folder. In addition, as with the other commands we have used so far, if you include a complete pathname specification for the `pathname-list` argument to the command, then you can list the names of files and folders along that pathname list. To see the names of the files now in your current working directory, type the following:

```
$ ls
Desktop
Mail
XF86Config.new
kdeinit.core
order.asp.html
order.asp_files
myfile
myfile2$
```

Please note that you will probably not get a listing of the same file names as we did here, because your system will have placed some files automatically in your home directory, as in the example we used, aside from the ones we created together named **myfile** and **myfile2**. Also note that this file name listing does not include the name **renamed_file**, because we deleted that file.

The next command you will execute is actually just an alternate or modified way of executing the `ls` command, one that includes the command name and options. As shown in

Section 2.2, a UNIX command has options that can be typed on the command line along with the command to change the behavior of the basic command. In the case of the `ls` command, the options `l` and `a` produce a longer listing of all ordinary and system (dot) files, as well as providing other attendant information about the files. Don't forget to put the space character between the `s` and the dash. Remember again from Section 2.2 that spaces delimit, or partition, the components of a UNIX command as it is typed on the command line.

Now, type the following command:

```
$ ls -la
drwxr-xr-x  10 bobk  wheel     1024 Oct 11 13:42 .
drwxr-xr-x  17 bobk  wheel      512 Sep 20 16:30 ..
lrwxr-xr-x   1 bobk  wheel       32 Oct 11 13:13
-rw-------   1 bobk  wheel      197 Oct 11 13:13 .ICEauthority
-rw-------   1 bobk  wheel      105 Oct 11 13:13 .Xauthority
-rw-r--r--   2 bobk  wheel      797 Jan 16  2004 .cshrc
-rw-r--r--   2 bobk  wheel      251 Jan 16  2004 .profile
drwxr-xr-x   2 bobk  wheel      512 Apr 15 15:11 .qt
-rwxr-xr-x   1 bobk  wheel       14 Apr 15 08:06 .xinitrc
-rwxr-xr-x   1 bobk  wheel       14 Apr 15 08:06 .xsession
drwx------   3 bobk  wheel      512 Sep 20 16:29 Desktop
drwx------   7 bobk  wheel      512 Apr 15 16:40 Mail
-rw-r--r--   1 bobk  wheel     2798 Apr 17 16:07 XF86Config.new
-rw-------   1 bobk  wheel  7360512 Sep 20 16:29 kdeinit.core
-rw-r--r--   1 bobk  wheel    35394 Apr 17 15:23 order.asp.html
drwxr-xr-x   2 bobk  wheel     1024 Apr 17 15:23 order.asp_files
-rw-r--r--   2 bobk  wheel      797 Jan 16  2004 myfile
-rw-r--r--   2 bobk  wheel      797 Jan 16  2004 myfile2
$
```

As you see in this screen display (which shows the listing of files in our home directory and will not be the same as the listing of files in your home directory), the information about each file in the current working directory is displayed in eight columns. The first column shows the type of file, where `d` stands for directory, `l` stands for symbolic link, and – stands for ordinary or regular file. Also in the first column, the access modes to that file for user, group, and others is shown as `r`, `w`, or `x`. In the second column, the number of links to that file is displayed. In the third column, the username of the owner of that file is displayed. In the fourth column, the name of the group for that file is displayed. In the fifth column, the number of bytes that the file occupies on disk is displayed. In the sixth column, the date that the file was last modified is displayed. In the seventh column, the time that the file was last modified is displayed. In the eighth and final column, the name of the file is displayed. This way of executing the command is a good way to list more complete information about the file. Examples of using the more complete information are (1) so that you can know the byte size and be able to fit the file on some portable storage medium, or (2) to display the access modes, so that you can alter the access modes to a particular file or directory.

You can also get a file listing for a single file in the current working directory by using another variation of the `ls` command, as follows:

```
$ ls -la myfile
-rw-r--r--  1 bobk  wheel  797 Jan 16 2015 myfile
$
```

This variation shows you a long listing with attendant information for the specific file named **myfile**. A breakdown of what you typed on the command line is 1) **ls**, the command name, 2) **-la** , the options, and 3) **myfile**, the command argument.

What if you make a mistake in your typing and misspell a command name or one of the other parts of a command? Type the following on the command line:

```
$ lx -la myfile
lx: not found
$
```

The `lx: not found` reply from UNIX is an error message. There is no `lx` command in the UNIX operating system, so an error message is displayed. If you had typed an option that did not exist, you would also get an error message. If you supplied a file name that was not in the current working directory, you would get an error message, too. This makes an important point about the execution of UNIX commands. If no error message is displayed, then the command executed correctly and the results might or might not appear on screen, depending on what the command actually does. If you get an error message displayed, you must correct the error before UNIX will execute the command as you type it. Typographic mistakes account for about 98% of the errors that beginners make.

2.4.4 Creating, Deleting, and Managing Directories

Another critical aspect of file maintenance is the set of procedures and the related UNIX commands you use to create, delete, and organize directories in your UNIX account on a computer. When moving through the file system, you are either ascending or descending to reach the directory you want to use. The directory directly above the current working directory is referred to as the *parent* of the current working directory. The directory or directories immediately under the current working directory are referred to as the *children* of the current working directory. For more information on file system structure, see Chapter 4. The most common mistake for beginners is misplacing files. They cannot find the file names listed with the `ls` command because they have placed or created the files in a directory either above or below the current working directory in the file structure. When you create a file, if you have also created a logically organized set of directories beneath your own home directory, you will know where to store the file. In the following set of commands, we create a directory beneath the home directory and use that new directory to store a file.

To create a new directory beneath the current working directory, you use the `mkdir` command. The general syntax for the `mkdir` command is as follows:

SYNTAX

```
mkdir [options] dirnames
```

Purpose: Creates directory or directories specified in **dirnames**
Output: New directory or directories
Commonly used options/features:
`-m MODE` Create a directory with given access modes
`-p` Create parent directories that don't exist in the pathnames specified in **dirnames**

To create a child, or subdirectory, named **first** under the current working directory, type the following:

```
$ mkdir first
$
```

This command has now created a new subdirectory named **first** under, or as a child of, the current working directory. Refer back to Figure 2.6 for a graphical description of the directory location of this new subdirectory.

In order to change the current working directory to this new subdirectory, you use the `cd` command. The general syntax for the `cd` command is as follows:

SYNTAX

```
cd [directory]
```

Purpose: Change the current working directory to **directory** or return to the home directory when **directory** is omitted
Output: New current working directory

To change the current working directory to **first** by descending down the path structure to the specified directory named **first**, type the following:

```
$ cd first
$
```

You can always verify what the current working directory is by using the `pwd` command. The general syntax of the `pwd` command is as follows:

SYNTAX

```
pwd
```

Purpose: Displays the current working directory on screen
Output: Pathname of current working directory

You can verify that **first** is now the current working directory by typing the following:

```
$ pwd
/usr1.b/bobk/first
$
```

The output from UNIX on the command line shows the pathname to the current working directory or folder. As previously stated, this path is a textual route through the complete file structure of the computer that UNIX is running on, ending in the current working directory. In the this example of the output, the path starts at **/**, the root of the file system. Then it descends to the directory **usr1.b**, a major branch of the file system on the computer running UNIX. Then it descends to the directory **bobk**, another branch, which is the home directory name for the user. Finally, it descends to the branch named **first**, the current working directory.

On some systems, depending on the default settings, another way of determining what the current working directory is can be done by simply looking at the command line prompt. This prompt may be prefaced with the complete path to the current working directory, ending in the current working directory.

You can ascend back up to the home directory, or the parent of the subdirectory **first**, by typing the following:

```
$ cd
$
```

An alternate way of doing this is to type the following, where the tilde character (~) resolves to, or is a substitute for, the specification of the complete path to the home directory:

```
$ cd ~
$
```

To verify that you have now ascended up to the home directory, type the following:

```
$ pwd
/usr1.b/bobk
$
```

You can also ascend to a directory above your home directory, sometimes called the parent of your current working directory, by typing the following:

```
$ cd ..
$
```

In this command, the two periods (..), represent the parent, or branch above the current working directory. Don't forget to type a space character between the `d` and the first

period. To verify that you have ascended to the parent of your home directory, type the following:

```
$ pwd
/usr1.b
$
```

To descend to your home directory, type the following:

```
$ cd
$
```

To verify that there are two files in the home directory that begins with the letters `my`, type the following command:

```
$ ls my*
myfile  myfile2
$
```

The asterisk following the `y` on the command line is known as a *metacharacter*, or a character that represents a pattern; in this case, the pattern is any set of characters. When UNIX interprets the command after you press the `<Enter>` key on the keyboard, it searches for all files in the current working directory that begin with the letters `my` and end in anything else.

Another aspect of organizing your directories is movement of files between directories, or changing the location of files in your directories. For example, you now have the file **myfile2** in your home directory, but you would like to move it into the subdirectory named **first**. See Figure 2.6 for a graphic description to change the organization of your files at this point. To accomplish this, you can use the second syntax method illustrated for the `mv file-list directory` command to move the file **myfile2** down into the subdirectory named **first**. To achieve this, type the following:

```
$ mv myfile2 first
$
```

To verify that **myfile2** is indeed in the subdirectory named first, type the following:

```
$ cd first
$ ls
myfile2
$
```

You will now ascend to the home directory, and attempt to remove or delete a file with the `rm` command. Caution: you should be very careful when using this command, because

once a file has been deleted, the only way to recover it is from archival backups that you or the system administrator have made of the file system.

```
$ cd
$ rm myfile2
rm: myfile2: No such file or directory
$
```

You get the error message because in the home directory, the file named **myfile2** does not exist. It was moved down into the subdirectory named first.

Directory organization also includes the ability to delete empty or nonempty directories. The command that accomplishes the removal of empty directories is `rmdir`. The general syntax of the `rmdir` command is as follows:

SYNTAX

`rmdir [options] dirnames`

Purpose: Removes the empty directories specified in **dirnames**
Output: Removes directories
Commonly used options/features:
`-p` Remove empty parent directories as well
`-r` Recursively delete files and subdirectories beneath the current directory

To delete an entire directory below the current working directory, type the following:

```
$ rmdir first
rmdir: first: Directory not empty
$
```

Since the file **myfile2** is still in the subdirectory named **first**, **first** is not an empty directory, and you get the error message that the `rmdir` command will not delete the directory. If the directory was empty, `rmdir` would have accomplished the deletion. One way to delete a nonempty directory is by using the `rm` command with the `-r` option. The `-r` option recursively descends down into the subdirectory and deletes any files in it before actually deleting the directory itself. Be cautious with this command, since you may inadvertently delete directories and files with it. To see how this command deletes a nonempty directory, type the following:

```
$ rm -r first
$
```

The directory **first** and the file **myfile2** are now removed from the file structure.

2.4.5 Obtaining Help with the Man Command

A very convenient utility available on UNIX systems is the online help feature, achieved via the use of the `man` command. The general syntax of the `man` command is as follows:

SYNTAX

```
man [options][-s section] command-list
man -k keyword-list
```

Purpose: First syntax: Display UNIX Reference Manual pages for commands in **command-list** one screen at a time
Second syntax: Display summaries of commands related to keywords in **keyword-list**
Output: Manual pages one screen at a time
Commonly used options/features:

-k keyword-list	Search for summaries of keywords in **keyword-list** in a database and display them
-s sec-num	Search section number **sec-num** for manual pages and display them

To get help by using the `man` command, on usage and options of the `ls` command, for example, type the following (shown for PC-BSD):

```
$ man ls

LS(1))                    FreeBSD General Commands Manual LS(1)
NAME
    ls - list directory contents
SYNOPSIS
    ls [-ABCFGHLPRTWZabcdfghiklmnopqrstuwx1] [file ...]
DESCRIPTION
    For each operand that names a file of a type other than
directory, ls displays its name as well as any requested,
associated information. For each operand that names a file of type
directory, ls displays the names of files contained within that
directory, as well as any requested, associated information.
If no operands are given, the contents of the current directory
are displayed. If more than one operand is given, non-directory
operands are displayed first; directory and non-directory operands
are sorted separately and in lexicographical order.
    The following options are available:
Press <SPACE> to continue, or q to quit q
$
```

This output from UNIX is a UNIX *manual page*, or *man page*, which gives a synopsis of the command usage showing the options, and a brief description that helps you understand how the command should be used. Typing `q` after one page has been displayed, as seen in the example, returns you to the command line prompt. Pressing the space key on the keyboard would have shown you more of the content of the manual pages, one screen at a time, related to the `ls` command.

To get help in using all the UNIX commands and their options, use the `man man` command to go to the UNIX reference manual pages.

TABLE 2.1 Sections of the UNIX Manual

Section	What It Describes
1	User commands
2	System calls
3	Language library calls (C, FORTRAN, etc.)
4	Devices and network interfaces
5	File formats
6	Games and demonstrations
7	Environments, tables, and macros for troff
8	System maintenance–related commands

The pages themselves are organized into eight sections, depending on the topic described and the topics that are applicable to the particular system. Table 2.1 lists the sections of the manual and what they contain. Most users find the pages they need in Section 2.1. Software developers mostly use library and system calls and thus find the pages they need in Sections 2.2 and 2.3. Users who work on document preparation get the most help from Section 2.7. Administrators mostly need to refer to pages in Sections 2.1, 2.4, 2.5, and 2.8.

The manual pages comprise multipage, specially formatted, descriptive documentation for every command, system call, and library call in UNIX. This format consists of seven general parts: name, synopsis, description, list of files, related information, errors, warnings, and known bugs. You can use the `man` command to view the manual page for a command. Because of the name of this command, the manual pages are normally referred to as UNIX man pages. When you display a manual page on the screen, the top-left corner of the page has the command name with the section it belongs to in parentheses, as with `LS(1)`, seen at the top of the output manual page.

The command used to display the manual page for the `passwd` command is:

```
$ man passwd
```

The manual page for the `passwd` command now appears on the screen, but we do not show its output. Because they are multipage text documents, the manual pages for each topic take up more than one screen of text to display their entire contents. To see one screen of the manual page at a time, press the space bar on the keyboard. To quit viewing the manual page, press the `Q` key on the keyboard.

There is no `-k` option listed on a PC-BSD system, but there is one on a Solaris system. And the `-k` option works on both systems!

Now type this command:

```
$ man pwd
```

If more than one section of the man pages has information on the same word and you are interested in the man page for a particular section, you can use the `-S` option (in Solaris

it is lowercase s). The following command line therefore displays the man page for the read system call and not the man page for the shell command read.

```
$ man -S2 read
```

The command `man -S3 fopen fread strcmp` sequentially displays man pages for three C library calls: **fopen**, **fread**, and **strcmp**.

On a Solaris system, using the `man` command includes typing the command with the `-k` option, thereby specifying a keyword that limits the search. The search then yields useful man page headers from all the man pages on the system that contain just the keyword reference. For example, the following session yields the on-screen output on a Solaris system:

```
% man -k passwd
1. passwd(4) /usr/share/man/man4/passwd.4
passwd - password file
2. passwd(1openssl) /usr/share/man/man1openssl/passwd.1openssl
passwd - compute password hashes
3. passwd(1) /usr/share/man/man1/passwd.1
passwd - change login password and password attributes
4. slapd-passwd(5oldap) /usr/share/man/man5oldap/
slapd-passwd.5oldap
slapd-passwd - /etc/passwd backend to slapd
5. getpw(3c) /usr/share/man/man3c/getpw.3c
getpw - get passwd entry from UID
6. vino-passwd(1) /usr/share/man/man1/vino-passwd.1
vino-passwd - change vino login password
7. pwconv(1m) /usr/share/man/man1m/pwconv.1m
pwconv - installs and updates /etc/shadow with information from /
etc/passwd
8. SSL_CTX_set_default_passwd_cb(3openssl) /usr/share/man/
man3openssl/SSL_CTX_set_default_passwd_cb.3openssl
SSL_CTX_set_default_passwd_cb,
SSL_CTX_set_default_passwd_cb_userdata
- set passwd callback for encrypted PEM file handling
9. SSL_CTX_set_default_passwd_cb_userdata(3openssl) /usr/share/
man/man3openssl/SSL_CTX_set_default_passwd_cb_userdata.3openssl
SSL_CTX_set_default_passwd_cb,
SSL_CTX_set_default_passwd_cb_userdata
- set passwd callback for encrypted PEM file handling
```

2.4.6 Other Methods of Obtaining Help

To get a short description of what any particular UNIX command does, you can use the `whatis` command. This is similar to the command `man -f`. The general syntax of the `whatis` command is as follows:

SYNTAX

whatis keywords

Purpose: Search the whatis database for abbreviated descriptions of each keyword
Output: Prints a one-line description of each keyword to the screen

The following is an illustration of how to use `whatis`.
The output of the two commands are truncated.

```
$ whatis man
...
man(1)       -format and display the online manual pages
…
$
```

You can also obtain short descriptions of more than one command by entering multiple arguments to the `whatis` command on the same command line, with spaces between each argument. The following is an illustration of this method:

```
$ whatis login set setenv
...
login(1)     -sign on
...
set(1)       -set runtime parameters for session
...
setenv(1)    -change or add an environment variable
...
$
```

The following in-chapter exercises ask you to use the `man` and `whatis` commands to find information about the `passwd` command. After completing the exercises, you can use what you have learned to change your login password on the UNIX system that you use.

EXERCISE 2.1

Use the `man` command with the `-k` option (in both PC-BSD and Solaris) to display abbreviated help on the `passwd` command. Doing so will give you a screen display similar to that obtained with the `whatis` command, but it will show all apropos command names that contain the characters `passwd`.

EXERCISE 2.2

Use the `whatis` command (in both PC-BSD and Solaris) to get a brief description of the `passwd` command shown in Exercise 2.1, and then note the difference between the commands `whatis passwd` and `man -k passwd`.

2.5 UTILITY COMMANDS

There are several major commands that allow the beginner to be more productive when using the UNIX system. A sampling of these kinds of utility commands is given in the following sections, and is organized as system setups, general utilities, and communications commands.

2.5.1 Examining System Setups

The `whereis` command allows you to search along certain prescribed paths to locate utility programs and commands, such as shell programs. The general syntax of the `whereis` command is as follows:

SYNTAX

```
whereis [options] filename
```

Purpose: Locate the binary, source, and man page files for a command
Output: The supplied names are first stripped of leading pathname components and extensions, then pathnames are displayed on screen
Commonly used options/features:
-b Search only for binaries
-s Search only for source code

For example, if you type the command `whereis csh` on the command line, you will see a list of the paths to the C shell program files themselves. Note that the paths to a built-in, or internal, command cannot be found with the `whereis` command. We provide more information about internal and external shell commands in Chapter 10.

When you first log on, it is useful to be able to view a display of information about your `userid`, the computer or system you have logged on to, and the operating system on that computer. These tasks can be accomplished with the `whoami` command, which displays your `userid` on the screen. The general syntax of the `whoami` command is as follows:

SYNTAX

```
whoami
```

Purpose: Displays the effective user id
Output: Displays your effective user id as a name on standard

The following shows how our system responded to this command when we typed it on the command line.

```
$ whoami
bobk
$
```

The following in-chapter exercises give you the chance to use `whereis`, `whoami`, and two other important utility commands, `who` and `hostname` to obtain important information about your system.

EXERCISE 2.3

On a PC-BSD system, use the `whereis` command to locate binary files for the Korn shell, the Bourne shell, the Bourne Again shell, the C shell, and the Z shell. Are any of these shell programs not available on your system?

EXERCISE 2.4

Use the `whoami` command to find your username on the system that you're using. Then use the `who` command to see how your username is listed, along with other users of the same system. What is the on-screen format of each user's listing that you obtained with the `who` command? Try to identify the information in each field on the same line as your username.

EXERCISE 2.5

Use the `hostname` command to find out what host computer you are logged on to. Can you determine from this list whether you are using a stand-alone computer or a networked computer system? Explain how you can know the difference from the list that the `hostname` command gives you.

2.5.2 Printing and General Utility Commands

2.5.2.1 For PC-BSD

A very useful and common task performed by every user of a computer system is the printing of text files at a printer. The command to perform printing on a PC-BSD system is `lpr`. The general syntax of the `lpr` command is as follows:

SYNTAX

```
lpr [options] filename
```

Purpose: Send files to the printer
Output: Files sent to the printer queue as print jobs
Commonly used options/features:
`-P printer` Send output to the named printer
`-# copies` Produce the number of copies indicated for each named file

The following `lpr` command, when using PC-BSD, accomplishes the printing of the file named **order.eps** at the printer designated on our system as **spr**. Remember from Section 2.2 that no space is necessary between the option (in this case `-P`) and the option argument (in this case `spr`).

```
$ lpr -Pspr order.eps
$
```

The following `lpr` command, when using PC-BSD, accomplishes the printing of the file named **memo1** at the default printer.

```
$ lpr memo1
$
```

The following multiple command combines the `man` command and the `lpr` command, and ties them together with the UNIX *pipe* (|) redirection character, to print the man pages describing the `ls` command at the printer named `hp1`. This will work when using PC-BSD.

```
$ man ls | lpr -Php1
$
```

2.5.2.2 For Solaris

The following shows how to perform printing tasks on Solaris using the `lp` command.

The general syntax of the `lp` command for Solaris is as follows:

SYNTAX

```
lp [options][option arguments] file(s)
```

Purpose: Submit files for printing on a designated system printer, or alter pending print jobs
Output: Printed files or altered print queue
Commonly used options/features:
-d destination Print to the specified destination
-P pagelist Print selected pages as specified in **pagelist**

In the first command, the file to be printed is named **file1**. In the second command, the files to be printed are named **sample** and **phones**. Note that the `-d` option is used to specify which printer to use. The option to specify the number of copies is `-n` for the `lp` command.

```
$ lp -d spr file1
request id is spr-983 (1 file(s))
$ lp -d spr -n 3 sample phones
request id is spr-984 (2 file(s))
$
```

Among the most useful of the general purpose, personal productivity utility commands, the `cal` command displays a calendar for a year or a month. The general syntax of the `cal` command is as follows:

SYNTAX

```
cal [[month]year]
```

Purpose: Displays calendar on screen as text
Output: Displays a calendar of the month or year

The optional parameter `month` can be between 1 and 12, and `year` can be 0–9999. Just like the UNIX system, the `cal` command is Y2K compliant. If no argument is specified, the command displays the calendar for the current month of the current year. If only one parameter is specified, it is taken as the year. Thus the `cal 3 2005` command displays the calendar for March 2005. The command `cal 1969` displays the calendar for the year 1969, the year the UNIX operating system was born.

2.5.3 Communications Commands

The `write` command is used to send a message to another user who is currently logged on to the system. The syntax and a brief description of the command is as follows:

SYNTAX

`write username [terminal]`

Purpose: Write on the terminal screen or console window of the user with login name username; the user must be logged on to the system, and the user's terminal must have write access privilege given by the `mesg` command.
Output: Message on another user's console window.

The example shown in the following command line dialog session illustrates the use of this command. The prerequisite for executing the `write` command is execution of the `mesg y` command by both sender (in anticipation of a reply) and receiver to allow writing to their respective terminal screens or console windows. The `who` command is used to determine whether the person to whom you want to write is logged on. In this case, both sender (**sarwar**) and receiver (**bobk**) are logged on to the computer **upibm7**, **sarwar** at terminal **ttyp0** and **bobk** at terminal **ttyC2**. The receiver's screen is garbled with the message, but no harm is caused to any work that the user is doing. Under the shell, pressing `<Enter>` performs the trick of resetting the screen, and inside the `vi` editor (discussed in Chapter 3), the screen can be reset by pressing the `<Ctrl>` and `R` keys on the keyboard at the same time. Notice also that the sending of the message is accomplished by holding down the `<Ctrl>` and `D` keys on the keyboard at the same time.

```
Sender's (sarwar) screen
$mesg y
$who
bobk      upibm7:ttyC2     Oct12     13:47 :34
sarwar    upibm7:ttyp0     Oct12     14:20 :15
$write bobk ttyC2
Bob,
How are the new chapter revisions coming along?
Take care,
Mansoor
<Ctrl+D>

Receiver's (bobk) screen
```

```
$mesg y
$
Message from sarwar@upibm7.egr.up.edu on ttyp0 at 14:26
Bob,
    How are the new chapter revisions coming along?
Take care,
Mansoor
EOF
```

The `mesg` command enables or disables real-time one-way messages and chat requests from other users with the write and talk commands, respectively. The `mesg y` command permits others to initiate communication with you by using the `write` or `talk` command. If you think that you are bothered too often with `write` or `talk`, you can turn off the permission by executing the `mesg n` command. When you do so, a user who runs a `write` or `talk` command sees the message `Permission denied`. When the `mesg` command is used without an argument, it returns the current value of permission, `n` or `y`.

The `biff` command lets the system know whether you want to be notified immediately of an incoming e-mail message. The system notifies you by sounding a beep on your terminal. You can use the command `biff y` to enable notification and `biff n` to disable notification. When the `biff` command is used without an argument, it displays the current setting, `n` or `y`.

2.6 COMMAND ALIASES

The `alias` command can be used to create pseudonyms, or nicknames, for commands. The `alias` command has one syntax in the Bourne, Korn, and Bourne Again (Bash; the default shell in Solaris) shells, and another in the C shell (the default shell for PC-BSD); both forms are illustrated in the following example. The general syntax for the `alias` command is as follows:

SYNTAX

```
alias [name [=string] ...]in Bourne, Korn, Bash shells
alias [name [string]]in C shell
```

Purpose: Create pseudonym `string` for the command `name`
Output: Pseudonyms that can be used for commands

Nicknames are usually created for commands, but they can also be used for other items, such as naming e-mail groups. Both C shell and Bash allow you to create aliases from the command line one at a time, or put them multiply in the resource file for the particular shell.

Command aliases can be placed in the **.profile** file or the **.login** file, but they are typically placed in the **.bashrc** file (for the Bash shell in Solaris) and the **.cshrc** file (for the C

TABLE 2.2 Some Useful Aliases for Various Shells

Bourne, Korn, and Bash Shells	C Shell
`alias dir='ls -la\!*'`	`alias dir 'ls -la\!*'`
`alias rename='mv\!*'`	`alias rename 'mv\!*'`
`alias spr='lpr -Pspr\!*'`	`alias spr 'lpr -Pspr\!*'`
`alias ls='ls -C'`	`alias ls 'ls -C'`
`alias ll='ls -ltr'`	`alias ll 'ls -ltr'`
`alias page='more'`	`alias page 'more'`

shell in PC-BSD). The **.profile** or **.login** file executes when you log on, and the **.cshrc** or **.bashrc** file executes every time you start a C or Bourne shell.

Table 2.2 lists some useful aliases to put in one of these files. If set in your environment by any of the these means, the aliases in the session below allow you to use the names `dir`, `rename`, `spr`, `ls`, `ll`, and `page` as commands, substituting them for the actual commands given in quotes. Thus when you type `dir unixbook`, the shell executes the `ls -la unixbook` command.

When you use the `alias` command without any argument, it lists all the aliases currently set by default.

The following session illustrates the use of this command with a Bourne, Korn, or Bash shell.

The aliases shown are those found on our PC-BSD system, and may not be the same as the ones defined by default on your system.

```
$ alias
dir='ls -la'
rename='mv'
spr='lpr -Pspr'
ls='ls -C'
ll='ls -ltr'
page='more'
$
```

Running the same command with the C shell produces the following output:

```
% alias
dir     ls -la
rename      mv
spr     lpr -Pspr
ls      ls -C
ll      ls -ltr
page    more
%
```

You can use the `unalias` command to remove one or more aliases from the alias list.

In Solaris, while in the Bash shell, you can use the `unalias -a` option to remove all aliases from the alias list. You can also use `unalias -a` in PC-BSD if you are in the Bash shell. You cannot use `unalias -a` in the C shell by default in PC-BSD or Solaris, you must `unalias` each alias one at a time.

In the following PC-BSD/Solaris/Bash session, the first of the two `unalias` commands removes the `alias` for `ls`, and the second removes all of the aliases from the alias list. Note that the output of the first `alias` command does not contain an alias for the `ls` command after the `unalias ls` command has been executed. Use of the second `alias` command produces no output because the `unalias -a` command removes all the aliases from the alias list.

```
$ unalias ls
$ alias
dir='ls -la'
rename='mv'
spr='lpr -Pspr'
ll='ls -ltr'
page='more'
$ unalias -a
$ alias
$
```

In the following in-chapter exercises, you will use the `write`, `alias`, and `unalias` commands to practice their syntax and gain more insight into their utility. You will also examine a system file that keeps track of users that can log in.

EXERCISE 2.6

Use the `write` command to communicate with a friend who is logged on to the system.

EXERCISE 2.7

Use the `alias` command to display the nicknames (aliases) of commands in your system, if there are any. If there aren't any, create a few useful ones for yourself according to what you might use frequently and beneficially as a nicknamed command. Then, in PC-BSD use `unalias` to remove one or more of them. In Solaris use `unalias -a` to remove all of the aliases. After you have unaliased all the defaults or defined aliases, how do you reinstate them?

EXERCISE 2.8

Display the contents of the **/etc/passwd** file on your system to determine how many users can log on to the system.

Table 2.3 shows some useful commands for beginners.

TABLE 2.3 Useful Commands for the Beginner

Command	What It Does
<Ctrl+D>	Terminates a process or command
alias	Allows you to create pseudonyms for commands
biff	Notifies you of new e-mail
cal	Displays a calendar on screen
cat	Allows joining of files
cd	Allows you to change the current working directory
cp	Allows you to copy files
exit	Ends a shell that you have started
hostname	Displays the name of the host computer that you are logged on to
login	Allows you to log on to the computer with a valid username/password pair
lpr or lp	Allows printing of text files
ls	Allows you to display names of files and directories in the current working directory
man	Allows you to view a manual page for a command or topic
mesg	Allows or disallows writing messages to the screen
mkdir	Allows you to create a new directory
more	Allows viewing of the contents of a file one screen at a time
mv	Allows you to move the path location of, or rename, files
passwd	Allows you to change your password on the computer
pg	Solaris command that displays one screen of a file at a time
pwd	Allows you to see the name of the current working directory
rm	Allows you to delete a file from the file structure
rmdir	Allows deletion of directories
talk	Allows you to send real-time messages to other users
telnet	Allows you to log on to a computer on a network or the Internet
unalias	Allows you to undefine pseudonyms for commands
uname	Displays information about the operating system running the computer
whatis	Allows you to view a brief description of a command
whereis	Displays the path(s) to commands and utilities in certain key directories
who	Allows you to find out login names of users currently on the system
whoami	Displays your username
write	Allows real-time messaging between users on the system

2.7 INTRODUCTION TO UNIX SHELLS

When you log on and enter a CUI using a console window or terminal, the UNIX system starts running a program that acts as an interface between you and the UNIX kernel. This program, called a *UNIX shell*, executes the commands that you have typed on the keyboard. When a shell starts running, it gives you a prompt and waits for your commands. When you type a command and press <Enter>, the shell interprets your command and executes it. If you type a nonexistent command, the shell tells you this, then redisplays the prompt and waits for you to type the next command. Because the primary purpose of the shell is to interpret your commands, it is also known as the *UNIX command line interpreter*.

A shell command can be internal/built-in or external. The code to execute an internal command is part of the shell process, but the code to process an external command resides in a file in the form of a binary executable program file or a shell script. (We describe in detail how a shell executes commands in Chapter 10.) Because the shell executes commands entered from the keyboard, it terminates when it finds out that it cannot read anything else from the keyboard. You can inform your shell of this by pressing `<Ctrl+D>` at the beginning of a new line. As soon as the shell receives `<Ctrl+D>`, it terminates and logs you off the system. The system then displays the `login:` prompt again, informing you that you need to log on again in order to use it.

The shell interprets single UNIX commands that are structured according to Section 2.2—that is, by assuming that the first word in a command line is the name of the command that you want to execute. It assumes that any of the remaining words starting with a hyphen (-) are options (possibly followed by option arguments) and that the rest are the command arguments.

After reading your command line, it determines whether the command is an internal or external command. It processes all internal commands by using the corresponding code segments that are within its own code. To execute an external command, it searches several directories in the file system structure (see Chapter 4), looking for a file that has the name of the command. It then assumes that the file contains the code to be executed and runs the code.

The names of the directories that a shell searches to find the file corresponding to an external command are stored in the shell variable named `PATH` (or `path` in the C shell). Directory names are separated by colons in the Bourne, Korn, and Bash shells and by spaces in the C shell. The directory names stored in the PATH variable form what is known as the *search path* for the shell. You can view the search path for your variable by using the `echo $PATH` command in the Bourne, Korn, Bash, and C shells.

The following are two sample sessions run with this command in the login shell, first the Bourne shell and then second the C shell. Note that in the Bourne shell the search path contains the directory names separated by colons and that in the C shell the directory names are separated by spaces.

```
$ echo $PATH
/usr/sbin:/usr/X11/include/X11:.:/users/faculty/sarwar/bin:/usr/
ucb
:/bin:/usr/bin:/usr/include:/usr/X11/lib:/usr/lib:/etc:/usr/etc:/
usr
/local/bin:/usr/local/lib:/usr/local/games:/usr/X11/bin
$
% echo $path
/usr/sbin /usr/X11/include/X11 . /users/faculty/sarwar/bin /usr/
ucb /bin
/usr/bin /usr/include /usr/X11/lib /usr/lib /etc /usr/etc /usr/
local/bin /usr/local/lib /usr/local/games /usr/X11/bin
%
```

The `PATH` (or `path`) variable is defined in a hidden file (also known as a dot file) called **.profile** (Solaris) or **.login** (PC-BSD). If you can't find this variable in one of those files, it is in the start-up file (also a dot file) specific to the shell that you're using. You can change the search path for your shell by changing the value of this variable. To change the search path temporarily for your current session only, you can change the value of PATH at the command line. For a permanent change, you need to change the value of this variable in the corresponding dot file.

In the following Bash shell example, the search path was augmented by two directories, **~/bin** and **.** (current directory). Moreover, the search starts with **~/bin** and ends with the current directory.

Be careful when editing or changing the PATH variable, so that you don't lose any component of the default search path set by the system administrator for all users of the system.

```
$ PATH=~/bin:$PATH:.
$
```

You can determine your login shell by using the `echo $SHELL` command, as described in Section 2.8.3. Each shell has several other environment variables set up in a hidden file associated with it. We describe these files in Section 2.8.4 and present a detailed discussion of UNIX files in Chapter 4.

2.8 VARIOUS UNIX SHELLS

Every UNIX system comes with a variety of shells, with the Bash and C shells (the default shells in our base systems, Solaris and PC-BSD) being the most common. The Bourne, Korn, TC, and Z shells are less popular, but offer advantages for certain applications and ways of working with the UNIX system. When you log on, one particular type of shell starts execution. This shell is known as your *login shell*, and it is determined by the system administrator of your UNIX system. If you want to use a different shell, you can do so by running a corresponding command available on your system. For example, if your login shell is Bash, but you want to use the C shell, you can do so by using the `csh` command.

2.8.1 Shell Programs

Essentially the shell program itself, which is implemented in the C programming language, allows you to do interpreted programming (as opposed to compiled programming). It does this in two senses: firstly, so you can employ simple, single or complex, multiple UNIX commands connected by redirection operators and/or utilities such as `sed`, `awk` or `grep`, to do common tasks, and secondly via user-written script files, coded in the shell interpreted language, that automate and simplify those common, perhaps highly repetitive, tasks. This interpreted language has all the features of any other structured, high-level programming language, such as Perl, Tcl, or Python. The shell language is just not as complex as the other common scripting languages. This fact should tell you why there are so many different shells, just as there are many different high-level programming and scripting languages.

TABLE 2.4 Shell Locations and Program Names

Shell	Location on PC-BSD System	Program (Command) Name
rc	NA	`rc`
Bourne shell	/bin/sh	`sh`
C shell	/bin/csh	`csh`
Bourne Again shell	/usr/local/bin/bash	`bash`
Z shell	NA	`zsh`
Korn shell	NA	`ksh`
TC shell	/bin/tcsh	`tcsh`

Programming languages have a tendency to evolve and grow with time, depending on the needs of users, and shell programs are typical of this evolution. Table 2.4 contains a list of the most common shells, their location on a PC-BSD system, and the program names of those shells.

The locations shown in Table 2.4 are typical for most UNIX systems. Consult your instructor or system administrator if you can't find the location shown for a shell on your system or if you can't use the `whereis` command, as shown in Section 2.5.1.

Figure 2.7 traces the development of various shell families, and indicates the increasing functionality of each family as it appears higher in the hierarchy. The Bourne shell (`sh`) is the *grandmother* of the main shell families and has nearly the least level of functionality. Near the top of the hierarchy is the Korn shell (`ksh`), which includes all the functionality of the Bourne shell and much more. The `rc` and `zsh` shells are outliers that cannot be readily associated with any of the primary shell families.

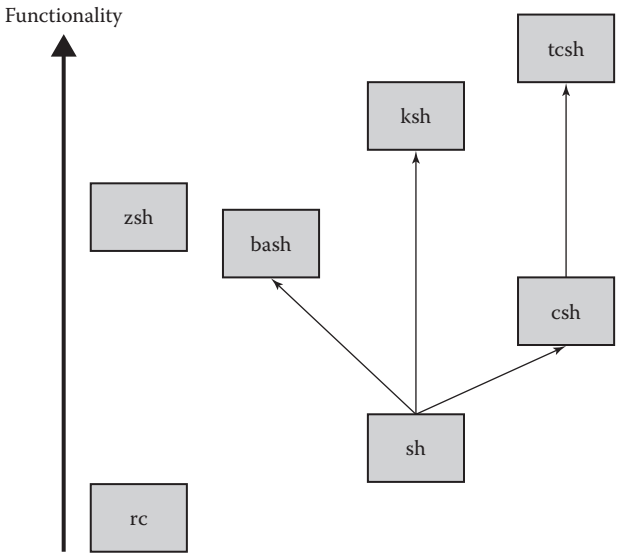

FIGURE 2.7 Shell families and their relative functionalities.

2.8.2 Which Shell Suits Your Needs?

Most shells perform similar functions, and knowing the details of how they do so is important in deciding which shell to use for a particular task. Also, using more than one shell during a session is a common practice, especially among shell script file programmers. For example, you might use the Bourne or Korn shell for their programming capabilities and use the C shell to execute individual commands. We discuss this example further in Section 2.8.3. The similarities of major shell functions are summarized in Table 2.5.

2.8.3 Ways to Change Your Shell

You can easily determine what your default shell is by typing `echo $SHELL` on the command line when you first log on to your computer system.

The question is: Why would you want to change your default shell, or for that matter, even use an additional shell? The answer is that you want the greater, or in some sense qualitatively different, functionality of another shell.

For example, your default shell might be the C shell (`csh`). A friend of yours offers you a neat and useful Bourne shell script that allows you to take advantage of the Bourne shell script programming capabilities, a script that wouldn't work if it ran under the C shell. You can use this script by running the Bourne shell at the same time you are running the default C shell. Because UNIX is a multiprocess operating system, more than one command line interpreter at a time can be active. That doesn't mean that a single command will be interpreted multiply; it simply means that input, output, and errors are "hooked" into whatever shell process has control over them currently. (See Chapter 10 for more information about process and shell command input/output.)

You can change your shell in one of two ways:

1. Changing to a new default for every subsequent login session on your system, and
2. Creating additional shell sessions running on top of, or concurrently with, the default shell.

The premise of both methods is that the shell you want to change to is available on your system.

To change your default shell, after you have logged on, type `chsh` and then press `<Enter>`. Depending on your system, you will be prompted for the name of the shell you want to change to.

On a PC-BSD system, you are prompted for the superuser password in order to accomplish the `chsh` command. There is no `chsh` command available in Solaris.

TABLE 2.5 Shell Similarities

Function	Description
Execution	The ability to execute programs and commands
I/O handling	The control of program and command input and output
Programming	The ability to execute sequences of programs and commands

Type the location of the shell you want to change to—for example, **/usr/bin/sh** to change to the Bourne shell. If this method doesn't work on your system, consult your instructor or system administrator for more help.

To create or run additional shells on top of your default shell, simply type the name of the shell program (see Table 2.4) on the command line whenever you want to run that shell. The following session illustrates the use of this method to change a default Bash shell, which uses the `$` as the shell prompt, to a C shell, which shows the `%` as the shell prompt.

```
$ echo $SHELL
/usr/bin/bash
$ csh
%
```

The first command line allows you to determine your default shell. In this case, the system shows you that the default setting is the Bash shell. The second command line allows you to run the C shell. The fourth line shows that you have been successful, because the default C shell prompt appears on your display. If the C shell was not available on your system or was inaccessible to you, you would get an error message after the third line. If your search path does not include `/usr/bin`, you either have to type `/usr/bin/csh` in place of `csh`, or include `/usr/bin` in your shell's search path and then use the `csh` command.

To terminate or leave this new, temporary shell and return to your default login shell, hold down `<Ctrl+D>` on a blank line. If this way of terminating the new shell doesn't work, type `exit` on the command line and then press `<Enter>`. By doing so, you halt the running of the new shell, and the default shell prompt appears on your display. If you have opened a console or terminal window on your desktop, typing `exit` also closes this console or terminal window.

The following in-chapter exercises ask you to determine whether various shells are available on your system by using the `whereis` command and, for those that are available, to read the manual pages for them by using the `man` command.

EXERCISE 2.9

Using the `whereis` command illustrated in Section 2.5.1, verify the locations of the various shells listed in Table 2.4. Are all these shells available on your system? Where are they located if you do not find them at the locations shown in Table 2.4?

EXERCISE 2.10

Using the `man` command illustrated in 2.4.5, read the manual pages for each shell listed in Table 2.4 that is on your system.

2.8.4 Shell Start-Up Files and Environment Variables

The actions of each shell, the mechanics of how it executes commands and programs, how it handles the command and program I/O, and how it is programmed, are affected by the setting of certain *environment variables*.

Each UNIX system has an initial system start-up file, usually named **.profile** in Solaris and **.login** in PC-BSD. This file contains the initial settings of important environment variables for the shell and some other utilities. In addition, hidden files for specific shells are executed when you start a particular shell. Known as the shell start-up files, they are **.cshrc** for C shell and **.bashrc** for Bash. These hidden files are initially configured by the system administrator for secure use by all users. Table 2.6 lists some important environment variables common to Bash, Bourne, Korn, and C shells; the C shell variable name, where applicable, is in lowercase following the Bash, Bourne, and Korn shell variable name. Note that your system administrator may not have set some of these variables, such as `ENV`.

The following in-chapter exercises let you view the settings of your environment variables. They assume that you are initially running the Bourne or Korn shells. If you aren't, run either of those shells as described in Section 2.8.3 and then do the exercises.

EXERCISE 2.11

At the default login shell prompt for your system, type `set | more` and then press `<Enter>`. What is displayed on your screen? Identify and list the settings for all the environment variables shown in Table 2.6.

EXERCISE 2.12

At the shell prompt, type `csh` or `bash` depending on your default system login shell, and then press `<Enter>`. Next, type `setenv | more` and then press `<Enter>`. Identify and list the settings for all the environment variables shown in Table 2.6.

In addition to the shells, several other programs have their own hidden files. These files are used to set up and configure the operating environment within which these programs execute. We discuss some of these hidden files in Chapters 4 and 5. They are called hidden files because when you list the names of files contained in your home directory—for example, with the `ls -l` command and option (see Chapter 4)—these files do not appear on the list. The hidden file names always start with a period (.), such as **.login**.

TABLE 2.6 Shell Environment Variables

Environment Variable	What It Affects
`CDPATH, cdpath`	The alias names for directories accessed with the **`cd`** command
`EDITOR`	The default editor used in programs such as the e-mail program Elm
`ENV`	The path along which UNIX looks to find configuration files
`HOME, home`	The name of the user's home directory when the user first logs on
`MAIL, mail`	The name of the system mailbox file
`PATH, path`	The directories that a shell searches to find a command or program
`PS1, prompt`	The shell prompt that appears on the command line
`PWD, cwd`	The name of the current working directory
`TERM`	The type of console terminal being used

2.9 SHELL METACHARACTERS

Most of the characters other than letters and digits have special meaning to the shell. These characters are called *shell metacharacters* and, therefore, cannot be used in shell commands as literal characters without specifying them syntactically in a particular way. Thus, try not to use them in naming your files. Also, when these characters are used in commands, no space is required before or after a character. However, you can use spaces before and after a shell metacharacter for clarity. Table 2.7 contains a list of the shell metacharacters and their purposes.

TABLE 2.7 Shell Metacharacters

Metacharacter	Purpose	Example
<New Line>	To end a command line	
<Space>	To separate elements on a command line	ls /etc
<Tab>	To separate elements on a command line	ls /etc
#	To start a comment	# This is a comment line
"	To quote multiple characters but allow substitution	"$file" bak
$	To end line and dereference a shell variable	$PATH
&	To provide background execution of a command	command &
'	To quote multiple characters	'$100,000'
( )	To execute a command list in a subshell	(command1; command2)
*	To match zero or more characters	chap*.ps
[ ]	To insert wild cards	[a-s] or [1,5-9]
^	To begin a line and negation symbol	[^3-8]
'	To substitute a command	PS1='command'
{ }	To execute a command list in the current shell	{command1; command2}
|	To create a pipe between commands	command1 | command2
;	To separate commands in sequential execution	command1; command2
<	To redirect input for a command	command < file
>	To redirect output for a command	command > file
?	To substitute a wild card for **lab.?** exactly one character	
/	To be used as the root directory and /usr/bin as a component separator in a pathname	
	To escape/quote a single character; n command arg1\used to quote <New Line> character arg2 arg3 to allow continuation of a shell \? command on the following line	
C and Korn Shells Only		
!	To start an event specification in the history list and the current event	!!, !$
%	The C shell prompt, or the starting character for specifying a job number	% or %3
~	To name home directory	~/.profile

The shell metacharacters allow you to specify multiple files in multiple directories in one command line. We describe the use of these characters in subsequent chapters, but we give some simple examples here to explain the meanings of some commonly used metacharacters:

- `*`
- `?`
- `~`
- `[ ]`

The `?.txt` string can be used for all the files that have a single character before **.txt**, such as **a.txt**, **G.txt**, **@.txt**, and **7.txt**. The `[0-9].c` string can be used for all the files in a directory that have a single digit before **.c**, such as **3.c** and **8.c**. The `lab1/c` string stands for **lab1/c**. Note the use of backslash (\) to quote (escape) the slash character (/).

The following command prints the names of all the files in your current directory that have two-character file names and an .html extension, with the first character being a digit and the second being an uppercase or lowercase letter. The printer on which these files are printed is **spr**.

```
$ lpr -Pspr [0-9][a-zA-Z].html
$
```

Note that **`[0-9]`** means any digits from 0 through 9 and **`[a-zA-Z]`** means any lowercase or uppercase letter. The following command displays the names of all six-character-long files with **.c** extension in your current directory, with the first three characters being `lab`, the fourth being a digit, and the remaining being any two characters.

```
$ ls lab[0-9]??.c
lab11a.clab1a1.c lab123.clab4ab.c
$
```

2.10 THE SUDO AND SU COMMANDS

The `sudo` command allows a permitted user to execute a command as the superuser, or to assume the role of another user, as specified by security policy. The `su` command allows an ordinary user to switch user roles or to also simulate being the superuser on the system. The superuser has file permission and access privileges to everything on the system.

In many of the operations shown in the following chapters, particularly in Chapter 23 on system administration, it will be necessary to execute the `su` command in order to accomplish the tasks shown. In order to use this command, it is necessary to know the root or superuser password.

We give a more complete explanation of the sudo command in Chapter 23, Section 23.9.3.1.

An example of using the su command on a PC-BSD system is as follows:

```
[bob@pcbsd-923]% su
Password: XXX
[bob@pcbsd-923] /usr/home/bob#
```

SUMMARY

The UNIX operating system is most famous for its text-based command execution, but in the twenty-first century it has a competitively developed GUI environment as well. This chapter serves to familiarize you with the basic structure of a CUI UNIX command. It also shows you how to log in via three popular and typical login methods, and how to gracefully log off.

A beginner must be able to do basic file maintenance, and a core set of CUI file maintenance commands and their options are introduced in this chapter. These commands will be useful throughout the rest of this book. Finally, we illustrate and give examples of some basic utility commands—most importantly, the commands and their options that allow you to print files and the alias command.

When you log on to a UNIX computer, the system runs a program called a shell that gives you a prompt and waits for you to type commands, either as single commands or as multiple commands connected by redirection or piping operators. The shell program, coded in C, is an interpreter, and as such has the same structured programming capabilities of high-level languages. When you type a command and press <Enter>, the shell interprets and tries to execute the command, assuming that the first word in the command line is the name of the command. A shell command can be built-in or external. The shell has the code for executing a built-in command, but the code for an external command is in a file. To execute an external command, the shell searches several directories, one by one, to locate the file that contains the code for the command. If the file is found, it is executed if it contains code (binary or shell script). The names of the directories that the shell searches to locate the file for an external command form are known as the search path. The search path is stored in a shell variable called PATH (for the Bourne, Korn, and Bash shells) or path (for the C shell). You can change the search path for your shell by adding new directory names in PATH or by deleting some existing directory names from it.

Several shells are available for you to use. These shells differ in terms of convenience of use at the command line level and features available in their programming languages. The most commonly used shells in a UNIX-based system are the Bash and C Shells. The Bourne shell is the oldest and has a good programming language. The C shell has a more convenient and rich command-level interface. The Korn shell has some good features of both and is a superset of the Bourne shell.

Certain characters, called shell metacharacters, have special meaning to the shell. Because the shell treats them in special ways, they should not be used in file names. If you must use them in commands, you need to quote them for the shell to treat them literally.

QUESTIONS AND PROBLEMS

1. Create a directory called UNIX in your home directory. What command line did you use to do this?
2. Give a command line for displaying the files **lab1**, **lab2**, **lab3**, and **lab4**. Can you give two more command lines that do the same thing? What is the command line for displaying the files **lab1.c**, **lab2.c**, **lab3.c**, and **lab4.c**? (Hint: use shell metacharacters.)
3. Give a PC-BSD command line for printing all the files in your home directory that start with the string memo and end with **.ps** on a printer called **upmpr**. What command line did you use to do this?
4. Give the command line for nicknaming the command `who -H` as `W`. Give both Bourne and C shell versions. Where would you put it if you want it to execute every time you start a new shell?
5. Type the command man `ls > ~/UNIX/ls.man` on your system. This command will put the man page for the `ls` command in the **ls.man** file in your **~/UNIX** directory (the one you created in Problem 1). Give the command for printing two copies of this file on a printer in your lab. What command line would you use on PC-BSD to achieve this printing? What command would you use on Solaris to achieve this printing?
6. What is the `mesg` value set to for your environment? If it is on, how would you turn off your current session? How would you set it off for every login?
7. What does the command `lpr -Pqpr [0-9]*.jpg` do in PC-BSD? Explain your answer.
8. Use the `passwd` command to change your password. If you are on a network, be aware that you might have to use the `yppasswd` command to modify your network login password. Also, make sure you abide by the rules set up by your system administrator for coming up with good passwords!
9. Using the correct terminology (e.g., command, option, option argument, and command argument), identify the constituent parts of the following UNIX single commands.
   ```
   ls -la *.exe
   lpr -Pwpr file27
   chmod g+rwx *.*
   ```
10. View the man pages for each of the useful commands listed in Table 2.3. Which part of the man pages is most descriptive for you? Which of the options shown on each of the man pages is the most useful for beginners? Explain.
11. How many users are logged on to your system at this time? What command did you use to discover this?

12. Determine the name of the operating system that your computer runs. What command did you use to discover this?
13. Give the command line for displaying manual pages for the socket, read, and connect system calls on a PC-BSD system. What will be the command line for a Solaris computer?
14. What is a shell? What is its purpose?
15. What are the two types of shell commands? What are the differences between them?
16. Give names of five UNIX shells. Which are the most popular? What is a login shell? What do you type in to terminate the execution of a shell? How do you terminate the execution of your login shell?
17. What shells do you think are *supersets* of other shells? In other words, which shells have other shells' complete command sets plus their own? Can you find any commands in a subset shell that are not in a superset shell? Refer to Figure 2.7.
18. What is the search path for a shell? What is the name of a shell variable that is used to maintain it for the Bourne, C, and Korn shells? Where (i.e., in which file) is this variable typically located?
19. What is the search path set to in your environment? How did you find out? Set your search path so that your shell searches your current and your **~/bin** directories while looking for a command that you type. In what order does your shell search the directories in your search path? Why?
20. What are hidden files? What are the names of the hidden files that are executed when you log on to System V and BSD UNIX systems?
21. What is a shell start-up file? What is the name of this file for the C shell? Where (i.e., in which directory) is this file stored?
22. What important features of each shell, as discussed on the manual pages for that shell, seem to be most important for you as a new, intermediate, or advanced user of UNIX? Explain the importance of these features to you in comparison with the other shells available and their features.
23. Suppose that your login shell is a C shell. You receive a shell script that runs with the Bourne shell. How would you execute it? Clearly write down all the steps that you would use.

Editing Text Files

Objectives

- To explain the general utility of editing text files on a UNIX system
- To show the basic capabilities of vi, vim, and gvim, and how to customize them
- To show the basic capabilities of GNU emacs and how to customize it
- To cover the commands and primitives

```
cp, emacs, gvim, ls, pwd, sh, vi, vim, who
```

3.1 INTRODUCTION AND QUICK START

In this chapter, we use the following editors that are commonly available in both of our base modern UNIX systems, PC-BSD and Solaris: vi, vim, gvim, and GNU emacs.

3.1.1 Quick Start: The Simplest Path through These Editors

To stress how the keyboard keys are used in these editors, we provide the following reference to the keys used to execute commands or change modes:

1. Pressing the Escape key is signified as `<Esc>`
2. Pressing the Enter key is signified as `<Enter>`
3. Pressing the `<Ctrl>` key in combination with another single key is signified as `<Ctrl+X>`, where you hold down the `<Ctrl>` key and press the `X` key (or any valid key for that combination) at the same time.
4. Pressing the Alt key in combination with another single key is signified as `<Alt+X>`, where you hold down the `<Alt>` key and press the `x` key (or any valid key for that combination) at the same time.

5. A variant of 3. and 4. is shown as `<Ctrl+X> a [b]`, where you first press and release `<Ctrl>` and `x` simultaneously, then press the `a` key, and optionally press the `b` key (or any valid combination of single keys or strings of characters).
6. In GNU emacs for PC-BSD and Solaris, the `Meta` key that is referred to in much of the literature on GNU emacs is the `<Alt>` key.

What you type or hold down on the keyboard is shown in **`bold`** text.

3.1.1.1 For vi, vim, and gvim

- At the shell prompt, run the program by typing **`vi file1`** then press **`<Enter>`**.
- Type **`A`**.
- Type some text.
- Press **`<Esc>`**.
- Type **`:`** (colon).
- Type **`wq`** then press **`<Enter>`**.

You now have a file in your default directory named **file1** with the text you typed in it.

If GNU emacs is not installed on your system, skip ahead to the next subsection for general instructions on how to install it. Then do the following:

3.1.1.2 For GNU emacs

- At the shell prompt, run the program by typing **`emacs file2`** then press **`<Enter>`**.
- Type some text.
- Hold down **`<Ctrl+U>`**, then **`<Ctrl+X>`**, then **`<Ctrl+C>`**.

You now have a file in your default directory named **file2** with the text you typed in it.

3.1.2 First Comments on UNIX Editors

As you can see from Section 3.1.1, with vi, vim, and gvim, you can't immediately begin to enter text into the file you are editing. You have to be in *Insert mode* to do that; that's what typing `A` as the second step is doing. Vi, vim, and gvim have modes.

In GNU emacs, you can start typing text into the file immediately. Emacs is a *modeless* editor.

We present the tutorial information in this chapter using typed commands, and by using graphical modes of input and editing.

It is very important to realize that vi, vim, and gvim all generally use the same commands and have basically the same functionality. But vim and gvim are not only more

graphical—allowing you to work more efficiently in GUI environments such as those on our base modern UNIX systems, PC-BSD and Solaris—but they also have an improved and expanded command structure. This will become more evident to you, for example, in Section 3.2.9, where vim has special improved macro-writing capabilities that vi does not.

At the time of writing, both PC-BSD and Solaris have vi, vim, and gvim preinstalled if you have done a basic installation of the system as detailed at the beginning of Chapter 23. But GNU emacs is not preinstalled. Therefore, you must look ahead to Chapter 23, Section 23.7, "System Updates and Software Upgrades by Using a Package Manager." In that section, you will be shown how to use the appropriate package management facilities to obtain GNU emacs for your system. In the App Café in PC-BSD, you must browse **Categories>Editors for Emacs** and install it. From the Solaris IPS repository, you must browse **Development>Editors**, and install `gnu-emacs`.

We will not cover the details of the installation of any of the editors we demonstrate here. In addition, if you are logging into a UNIX system via a terminal window, such as with PuTTY from a Windows machine, many of the graphical modes and techniques shown in this chapter will not be available to you. But that does not prevent you from using the traditional typed commands and keyboard edits that we show.

3.1.3 Using Text Editors

Modern UNIX uses both a GUI, with powerful window management systems like Gnome and KDE, and a CUI. Therefore, to do useful things such as execute multiple commands from within a script file, write e-mail messages, or create C language programs, you must be familiar with one or perhaps multiple ways of entering text into a file. In addition, you must also be familiar with how to edit existing files efficiently—that is, to change their contents or otherwise modify them in some way. Text editors allow you to view a file's contents, similar to the `more` command, so that you can identify the key features of the file, and then read and utilize the information contained in it. For example, a file without any extension, such as **foo** (rather than **foo.txt**) might be a text file that you can view with a text editor.

The editors that we consider here are all considered full-screen display editors. That is, on the display screen or monitor that you are using to view or edit a file, you are able to see a portion of the file, which fills most or all of the window allocated to the text editor screen display. You are also able to move the cursor, or point, to any of the text you see in this full-screen display, with either the arrow keys on the keyboard or with a mouse. That text material is usually held in a temporary storage area in computer memory called the editor *buffer*. If your file is larger than one screen, the buffer contents change as you move the cursor through the file. The difference between a file, which you edit, and a buffer is crucial. For text-editing purposes, a file is stored on disk as a sequence of data. When you edit that file, you edit a copy that the editor creates, which is in the editor buffer. You make changes to the contents of the buffer—and can even manipulate several buffers at once—but when you save the buffer, you write a new sequence of data to the disk, thereby saving the file.

TABLE 3.1 Basic Text-Editing Functions

Function	Description
Cursor movement	Moving the location of the insertion point or current position in the buffer
Cut or copy, paste	"Ripping out" text blocks or duplicating text blocks, reinserting ripped or duplicated blocks
Deleting text	Deleting text at a specified location or in a specified range
Inserting text	Placing text at a specified location
Opening, starting	Opening an existing file for modification, beginning a new file
Quitting	Leaving the text editor, with or without saving the work done
Saving	Retaining the buffer as a disk file
Search, replace	Finding instances of text strings, replacing them with new strings

Another important operational feature of all the editors discussed in this chapter is that, traditionally, their actions are based on keystroke commands, whether they are a single keystroke or combinations of keys pressed simultaneously or sequentially. Because one of the primary input devices in UNIX is the keyboard, using the correct syntax of keystroke commands is mandatory. But the keyboard method of input, once you have become accustomed to it, is as efficient or, for some users, even more efficient than mouse/GUI input. Keystrokes also are more flexible, giving you more complete and customizable control over editing actions. Generally, you should choose the editor you are most comfortable with, in terms of the way you prefer to work with the computer. However, your choice of editor also depends on the complexity and quantity of text creation and manipulation that you want to do. Practically speaking, editors such as vi, vim, gvim, and GNU emacs are capable of handling complex editing tasks in multiple windows on multiple files, and provide you with a visual software development environment, as well as document production and management capability. But to take advantage of that power, you have to learn the mechanics of the commands that are needed to perform those tasks and how they are implemented either graphically or by typing them—and retain that knowledge. The basic functions common to the text editors that we cover here are listed in Table 3.1, along with a short description of each function.

3.2 USING THE vi, vim, AND gvim EDITORS

The vi, vim, and gvim UNIX text editors have almost all the features of a word processor and have tremendous flexibility in creating text files. They are complex to learn, but their advantages give you the ability to create, manipulate, and use the kinds of text files that the full range of UNIX users, from absolute novice to seasoned veteran, commonly work with. We will proceed in the following section and subsections by demonstrating vi as a

text-only interface editor, then move to a more graphical interface approach with vim and gvim.

Buffers: As we mentioned in Section 3.1, the notion of a *buffer* as a temporary storage facility for the text that you are editing is very useful and important in vi, vim, and gvim. The main buffer, sometimes referred to as the editing buffer or the work buffer, is the main repository for the body of text that you are trying to create or to modify from some previous permanently archived file on disk. The general purpose buffer is where your most recent "ripped-out" (cut/copied) text is retained. Indexed buffers allow you to store more than one temporary string of text.

3.2.1 Basic Shell Script File Creation, Editing, Execution

Shell Script File: Practice Session 3.1 shows how to create a script file, or collection of UNIX commands that are executed in sequence, and then execute the script. We present more about shell programming and script files in Chapters 12 through 15. For this example, we assume that you are running the Bourne shell. If you are running some other shell by default, go back to Chapter 2, Section 2.8, and review how to identify and change shells.

In PC-BSD, to run an interactive shell, such as the Bourne shell shown in Practice Session 3.1, on top of your login shell (which is the C shell by default in that system), at the C shell prompt you type `sh` and press `<Enter>`.

If you are using Solaris, do Practice Session 3.1 for the Bourne Again (Bash) shell, which is the default shell in that system, and don't change shells.

And do not worry too much if you make an error in Steps 2, 3, and 4; you can go through the rest of the script file discussion and then come back to this example after you have learned some of the editing commands and become more familiar with them.

3.2.1.1 Practice Session 3.1

Step 1: At the shell prompt, start vi by typing **`vi firscrip`** and then pressing <**Enter**>. The vi screen appears on your display.

Step 2: Type A. Then type **`ls -la`** and then press **`<Enter>`**.

Step 3: Type **`who`** and then press **`<Enter>`**.

Step 4: Type **`pwd`** and then press the **`<Esc>`** key. At this point, your screen should look like that shown in Figure 3.1.

Step 5: Type **`:wq`** and then press **`<Enter>`**.

Step 6: At the shell prompt, type **`sh firscrip`** and then press **`<Enter>`**.

Step 7: Note the results. How many files do you have in your present working directory? What are their names and sizes? Who else is using your computer system? What is your present working directory?

FIGURE 3.1 File **firscrip** after Step 4.

In Practice Session 3.1, you accomplished these things:

- At Step 2, typing `A` took vi out of *Command mode* (which is what vi starts in by default) and placed it in one of the forms of Insert mode. In other words, anything that you typed at the keyboard was appended as text on the first line in the text area of the editor.
- When you pressed the `<Esc>` key in Step 4, vi was taken out of Insert mode and put back into Command mode.
- When you typed `:` in Step 5, that was a valid Command mode prefix character for the two commands that followed, and put vi in *Last Line mode*.
- When you typed `wq` after the `:`, vi interpreted those commands in Last Line mode as write out or save the file, and quit the editor.
- Step 6 executes the Bourne shell script file.

3.2.2 How to Start, Save a File, and Exit

When you need to do UNIX text editing that gives you as much functionality as a typical word processor, you can use the vi text editor. To start vi from the command line, use the following general syntax (anything enclosed in square brackets `[ ]` is optional):

SYNTAX

```
vi [options] [file(s)]
```

Purpose: Allows you to edit a new or existing text file(s)
Output: With no options or file(s) specified, you are placed in the vi program and can begin to edit a new buffer
Commonly used options/features:

+n	Begin to edit file(s) starting at line number `n`
+/exp	Begin to edit at the first line in the file matching string `exp`

The operations that you perform in vi fall into two general categories: *Command mode* operations, which consist of key sequences that are commands to the editor to take certain actions, and *Insert mode* operations, which allow you to input text.

The general organization of the vi text editor and how to start, exit, and switch modes are illustrated in Figure 3.2. The general organization of vim and gvim, and how to start, exit, and switch modes in those editors, is the same as shown for vi in Figure 3.2.

For example, to change from Command mode, which you are in when you first enter the editor, to Insert mode, type a valid command, such as `A` to append text at the end of the current line. Certain commands that are prefixed with the `:`, `/`, `?`, or `:!` characters are echoed or shown to you on the last line on the screen and must be terminated by pressing `<Enter>`. *Last Line mode*, sometimes called *ex mode* because it is derived from the ex editor, allows you to execute certain commands and leave the editor. To change from Insert mode to Command mode, press the `<Esc>` key.

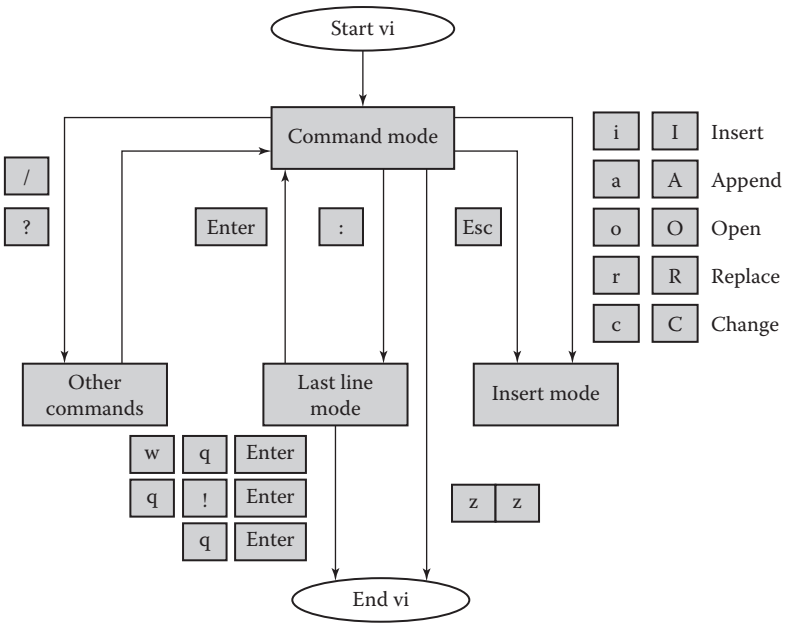

FIGURE 3.2 General organization of vi, vim, and gvim.

The keystroke commands that you execute in vi are case sensitive; for example, uppercase `A` appends new text after the last character at the end of the current line, whereas lowercase `a` appends new text after the character the cursor is on.

To start vi, at the shell prompt, type `vi` (and optionally designate some option[s] and file name[s]) and then press `<Enter>`. You are now in Command mode. To enter Insert mode, type `A` and you are now able to insert text on the first line of the file.

After entering text, you can press the `<Esc>` key to enter Command mode.

At any point in your creation or manipulation of text, you can press the `u` key on the keyboard to undo the last operation.

From Command mode, you can save the text that you just inserted into the buffer to a file on disk by typing `:w filename` and pressing `<Enter>`, where `filename` is the name of the file you want to save the text to. To quit the editor, type `:q`.

3.2.3 The Format of a vi Command and the Modes of Operation

In Command mode, the generic syntax of keystrokes is:

```
[#1] operation [#2] target
```

where:

anything enclosed in [ ] is optional;

#1 is an optional number, such as 5, specifying how many operations are to be done;

operation is what you want to accomplish, such as deleting lines of text;

#2 is an optional number, such as 5, specifying how many targets are affected by the **operation**; and

target is the text that you want to do the operation on, such as an entire line of text.

Note that if the current line is the target of the operation, the syntax for specifying the target is the same as the syntax of the operation; for example, `dd` deletes the current line. Also, a variation on this generic syntax is the cursor movement command, whereby you can omit the numbers and operation and simply move the cursor by word, sentence, paragraph, or section. Table 3.2 lists some specific examples of this generic syntax and variations used in Command mode.

As previously stated, when you start vi, it is in Command mode. When you want to be in Insert mode instead of Command mode, press a valid key to accomplish the change. Some of these keys are shown in Table 3.3.

After inserting text, you can edit the text, move the cursor to a new position in the buffer, and save the buffer and exit the editor—all from within Command mode. When you want to change from Insert mode to Command mode, press the `<Esc>` key.

To save the buffer and exit the editor, press the `:` key (colon) to enter Last Line mode. The general commands that are useful in Last Line mode are shown in Table 3.4.

TABLE 3.2 Examples of Vi Command Syntax

Command	Action
`cw`	Change word.
`cc`	Change line.
`c$`	Change text from current position to end of line.
`C`	Same as `c$`.
`dd`	Delete current line.
`7 dd`	Delete 7 lines.
`d$`	Delete text from current position to end of line.
`D`	Same as `d$`.
`5dw`	Delete 5 words.
`d7,14`	Delete lines 7 through 14 in the buffer.
`d}`	Delete up to next paragraph.
`d^`	Delete back to beginning of line.
`d/ pat`	Delete up to first occurrence of pattern.
`dn`	Delete up to next occurrence of pattern.
`df x`	Delete up to and including x on current line.
`dt x`	Delete up to (but not including) x on current line.
`dL`	Delete up to last line on screen.
`dG`	Delete to end of file.
`gqap`	Reformat current paragraph to text width (vim and gvim).
`g~w`	Switch case of word (vim and gvim).
`guw`	Change word to lowercase (vim and gvim).
`gUw`	Change word to uppercase (vim and gvim).
`p`	Insert last deleted or yanked text after cursor.
`gp`	Same as `p`, but leave cursor at end of inserted text (vim and gvim).
`gP`	Same as `P`, but leave cursor at end of inserted text (vim and gvim).
`]p`	Same as `p`, but match current indention (vim and gvim).
`[p`	Same as `P`, but match current indention (vim and gvim).
`P`	Insert last deleted or yanked text before cursor.
`r x`	Replace character with x. Does not require the use of `<Esc>`!
`R text`	Replace with new text (overwrite), beginning at cursor. `<Esc>` ends replace mode.
`s`	Substitute character. `<Esc>` ends substitute mode.
`4s`	Substitute four characters. `<Esc>` ends substitute mode.
`S`	Substitute entire line. `<Esc>` ends substitute mode.
`u`	Undo last change.
`<Ctrl+R>`	Redo last change (vim and gvim).
`U`	Restore the current line, if you have not moved off of it.
`x`	Delete current cursor position.
`X`	Delete back one character.
`5X`	Delete previous 5 characters
`.`	Repeat last change.
	Change case and move cursor right.
`<Ctrl+A>`	Increment number at the cursor (vim and gvim).
`<Ctrl+X>`	Decrement number at the cursor (vim and gvim).

TABLE 3.3 Important Keys to Switch from Command Mode to Insert Mode

Key	Action
a	Appends text after the character the cursor is on
A	Appends text after the last character of the current line
c	Begins a change operation, allowing you to modify text
C	Changes from the cursor position to the end of the current line
i	Inserts text before the character the cursor is on
I	Inserts text at the beginning of the current line
o	Opens a blank line below the current line and puts the cursor on that line
O	Opens a blank line above the current line and puts the cursor on that line
R	Begins overwriting text
s	Substitutes single characters
S	Substitutes whole lines

TABLE 3.4 Important Commands for Command Mode

Command	Action
`: n, m w file`	Write lines `n` to `m` to new file.
`: n, m w >> file`	Append lines `n` to `m` to existing file.
`:r filename`	Reads and inserts the contents of the file filename at the current cursor position
`:wq`	Saves the buffer and quits
`:w`	Saves the current buffer and remains in the editor.
`:w filename`	Saves the current buffer to filename
`:w! filename`	Overwrites filename with the current text
`:w!`	Write file (overriding protection).
`:w! file`	Overwrite file with current text.
`:w %.new`	Write current buffer named **file** as **file.new.**
`:q`	Quit vi (fails if changes were made).
`:q!`	Quit vi without saving the buffer.
`:Q`	Quit vi and invoke `ex`.
`:vi`	Return to vi after `Q` command.
`ZZ`	Quits vi, saving the file only if changes were made since the last save
`%`	Replaced with current filename in editing commands.
`#`	Replaced with alternate filename in editing commands.

For now, we recommend that you use the arrow keys on the keyboard to move the cursor around in the buffer. It is possible to also use the `h`, `j`, `k`, and `l` keys on the keyboard to move the cursor. In gvim, you can use the mouse and its buttons!

The following practice session introduces you to some of the commands presented in Tables 3.2 through 3.4:

3.2.3.1 Practice Session 3.2

Step 1: At the shell prompt, type **`vi firstvi`** and then press `<Enter>`.

Step 2: Type `A`, then type **`This is the first line of a vi file.`** and then press **`<Enter>`**.

Step 3: Type **`This is the line of a vi file.`** and then press **<Enter>**.

Step 4: Type **`is the 3r line of a vi.`**

Step 5: Press the **<Esc>** key.

Step 6: Type **`:w`** and then press **<Enter>**.

Step 7: Use the arrow keys on the keyboard to position the cursor on the character `l` in the word `line` on the second line of the file.

Step 8: Type **`i`** and then **`2nd`**.

Step 9: Press the **<Esc>** key.

Step 10: Use the arrow keys to position the cursor anywhere on the third line of the file.

Step 11: Type I and then **`This`**.

Step 12: Press the **<Esc>** key.

Step 13: Use the arrow keys on the keyboard to position the cursor on the character `r` in `3r` on this line.

Step 14: Type **`a`** and then **`d`**.

Step 15: Press the **<Esc>** key.

Step 16: Type A and then **`file`**.

Step 17: Press the **<Esc>** key on the keyboard. Your screen display should look similar to Figure 3.3.

Step 18: Type **`:wq`**. You will be back at the shell prompt.

The following in-chapter exercise asks you to apply some of the operations you learned about in the previous practice session.

EXERCISE 3.1

With vi you begin editing a file that you created yesterday. You want to save a copy of it with a different filename while still in vi, but you don't want to quit this editing session. How do you accomplish this result in vi?

EXERCISE 3.2

What happens if you accomplish five operations in vi and then type `5u` when in Command mode?

3.2.4 Cursor Movement and Editing Commands

In Command mode, several commands accomplish cursor movement and text-editing tasks. Table 3.5 lists important cursor movement and keyboard editing commands. As

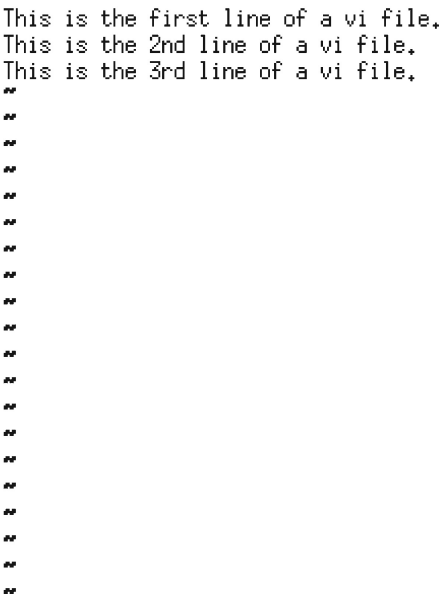

FIGURE 3.3 File **firstvi**.

TABLE 3.5 Cursor Movement and Keyboard Editing Commands

Command	Action
`1G`	Moves the cursor to the first line of the file
`G`	Moves the cursor to the last line of the file
`0` (zero)	Moves the cursor to the first character of the current line
`<Ctrl+G>`	Reports the position of the cursor in terms of line # and column #
`$`	Moves the cursor to the last character of the current line
`w`	Moves the cursor forward one word at a time
`b`	Moves the cursor backward one word at a time
`x`	Deletes the character at the cursor position
`dd`	Deletes the line at the current cursor position
`u`	Undoes the most recent change
`r`	Replaces the character at the current cursor location with what is typed next

we have already noted, character-at-a-time or line-at-a-time moves of the cursor can be accomplished easily with the arrow keys, or alternatively with the `h`, `j`, `k`, and `l` keys on the keyboard.

The following practice session lets you continue editing the file you created in Practice Session 3.2 by using commands presented in Table 3.5.

3.2.4.1 Practice Session 3.3

Step 1: At the shell prompt, type **vi firstvi** and then press **<Enter>**.

Step 2: Type **G**. The cursor moves to the last line of the file.

Step 3: Hold down the **<Ctrl>** and **g** keys at the same time. On the last line of the screen display, vi reports the following:

```
"firstvi" line 3 of 3 - - 100%-- col 1
```

This is a report of the buffer that you are editing, the current line number, the total number of lines in the buffer, the percentage of the buffer that this line represents, and the current column position of the cursor.

Step 4: Type **o**. A new line opens below the third line of the file

Step 5: Type **This is the 5th line of a vi file.** Type **<Esc>**.

Step 6: Type **0** (zero). The cursor moves to the first character of the line you just typed in.

Step 7: Type **$**. The cursor moves to the last character of the current line.

Step 8: Type **O**. A new line opens above the current fourth line.

Step 9: Type **This is the 44th line of a va file.** Type **<Esc>**.

Step 10: Use the arrow keys to position the cursor over the first `4` in `44` on this line.

Step 11: Type **x**.

Step 12: Use the arrow keys to position the cursor over the `a` in `va` on this line.

Step 13: Type **r** and then type **i**.

Step 14: Type **dd**.

Step 15: Type **:wq** to go back to the shell prompt.

Step 16: At the shell prompt, type **more firstvi** and then press **<Enter>**. How many lines with text on them does **more** show in this file?

3.2.5 Yank and Put (Copy and Paste) and Substitute (Search and Replace)

Every word processor is capable of copying and pasting text and also of searching for old text and replacing it with new text. Copying and pasting are accomplished with the vi commands `yank` and `put`. In general, you use yank and put in sequence and move the cursor (with any of the cursor movement commands or methods) only between yanking and putting. Some examples of the syntax for yank and put are given in Table 3.6.

The simple vi forms of search and replace are accomplished using the `substitute` command. This command is executed when vi is in Last Line mode, where you preface the

TABLE 3.6 Examples of Syntax for the Yank and Put Commands

Command Syntax	What It Accomplishes
`y2W`	Yanks two words, starting at the current cursor position, going to the right
`4yb`	Yanks four words, starting at the current cursor position, going to the left
`yy or Y`	Yanks the current line
`p`	Puts the yanked text after the current cursor position
`P`	Puts the yanked text before the current cursor position
`5p`	Puts the yanked text in the buffer five times after the current cursor position
`Y`	Copy current line
`yy`	Copy current line
`" x yy`	Copy current line to register `x`
`ye`	Copy text to end of word
`yw`	Like `ye`, but include the whitespace after the word
`y$`	Copy rest of line
`&quot; x dd`	Delete current line into register `x`
`&quot; x d`	Delete into register `x`
`&quot; x p`	Put contents of register `x`
`y]]`	Copy up to next section heading
`J`	Join current line to next line
`gJ`	Same as `J`, but without inserting a space (vim and gvim)
`:j`	Same as `J`
`:j!`	Same as `gJ`

command with the `:` character and terminate the command by pressing `<Enter>`. The format of the `substitute` command as it is typed on the status line is:

`:[range]s/pattern/string[/option(s)][count]`

where:

anything enclosed in **`[ ]`** is not mandatory;

`:` is the colon prefix for the Last Line mode command;

`range` is a valid specification of lines in the buffer (or the current line is the **`range`**);

`s` or **`substitute`** is the syntax of the substitute command;

`/` is a delimiter for searching;

`pattern` is the text or objects you want to replace;

`/` is a delimiter for replacement;

`string` is the new text or objects;

TABLE 3.7 Examples of Syntax for the Substitute Command

Command Syntax	What It Accomplishes
`:s/john/jane/`	Substitutes the word `jane` for the word `john` on the current line, only once.
`:s/john/jane/g`	Substitutes the word `jane` for every word `john` on the current line.
`:1,10s/big/small/g`	Substitutes the word `small` for every word `big` on lines 1–10.
`:1,$s/men/women/g`	Substitutes the word `women` for every word `men` in the entire file.
`:'<,'>s/this/that/g`	Select the range in Command mode first by typing `<Ctrl+V>` and using the arrow keys. Then type `:`. The word `that` will be substituted for the word `this` (vim, gvim only).
`:s/ \<tim\>/tom/`	Substitutes only the whole word `tim` with the word `tom`, not the partial match of `tim` in any string.
`:%s/terrible/wonderful/gc`	Interactive substitution using c option of the word `terrible` with the word `wonderful` (vim, gvim only).
`:%s/^/ \=line(".") . ". "/g`	Makes the line numbers of all lines in the buffer permanently part of each line (vim, gvim only).

`/option(s)` is a modifier, usually **`g`** for global, to the command; and

`count` is the number of lines to execute the command on from the current position.

The grammar of `pattern` and `string` can be extremely explicit and complex, and may take the form of a regular expression. (We present more information on the formation of regular expressions in Chapter 7, Section 7.2.) Some examples of the syntax for the `substitute` command, including vim/gvim-only constructions, are given in Table 3.7.

Practice Session 3.4 shows you how to use the vi commands `yank` and `put` to copy and paste. It also allows you to do individual and multiple searches and replace text with the vi `substitute` command.

3.2.5.1 Practice Session 3.4

Step 1: At the shell prompt, type **`vi multiline`** and then press **`<Enter>`**.

Step 2: Type **`A`** and then type **`Windows is the operating system of choice for everyone.`**

Step 3: Press the **`<Esc>`** key. You have left Insert mode and are now in Command mode.

Step 4: Press the **`0`** (zero) key. The cursor moves to the first character of the first line.

Step 5: Type **`yy`**. This action yanks, or copies, the first line to a special buffer.

Step 6: Type **`7p`**. This action puts, or pastes, the first line seven times, creating seven new lines of text containing the same text as the first line. The cursor should now be on the first character of the eighth line.

Step 7: Type **1G**. This action puts the cursor on the first character of the first line in the buffer.

Step 8: Hold down the **<Shift>** and **;** keys at the same time. Doing so places a **:** in the status line at the bottom of the vi screen display, allowing you to type a command.

Step 9: Type **s/everyone/students/** and then press **<Enter>**. The word `everyone` at the end of the first line is replaced with the word `students`.

Step 10: Use the arrow keys to position the cursor on the first character of the second line.

Step 11: Type **:s/everyone/computer scientists/** and then press **<Enter>**.

Step 12: Repeat Steps 8–10 on the third through eighth lines of the buffer, substituting the words `engineers`, `system administrators`, `web servers`, `scientists`, `networking`, and `mathematicians` for the word `everyone` on each of those six lines.

Step 13: Type **:1,$s/Windows/UNIX/g** and then press **<Enter>**. You have globally replaced the word `Windows` on all eight lines of the file with the word `UNIX`. Correct?

Step 14: Type **:wq**. You have now saved the changes and exited from vi.

3.2.6 vim and gvim

Vim and gvim are two examples among many of enhanced, "improved" versions of vi. The following subsections illustrate some of the advantages of using vim and gvim over the traditional vi editor.

3.2.6.1 Vim Enhancements

The following capabilities of vim that enhance vi functionality, particularly the first one shown, are suggestions that you can use to expedite your editing tasks with vim and gvim over and above the capabilities of vi:

* vimrc

If you want to enable any of the improved facilities of vim and gvim, you should create a **~/.vimrc** file. Even if this file is empty, it will enable the facilities that we illustrate in this section!

* Help

In vim Last Line mode, type `help` or press the `F1>` function key.

Vim opens a help buffer that gives you extensive help on its facilities. In Last Line mode, when in the help buffer, type `q` to exit help.

* Multiple Windows

The Last Line mode command `split` splits the current window in two. You can then move the cursor up to a window with `<Ctrl+W> j` and down a window with `<Ctrl+W> k`. For example, the Last Line mode command `split new.c` splits the window and begins editing the file named **new.c**. To close a window, use the normal vim exit commands `ZZ` or `:q!`.

* Multiple Levels of Undo

Unlike vi, you can use the undo command to undo several steps back in the command history. For example, typing `u` in Command mode undoes the last action in vim, and typing `3u` in Command mode undoes the last three actions you did in vim. The undo level is set by default to 1000. You can redo multiply as well, using `<Ctrl+R>`. For example, `3 <Ctrl+R>` redoes the last 3 actions that were undone with `u`.

* Visual Mode

Typing `v` causes vim to enter *Visual mode*. You can then highlight a block of text and execute a vim Command mode operation on it. The `v` command selects text by character. The `<Ctrl+V>` command selects text as a block. The `V` command selects the current line. See Section 3.2.6.2 for more details on this facility in vim.

* The `incsearch` and `hlsearch` Environmental Options (Incremental Search and Highlight Search)

For the incremental search, by default, searching starts after you enter the string. With the option:

```
:set incsearch
```

incremental searches will be done. The vim editor will start searching when you type the first character of the search string. As you type in more characters, the search is refined.

For the highlight search option, setting the option turns on search highlighting. This option is enabled by the command:

```
:set hlsearch
```

After the option is enabled, any search highlights the string matched by the search.

* The `cindent` Environmental Option and the = Command Option

Like vi autoindent, the vim editor does a more specific form of indentation. The `cindent` option is set with the command:

```
:set cindent
```

This turns on C style indentation. Each new line will be automatically indented the correct amount according to the C indentation standard.

* The `:make` Command

To compile a C program with an accompanying make file, and correct the errors, you can type this command in Last Line mode:

`make`

This runs the `make` command and captures the output. When the command finishes the editor starts editing the first file. The next step is to fix the error. After that you need to go to the line causing the next error. This is done using the command:

`cn`

This command will go to the location of the next error even if it is in another file.

You can continue fixing problems and using `cn` until all your problems are over or you want to do a recompile. If you want to see the current error message again, use the command:

`cc`

* Last Line Mode Command History

When you are in Last Line mode, you can use the `<Up>` arrow key to recall an older command line entry, and then can use the `<Down>` arrow key to go forward to newer commands. Then, when you press `<Enter>` after you have indexed to that previous command in the history, that previous command is executed again.

There are four histories you can utilize in vim, but the two most important ones are for:

- Last Line mode command history
- `/` and `?` search command history

Your search history is most useful to you, particularly because if you type complex search criteria, you do not want to have to retype them every time you want to repeat that search!

The two other histories are for expressions and input lines for the `input()` function.

As an example, you have done a Last Line mode command, typed five more Last Line mode commands, and then want to repeat the first command again. To do this, in Last Line mode press the `<Up>` arrow key five times. Another way of doing this is to type the first few letters of the Last Line mode command you want to return to.

The `<Up>` key will use the text typed so far and compare it with the lines in the history. Only matching lines will be used.

If you do not find the line you were looking for, use the `<Down>` arrow key to go back to what you typed and correct that. You can also type `<Ctrl+U>` to start all over again.

To see all the lines in your Last Line mode command history, while in Last Line mode, type:

`history`

You will then see a complete history of the Last Line mode commands for this session at the bottom of the screen display.

Your entire search history for this session is displayed by typing `history/` in Last Line mode.

`<Ctrl+P>` will work like the `<Up>` arrow key, except that it doesn't matter what you already typed. `<Ctrl+N>` works like the `<Down>` arrow key.

* The Last Line Mode Command Line Window

Typing any text in the Last Line mode command history to modify a previous command and then execute it is possible but difficult.

A better way to use a modified form of a Last Line mode command from the history is to open the *command line window* while in Command mode by typing:

`q:`

Vim now opens a small utility window at the bottom of the screen. It contains the command line history and an empty line at the end, similar to this illustration:

```
+-----------------------------------+
|other window |
|~ |
|first.c============================|
|:w second.c |
|:w third.c |
|:w fourth.c |
|:w fifth.c |
|:w sixth.c |
|:history
|: |
|command-line======================|
| |
~/project.c
+-----------------------------------+
```

In the buffer in this small utility window, you are in Insert mode, and can use Insert mode commands to modify text and also move commands. You can use the arrow keys to move around.

For example, move up the history tree with the `<Up>` arrow key to the `:w third.c` line.

Change the word `third` to `thirteenth`.

Now press <Enter> when on that line, and this command will be executed. The command line window will then close. The <Enter> command will execute the line under the cursor. This works if vim is in Insert mode or in Command mode.

Unfortunately, changes you make in the command line window are lost! They do not result in any changes in the command history itself, but the command you execute when you are in the command line window will be added at the end of the history, similar to all other executed commands. Also, only one command line window can be open at a time.

The command line window is very useful when you want to see your old command history, index to a particular command, edit it, and execute it.

A search command in your history can be used to find something new if you index to it and modify it. For example, if in the command line window one of the lines contained `:s/everyone/computer scientists/`, you could index to it in the command line window and modify and execute it.

* Word Completion

When you are typing and you enter a partial word, you can cause vim to search for a completion by using <Ctrl+P> (search for previous matching word) and <Ctrl+N> (search for next match).

* Record and Playback

The . (period) command repeats the previous change in Command mode. To accomplish multiple, complex changes in vim Command mode, you can use the *record and playback* facility. There are three steps in record and playback:

1. The `q(register)` command starts recording keystrokes into the key named `register`. The register name must be a letter of the alphabet.
2. Type the commands you want to record in the register.
3. To end recording, press `q`.

You can now execute the macro by typing the command `@register`. For example, you have a list of filenames in a buffer that looks like this:

```
stdio.h
fcntl.h
unistd.h
stdlib.h
```

And what you want is the following:

```
#include "stdio.h"
#include "fcntl.h"
```

```
#include "unistd.h"
#include "stdlib.h"
```

You start by moving to the first character of the first line. Next, in Command mode, you execute the following commands:

```
qa
^
i#include "<Esc>
$
a"<Esc>
j
q
```

These commands do the following:

1. Start recording a macro in register `a`.
2. Move to the beginning of the line.
3. Insert the string `#include "` at the beginning of the line.
4. Move to the end of the line.
5. Append the double quotation mark (`"`) character to the end of the line.
6. Go to the next line.
7. Stop recording the macro.

Now that you have done the work once, you can repeat the change by typing the command `"@a"` three times.

The `"@a"` command can be preceded by a count, which will cause the macro to be executed that number of times. In this case you would type: `"3@a"`.

EXERCISE 3.3

How would you open a unique history window for the `/` and `?` commands?

EXERCISE 3.4

Where does the cursor have to be positioned in the buffer if you want to execute a modified version of the substitute command `:s/everyone/computer scientists/` correctly?

EXERCISE 3.5

Can you include Last Line mode commands, such as `substitute`, or write to a file, in a record and playback session?

3.2.6.2 Vim Visual Mode

Because vi does not have a graphical, or "visual," method of selecting and operating on blocks of text, we use vim Visual mode. In vim, Visual mode is the graphical and easy way to select a block of text in order to use a prescribed operator on it. The following will briefly describe Visual mode's features and give a simple example. Vim Visual mode allows you to apply commands to blocks of text that can be selected graphically, even though you may not be in a GUI environment. In general, all of the vi commands and operating modes shown previously work in both vim and gvim.

Using Visual mode is done in three steps:

Step 1: Move the cursor to the start of the text block, mark the start of the block with `"v"` (character mode), `"V"` (line mode), or `<Ctrl+V>` (blockwise mode). The character under the cursor will be used as the start of the block.

Step 2: Depending on what kind of functionality is provided in the terminal or console window you are working in, move to the end of the text block, either with the arrow keys on the keyboard, with the `h`, `j`, `k`, or `l` keys on the keyboard, or with the mouse and mouse button(s). The text from the character where you start Visual mode, up to and including the character under the cursor, is highlighted. Generally `v` and `V` modes allow definition of nonrectangular blocks, whereas `<Ctrl+V>` allows definition of only rectangular blocks.

Step 3: Type a prescribed operator command. The highlighted characters from Step 2 will be operated upon depending on the nature of the prescribed operator listed.

You can use `<Esc>` to stop the definition of a block any time before you use a prescribed operator.

A simple example that illustrates how you can copy and paste using Visual mode follows:

Step 1: At the shell prompt type **`vim visualtest1`**, then press **`<Enter>`** on the keyboard.

Step 2: Type three or four arbitrary lines of text of uneven length (five to ten words each) into the buffer that opens on screen. Put some spaces at the beginning of some of the lines.

Step 3: Position the cursor on the first character of the first line of the buffer.

Step 4: Type **v**. On the last line display you will be notified that you have entered Visual mode! Now you can define the block that will be all or possibly only a portion of Step 2 text.

Step 5: Expand the highlighted area by using the input device of your choice until all the text you typed in Step 2 is highlighted. If you make a mistake in defining the block, use **<Esc>** to stop the block definition and begin highlighting again at the first character in the buffer until you get the block definition you desire.

Step 6: Type **y**.

Step 7: On the last line display you will see a report of how many lines you just yanked.

Step 8: Position the cursor anywhere on the last line of the buffer.

Step 9. Type **o**. A new line opens below the last line in the buffer. Press **<Esc>**.

Step 10: Type **p**. The yanked block from Step 6 is put back in the buffer, starting on the new line you opened in Step 9 and proceeding downward. Save the file if you want to.

3.2.6.3 Using Gvim to Cut and Paste between Multiple Open Buffers

To illustrate the speed and efficiency of using gvim as a modern graphical UNIX text editor, and to describe some of gvim's functions, the following practice session allows you use gvim to create text in two different files, open buffers into those files in two different windows, and copy and paste between those buffers.

In general, all of the vi commands and operating modes shown previously work in both vim and gvim.

3.2.6.4 Practice Session 3.5

Step 1: At the shell prompt, type **gvim gvim1** and then press the **<Enter>** key.

Step 2: Type A and then type **This is the first line of text.** Then press the **<Enter>** key twice.

Step 3: Type **This is the third line of text.** Then press the **<Enter>** key.

Step 4: From the gvim pull-down menu, make the choice **Window>Split**. You now are looking into two windows on the same buffer.

Step 5: From the gvim pull-down menu, make the choice **File>Save**. Click the OK button in the `Save` window. The buffer is saved to the file **gvim1**.

Step 6: Use the mouse and click anywhere in the lower window with the left mouse button. You are now working in the lower buffer.

Step 7: From the gvim pull-down menu, make the choice **File>Save As**. In the `Name` box, change the name of the file to **gvim2**, and then make the **Save** button choice. The buffer is saved as **gvim2**, and you are looking into the buffer through two windows.

Step 8: The active buffer is still seen in the lower window. Use the mouse and **<Delete>** key on the keyboard to change the word `first` to the word `second`, and the word `third` to the word `fourth` in the lower window.

Step 9: Click anywhere in the top window.

Step 10: Make the gvim pull-down menu choice **File>Open**. Scroll down and open **gvim1** in the current directory by selecting it and making the `OK` button choice. You should now be seeing **gvim1** in the upper window, and **gvim2** in the lower window.

Step 11: Click anywhere in the bottom window.

Step 12: Use the mouse and left mouse button to highlight the text `This is the second line of text.` Make sure the cursor is flashing on the period as you finish selecting that line.

Step 13: Make the gvim pull-down menu choice **Edit>Copy**. You have "yanked" a line of text in the lower buffer graphically.

Step 14: Click on the second blank line in the upper window buffer.

Step 15: Make the gvim pull-down menu choice **Edit>Paste**. The line `This is the second line of text.` is now on the second line of the upper window buffer. You can use the gvim pull-down menu choice **Edit>Undo** to correct mistakes in copying and in pasting.

Step 16: Repeat Steps 11 through 15 to copy and paste the line `This is the fourth line of text.` from the lower window buffer to the upper window buffer. When you are done, your screen display should look similar to Figure 3.4.

Step 17: While the active window buffer is the upper window, make the gvim pull-down menu choice **File>Save-Exit**.

Step 18: At the shell prompt, type **more gvim1**. What appears on screen? Do the same for the file **gvim2**. What appears on screen?

3.2.7 Changing vi, vim, and gvim Behavior

In general, all of the environment options commands shown in this section work in vi, vim, and gvim. Note that, because vim stands for *vi improved*, vim and gvim have many more environmental options. As previously suggested, you can create an empty version of the **~/.vimrc** file to enable many of the behavioral changes we show here. We also suggest you modify both your **~/.exrc** and **~/vimrc** files to accomplish the behavioral changes

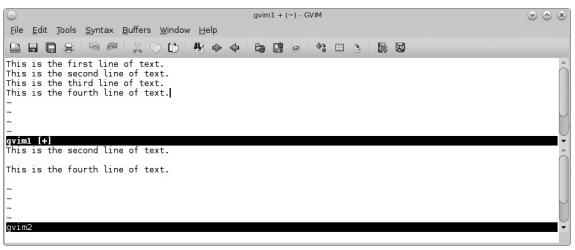

FIGURE 3.4 File **gvim1** after Step 16.

illustrated, depending on which editor you want the changes to be implemented in. If you put a *vim-specific behavior-changing option* in the **.exrc** file, when you run vi, you will probably get a warning message in vi, but not a fatal error message.

You can modify any of several environment options to customize the behavior of the vi, vim, and gvim editors, either when you are in the editor at a given time, or for every editor session. These options include, for example, specifying maximum line length and automatically wrapping the cursor to the next line, displaying line numbers as you edit a file, and displaying the mode that the editor is in. You can use full or abbreviated names for most of the options. Some of the most important and useful options and their abbreviations are summarized in Table 3.8. Also see Table 3.9 for a summary of the use of the `set` command.

The `set` command in Last Line mode changes environmental options. There are two types of environmental options that can be modified with the `set` command: toggle options, which are either "on" or "off," and options that require the use of an argument.

For example, after typing `:set showmode`, you have toggled the mode display "on," and the editor displays the current mode at the bottom of the screen. If you then type `:set noshowmode`, you have toggled the mode display "off." Similarly, after typing `:set nu`, vi displays the line numbers for all the lines in the file. To turn "off" the line number display, type `:set nonu`. When the `:set ai` command has been executed, the next line is aligned with the beginning of the previous line. This useful feature allows you to easily indent source codes that you compose with vi. Pressing `<Ctrl+D>` on a new line moves the cursor to the previous indentation level.

TABLE 3.8 Important Environmental Options for Vi, Vim, and Gvim

Option	Abbreviation	Purpose
`autoindent`	`ai`	Aligns the new line with the beginning of the previous line.
`ignorecase`	`ic`	Ignores the case of a letter during the search process (with a / or the ? command).
`list`	`list`	Displays invisible characters, such as `^I` for `<Tab>` and a `$` for end-of-line characters.
`nolist`	`nolist`	Turns off the display of invisible characters.
`noignorecase`	`noic`	Instructs cases to be case sensitive.
`number`	`nu`	Displays line numbers when a file is being edited; line numbers are not saved as part of the file.
`nonumber`	`nonu`	Hides line numbers.
`scroll`		Sets the number of lines to scroll when the `<Ctrl+D>` command is used to scroll the vi screen up.
`set`		Displays all the vi variables that are set.
`all`		Displays all set vi variables and their current values.
`showmode`	`smd`	Displays the current vi mode in the bottom right corner of the screen.
`noshowmode`	`nosmd`	Turns off the mode of operation display.
`wrapmargin`	`wm`	Sets the wrap margin in terms of the number of characters from the end of the line, assuming a line length of 80 characters.

TABLE 3.9 Last Line Mode Syntax

Last line mode syntax	What it does
abbr command	
`:ab in out`	Use `in` as abbreviation for `out` in Insert mode.
`:unab in`	Remove abbreviation for `in`.
`:ab`	List abbreviations.
map!, map commands	
`:map string sequence`	Map characters string as sequence of commands. Use #1, #2, etc., for the function keys.
`:unmap string`	Remove map for characters string.
`:map`	List character strings that are mapped.
`:map! string sequence`	Map characters string to input mode sequence.
`:unmap! string`	Remove input mode map (you may need to quote the characters with `<Ctrl+V>`).
`:map!`	List character strings that are mapped for input mode.
`qx`	Record typed characters into register specified by letter x (vim and gvim).
`q`	Stop recording (vim and gvim).
`@x`	Execute the register specified by letter `x`. Use @@ to repeat the last @ command.
set command	
`:set x`	Enable boolean option `x`, show value of other options.
`:set nox`	Disable option `x`.
`:set x=value`	Give value to option `x`.
`:set`	Show changed options.
`:set all`	Show all options.
`:set x?`	Show value of option `x`.

To see a listing of what all environment options in the editor are (the ones you have modified and the defaults) at any time, type `:set all`.

To see a listing of what environment options you have modified, either for this session only, or for all sessions, type `:set`.

When you use `set` to modify the environment options within an editor session, the options are set for that session only!

If you want to customize your environmental options for all vi, vim, and gvim sessions, you need to put your options in the **.exrc** file in your home directory. You can use the set command to modify one or more options in the **.exrc** file as follows (typing the two keyboard keys `<Ctrl+C>` terminates the creation of the cat command):

```
$ cat > .exrc
set wm=5 showmode nu ic
<Ctrl+C>
$
```

The `wm=5` option sets the wrap margin to 5, and is an example of a set command that requires an argument. That is, each line will be up to 75 characters long. The `ic` option allows you to search for strings without regard to the case of a character. Thus, after this option has been set, the `/Hello/` command searches for strings `hello` and `Hello`.

EXERCISE 3.6

After examining Tables 3.8 and 3.9, select a few of the environment options that most appeal to you and then place them in your **~/.exrc file**. Test them by running vi.

EXERCISE 3.7

If you haven't already done so, place the `set showmode` environment setting in your **~/.exrc** and in your **~/.vimrc** file. Run vim and then gvim. Do various operations in both those editors. Does the mode you are in appear in the mode line in both editors?

3.2.8 Executing Shell Commands from within vi, vim, and gvim

At times you will want to execute a shell command without quitting vi and then restarting it. You can do so in Command mode by preceding the command with `:!`. Thus, for example, typing `:! pwd` would display the pathname of your current directory, and typing `:! ls` would display the names of all the files in your current directory. After executing a shell command, the editor returns to Command mode.

3.2.9 vi, vim, and gvim Keyboard Macros

Vi, vim, and gvim offer a variety of *macro* facilities; a macro is a keystroke construction that uses one or more compact set keystrokes to represent another larger number of keystrokes that are substituted for the single or compact set. Macros are used in vi, vim, and gvim for the following reasons:

1. During Insert mode, to construct an abbreviation. For example, in a text file where you use often-repeated blocks of the same text.
2. In Command mode, vi, vim, and gvim commands can be associated or *mapped* to other keys, such as the function keys at the top of the keyboard.
3. Complex commands and their arguments can be triggered by a single keystroke or a shorter sequence of keystrokes.

Here is a brief summary description of the various vi, vim, and gvim macro operations, two of which are covered in the indicated subsections below:

Text abbreviation (Section 3.2.9.1), which operates in Insert mode. An abbreviation works only in vi, vim, and gvim Insert mode.

Keystroke mapping (Sections 3.2.9.2 and 3.2.9.3), which can operate either in Insert mode or in Command mode, and uses the `map!` and `map` Last Line mode commands.

Once defined, a `map!` sequence is triggered only in Insert mode, and a `map` sequence is triggered only in vi, vim, and gvim Command mode.

Text-buffer execution, which operates only in Command mode. Once text has been placed in any of the named text buffers, that text can be executed as if it were a sequence of vi, vim, and gvim commands.

In the following sections, we will describe and give examples of some of the vi, vim, and gvim macro facilities, and also give an additional example of a specialized vim macro feature that can be used in gvim as well. Table 3.9 summarizes the uses of the `abbr`, `map!`, and `map` commands.

3.2.9.1 Text Abbreviation Macros Used in Insert Mode

To save keystrokes while entering text, in Last Line mode, use the `abbr`(eviate) or just `ab` command.

It has the following general syntactic form:

```
:ab[br] [abbreviation abbreviated]
```

where:

`:` gets you into Last Line mode;

`[ ]` designates optional components;

`ab` or **`abbr`** is the command for creating an abbreviation;

`abbreviation` is a valid string of contiguous (no spaces allowed) characters; and

`abbreviated` is the substitute text you want to be placed in the buffer.

Text abbreviations can be canceled with the Last Line mode `unabbr` command, followed by typing the abbreviation you want to cancel. Also, if you just type `abbr` in Last Line mode, you get a listing of all the abbreviations that are active.

To use the abbreviation, when you are in Insert mode, whenever you type the string that represents `abbreviation` and precede as well as follow it by a nonalphanumeric character, the substitution will take place.

The editor will examine the character before and the next character after you type the `abbreviation` to see if it's nonalphanumeric or underscore, and if so, `abbreviation` will be erased and the string that represents `abbreviated` will be substituted for it. Also, you are no longer in Insert mode.

For example, in Last Line mode, if you type `ab kts Know this stuff!` and then press `<Enter>`, `kts` is the abbreviation. Then anywhere in Insert mode, when you type `sts` and precede it and follow it by pressing the space key, the left or right arrow keys (all of which yield nonalphanumeric characters and are not the underscore keys on the keyboard

for our system), the string `Know this stuff!` will be substituted on that line, and you will no longer be in Insert mode.

Note: With `abbr`, your text appears as you type it, and no substitution is performed on `abbreviation` until you type nonalphanumeric characters before it and after it. As shown in the next section, this is different from keystroke mapping using the `map!` or `map` commands.

The following are some useful abbreviations for Python program file creation.

```
:ab 1 #!/usr/local/bin/python
:ab 2 from Tkinter import *
:ab 3 import os
:ab 4 import sys
```

3.2.9.2 Keystroke-Mapping Macros Used in Insert Mode

`map`, shown in the next subsection, works on characters that are typed in Command mode, and `map!`, shown in this subsection, works on characters that are typed in Insert mode.

As shown in the previous section, `abbr` won't substitute text until you type a nonalphanumeric before and after the `abbreviation` string. Notice the editor echoes each character of the `abbreviation` as you type it, just in case you really want the string of characters that represents `abbreviation` to be an actual string of characters that you want in your text. Keystroke mapping works in a more keystroke- and time-dependent way. Keystroke mapping used in Insert mode is handled by the Last Line mode `map!` command, which takes the following general form:

```
:map! [substitution substituted]
```

where:

: gets you into Last Line mode;

[ ] designates optional components;

map is the command for creating a keyboard mapping;

substitution is a valid string of contiguous (no spaces allowed) characters; and

substituted is the substitute text you want to be placed in the buffer.

For example, in Last Line mode, if you type `map! ts This will save you time!` and then press `<Enter>`, `ts` is the `substitution`. Then, anywhere in Insert mode, when you type `ts` in a short amount of time (under approximately half a second), the string `This will save you time!` will be substituted on that line and you will still be in Insert mode. If you type more slowly, the literal string `ts` will be inserted.

The keystroke sequence <Ctrl+V> will let you escape the mapping, as long as you precede the macro with it. So no matter how fast you type in <Ctrl+V> ts, you get the literal string ts inserted.

Remapping abbreviations can be canceled with the Last Line mode unmap! command, followed by typing the substitution you want to cancel. Also, if you just type map! in Last Line mode, you get a listing of all the mappings that are active. You will see that the editor already has several mappings defined by default.

3.2.9.3 Keystroke-Remapping Macros Used in Command Mode

Command mode remapping is accomplished with the map Last Line mode command.

The general form of the map command is as follows.

```
:map [substitution substituted]
```

where:

: gets you into Last Line mode;

[ ] designates optional components;

map is the command for creating a keyboard mapping;

substitution is a valid string of contiguous(no spaces allowed) characters; and

substituted is the substitute text you want to be placed in the buffer.

Some editor command keys cannot be remapped in Command mode. Two examples of these keys are : (colon) and u.

Remapping substitutions can be canceled with the Last Line mode unmap command, followed by typing the remapping you want to cancel. Also, if you just type map in Last Line mode, you get a listing of all the mappings that are active. You will see that the editor already has several mappings defined by default.

As an example, in Last Line mode, if you type map #8: wq<Ctrl+V><Ctrl+Enter> and then press <Enter>, the function key F8> at the top of your keyboard is the substitution. The substituted is the command to write the buffer to a file and quit the editor. The <Ctrl+V> and <Ctrl+Enter> keystrokes are the way to enter control characters on the command line, in this case the <Enter> key at the end of the command. After this mapping is done, anytime you are in Command mode, when you press the function key <F8>, the buffer will be written to the default file and you will exit the editor.

Another interesting and useful example is the following map command, which can be placed in your **.exrc** file, so that you can use the function key <F3> during editor sessions to generate a skeleton C program construct:

```
map #3 ^[i#include stdio.h ^Mmain(argc, argv) ^M int argc;^M
  char #argv[];^Ml{^M}^M^[
```

where:

^[stands for pressing <Ctrl+V> and then <Esc>; and

^M stands for pressing <Ctrl+V> and then the <Enter> key

The relative number of spaces in this map command definition controls the indentation of the skeleton construct. Also, the ^M entries put each of the skeleton construct components on a new line.

3.2.9.4 Vim/Gvim Macro Example

Here is a repeat of Practice Session 3.4, slightly enlarged, that uses a vim-specific macro command sequence to accomplish the same thing that Practice Session 3.4 did, but in another way.

3.2.9.5 Practice Session 3.6

Step 1: From the shell prompt, type **vim unixos2** and then press **<Enter>** on the keyboard.

Step 2: In vim, type **A** and then type the following 10 lines of text, each on its own line:

```
computer scientists
students
hackers
systems analysts
newbies
UNIX gurus
computer programmers
systems administrators
network administrators
LINUX users
```

Step 3: Press **<Esc>**, then place the cursor anywhere on the first line of text.

Step 4: Type **q a**. This puts you in record mode and associates the macro you are about to record with the **a** key.

Step 5: Type **I**. The cursor is now at the start of the first line in Insert mode.

Step 6: Type **UNIX is the operating system of choice for** with a single space after the **r** in the word **for**. Press **<Esc>**.

Step 7: Place the cursor anywhere on the second line of text.

Step 8: Type **q**. This ends record mode.

Step 9: Type **a@9**. This "plays back" the macro defined with the **a** key nine times, once on each of the lines below the first line, inserting the text string `UNIX is the operating system of choice for`.

Step 10: Save the file, print it out, and memorize its contents.

3.3 THE EMACS EDITOR

The emacs editor is the most complex and customizable of the UNIX text editors, and it gives you the most freedom, flexibility, and control over the way you edit text files. It can format text for very specific technical applications, such as program source code development, more effectively than a word processor. Its use in that application makes the process of program development more efficient. In addition, from within the emacs program, in multiple windows, you can accomplish a wide variety of personal productivity and operating system tasks, such as sending e-mail and executing shell commands and scripts. But along with more control, specificity, and capabilities comes some additional learning in terms of a more complex keystroke command structure. This complexity can be offset in part for some users, and totally for others, by using the graphical forms of input and command execution that we will emphasize in the sections that follow.

To stress how the keyboard keys are used in GNU emacs, we repeat the following note shown at the beginning of this chapter here:

1. Pressing the Escape key is signified as `<Esc>`
2. Pressing the Enter key is signified as `<Enter>`
3. Pressing the `<Ctrl>` key in combination with another single key is signified as `<Ctrl+X>`, where you hold down the `<Ctrl>` key and press the `X` key (or any valid key for that combination) at the same time.
4. Pressing the `<Alt>` key in combination with another single key is signified as `<Alt+X>`,

   where you hold down the `<Alt>` key and press the `X` key (or any valid key for that combination) at the same time.
5. A variant of 3. and 4. is shown as `<Ctrl+X> a [b]`, where you first press and release `<Ctrl>` and `X` simultaneously, and then press the `a` key, and optionally press the `b` key (or any valid combination of single keys or strings of characters.
6. In GNU emacs for PC-BSD and Solaris, the `Meta` key that is referred to in much of the literature on GNU emacs, is the `<Alt>` key.

It is important to realize before you begin that there are some common terms used in vi, vim, and gvim (the editors from the previous section) and emacs that describe the facilities of each editor, but the terms do not have the same meaning between the two major families of editor.

As you saw in Section 3.1.2, with vi, vim, and gvim, you can't immediately begin to enter text into the file you are editing. You have to be in Insert mode to do that, that's what typing `A` as the second step is doing. Vi, vim, and gvim have modes. In GNU emacs, you can start typing text into the file immediately. Emacs is a *modeless* editor.

Vi, vim, and gvim operate in three distinct modes: Command mode, Insert mode, and Last Line mode. Emacs is a *modeless* editor in the sense that, when you launch emacs, you do not have to switch modes to immediately type characters on the keyboard and enter text into a buffer, or change modes to save the buffer to a file.

For example, emacs does have major modes of operation, such as *Lisp mode*, *Python mode*, and *C mode*, but they are for the special formatting of text and for specialized operations when editing files for use in those language applications. This is different from allowing you to switch between significant forms of action in the editor, as the vi, vim, and gvim Command, Insert, and Last Line modes do, as seen in the previous section. The keystroke command syntax itself in emacs is different and more complex than in vi, involving use of the `<Ctrl>` and `<Alt>` prefix characters, as previously noted. The emacs concepts of *point* and the cursor location are also more refined and specific than in vi. In emacs, the point is the location in the buffer where you are currently doing your editing; the point is assumed to be at the left edge of the cursor, or always between characters or white space (what you enter into a text file when you press the space bar). This difference becomes an important issue when you want to use the cut/copy/paste operations. In vi, yanking removes text from the main buffer, much like cutting/copying, whereas in emacs yanking is more like pasting into the main buffer. The concept of a buffer is very important in emacs, and is very much the same in emacs as it is in vi.

Currently, there is one major "brand" of emacs for UNIX: GNU emacs. We use the graphical form of GNU emacs version 24.3.1 in both PC-BSD and Solaris, running in its own frame, in the following illustrations, exercises, practice sessions, and problems.

If you cannot run a graphical emacs because you are working in a text-only console or terminal, you can still gain access to the Menu Bar at the top of the emacs screen by pressing `<Esc>` on the keyboard and then pressing the single back quote (`` ` ``) key. You can then descend through the menu bar choices by pressing the letter key of the menu choice you want to make. For example, pressing the `f` key gives you access to the File pull-down menu choices, and then pressing the `s` key allows you to save the current buffer. Unfortunately, you cannot access the speed button bar menu choices from within a text-only display of emacs. A summary of emacs commands is given in Table 3.10.

3.3.1 Launching Emacs, Emacs Screen Display, General Emacs Concepts and Features

The general syntax for launching the emacs program from the command line in a console window is as follows (anything enclosed in square brackets `[ ]` is optional):

TABLE 3.10 Summary of emacs Commands

Command	Action
<Ctrl+X> <Ctrl+F>	Visit a file (find-file)
<Ctrl+X> <Ctrl+R>	Visit a file for viewing, without allowing changes to it (find-file-read-only)
<Ctrl+X> <Ctrl+V>	Visit a different file instead of the one visited last (find-alternate-file)
<Ctrl+X> 4 f	Visit a file, in another window (find-file-other-window)
<Ctrl+X> 5 f	Visit a file, in a new frame (find-file-other-frame)
<Alt+X> find-file-literally	Visit a file with no conversion of the contents (shows you control characters, etc.)
<Ctrl+X> <Ctrl+S>	Save the current buffer to its file (save-buffer)
<Ctrl+X> s	Save any or all buffers to their files (save-some-buffers)
<Alt+~>	Forget that the current buffer has been changed (not-modified)
<Ctrl+X> <Ctrl+W>	Save the current buffer with a specified file name (write-file)
<Alt+X> set-visited-file-name	Change the file name under which the current buffer will be saved
<Ctrl+R>	View the buffer that you are currently being asked about
<Ctrl+H>	Display a help message about these options
<Ctrl+X> <Ctrl+C>	Exits emacs
<Ctrl+X> <Ctrl+Z>	Suspends emacs and exits to the shell

SYNTAX

emacs [options][file(s)]

Purpose: Allows you to edit a new or existing file(s)

Output: With no options or file(s) specified, emacs runs and begins or opens on the Welcome Screen buffer

Commonly used options/features:

+n	Begin to edit file(s) starting at line number **n**
-nw	Run emacs without opening a window, useful in an elementary GUI environment
emacs file1 file2 file3	Open three buffers in emacs on three different files at the same time

For example, if you run the emacs program by typing `emacs alien` in a terminal window in our base PC-BSD system, emacs launches and shows the Welcome Screen buffer.

To close the Welcome Screen buffer display that opens in the bottom window of the emacs frame, while the cursor is flashing in the top window, make the emacs pull-down

menu choice **Remove Other Windows**. Or you can type `<Ctrl+X> 1`. In either case, you will only have one buffer shown in the screen display, similar to Figure 3.5.

A brief description of the major components of the emacs screen display labeled in Figure 3.5 is as follows (note: items J, A, B, D, and C are found on what is called the *mode line*):

A. Name of the current buffer: This is the name of the entity or "file" you are editing in this window. In Figure 3.5, the name of the buffer is **alien**.

B. Major and minor mode: Different major modes are used to edit different kinds of files, like C programs, Lisp, or HTML, and special configurations of the major modes define the minor modes. In Figure 3.5, only the major mode `Fundamental` is shown, with no minor mode set.

C. Percentage of the text shown on screen: This shows how much of the text in the buffer is seen on screen. In Figure 3.5, `All` of the text in the current buffer is shown on screen.

D. Current line number: The line location of the cursor in the current buffer is displayed here.

E. Minibuffer: Information and questions/prompts from emacs appear here. In Figure 3.5, `Wrote /usr/home/bob .emacs` is shown on screen, because the help file initial screen display buffer was closed.

F. Speed button bar: This allows you to do quick, common operations graphically.

G. Menu bar: This gives you pull-down menus that contain all of the important emacs operations.

H. Text: The actual text you are editing appears here.

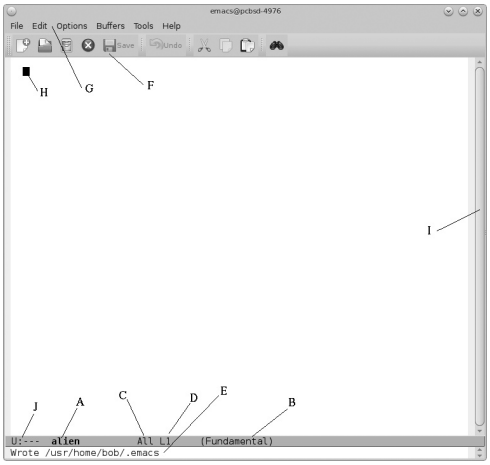

FIGURE 3.5 First GNU emacs screen display.

I. Scroll bar: The scroll bar allows you to graphically scroll or move through the text.

J. Status indicator: Two-character codes are used to tell you about your file. In Figure 3.5, a `U-` and two hyphens (`U---`) indicate that the file has not changed in emacs and is the same as the version saved to disk, and that you can work on the file.

3.3.1.1 Emacs Help

Emacs provides a wide variety of help commands, all accessible through the key sequence `<Ctrl+H>` or graphically with the function key `<F1>`. You can also type `<Ctrl+H> <Ctrl+H>` to view a list of help commands. You can scroll the list with `<Space>` and `<Del>`, then type the help command you want. To cancel, type `<Ctrl+G>`. Many help commands display their information in a special help buffer. In this buffer, you can type `<Space>` and `<Del>` to scroll and press `<Enter>` to follow hyperlinks.

The following are the most general ways of obtaining help on a topic or command:

`<Ctrl+H> a topic(s) <Enter>`

This searches for commands whose names match the argument **`topic(s)`**. The argument can be a keyword, a list of keywords, or a regular expression.

`<Ctrl+H> i d m emacs <Enter> i topic <Enter>`

This searches for **topic** in the indices of the emacs Info manual, displaying the first match found. Press **,** (comma) to see subsequent matches. You can use a regular expression as a topic.

`<Ctrl+H> i d m emacs <Enter> s topic <Enter>`

Similar, but searches the text of the manual rather than the indices.

`<Ctrl+H> <Ctrl+F>`

This displays the emacs FAQ, using Info.

`<Ctrl+H> p`

This displays the available emacs packages based on keywords.

A summary of help command syntax is found in Table 3.11.

3.3.1.2 Graphical Features

The most useful graphical features of emacs are the menu bar and speed button bar seen in Figure 3.5 as F and G. These features incorporate all of emacs's functionality into a