Operating System Concepts

TENTH EDITION

ABRAHAM SILBERSCHATZ • PETER BAER GALVIN • GREG GAGNE

WILEY

OPERATING SYSTEM CONCEPTS

OPERAT

ING

SYSTEM

CONCEP

TS

OPERATING
SYSTEM
CONCEPTS

OPERAT
ING
SYSTEM

CONCE PTS

ABRAHAM SILBERSCHATZ

PETER BAER GALVIN

GREG GAGNE

Publisher Laurie Rosatone
Editorial Director Don Fowley
Development Editor Ryann Dannelly
Freelance Developmental Editor Chris Nelson/Factotum
Executive Marketing Manager Glenn Wilson
Senior Content Manage Valerie Zaborski
Senior Production Editor Ken Santor
Media Specialist Ashley Patterson
Editorial Assistant Anna Pham

Cover Designer Tom Nery
Cover art © metha189/Shutterstock

This book was set in Palatino by the author using LaTeX and printed and bound by LSC Kendallville. The cover was printed by LSC Kendallville.

Library of Congress Cataloging-in-Publication Data

Names: Silberschatz, Abraham, author. | Galvin, Peter B., author. | Gagne, Greg, author.
Title: Operating system concepts / Abraham Silberschatz, Yale University, Peter Baer Galvin, Pluribus Networks, Greg Gagne, Westminster College.
Description: 10th edition. | Hoboken, NJ : Wiley, [2018] | Includes bibliographical references and index. |
Identifiers: LCCN 2017043464 (print) | LCCN 2017045986 (ebook) | ISBN 9781119320913 (enhanced ePub)
Subjects: LCSH: Operating systems (Computers)
Classification: LCC QA76.76.O63 (ebook) | LCC QA76.76.O63 S55825 2018 (print) | DDC 005.4/3--dc23
LC record available at https://lccn.loc.gov/2017043464

The inside back cover will contain printing identification and country of origin if omitted from this page. In addition, if the ISBN on the back cover differs from the ISBN on this page, the one on the back cover is correct.

Enhanced ePub ISBN 978-1-119-32091-3

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To my children, Lemor, Sivan, and
Aaron and my Nicolette

Avi Silberschatz

To my wife, Carla,
and my children, Gwen, Owen, and Maddie

Peter Baer Galvin

To my wife, Pat,
and our sons, Tom and Jay

Greg Gagne

Preface

Operating systems are an essential part of any computer system. Similarly, a course on operating systems is an essential part of any computer science education. This field is undergoing rapid change, as computers are now prevalent in virtually every arena of day-to-day life— from embedded devices in auto mobiles through the most sophisticated planning tools for governments and multinational firms. Yet the fundamental concepts remain fairly clear, and it is on these that we base this book.

We wrote this book as a text for an introductory course in operating systems at the junior or senior undergraduate level or at the first-year graduate level. We hope that practitioners will also find it useful. It provides a clear description of the *concepts* that underlie operating systems. As prerequisites, we assume that the reader is familiar with basic data structures, computer organization, and a high-level language, such as C or Java. The hardware topics required for an understanding of operating systems are covered in Chapter 1. In that chapter, we also include an overview of the fundamental data structures that are prevalent in most operating systems. For code examples, we use pre dominantly C, as well as a significant amount of Java, but the reader can still understand the algorithms without a thorough knowledge of these languages.

Concepts are presented using intuitive descriptions. Important theoretical results are covered, but formal proofs are largely omitted. The bibliographical notes at the end of each chapter contain pointers to research papers in which results were first presented and proved, as well as references to recent material for further reading. In place of proofs, figures and examples are used to suggest why we should expect the result in question to be true.

The fundamental concepts and algorithms covered in the book are often based on those used in both open-source and commercial operating systems. Our aim is to present these concepts and algorithms in a general setting that is not tied to one particular operating system. However, we present a large number of examples that pertain to the most popular and the most innovative operating systems, including Linux, Microsoft Windows, Apple macOS (the original name, OS X, was changed in 2016 to match the naming scheme of other Apple products), and Solaris. We also include examples of both Android and iOS, currently the two dominant mobile operating systems.

The organization of the text reflects our many years of teaching courses on operating systems. Consideration was also given to the feedback provided by the reviewers of the text, along with the many comments and suggestions we received from readers of our previous editions and from our current and former students. This Tenth Edition also reflects most of the curriculum guide lines in the operating-systems area in *Computer Science Curricula 2013*, the most recent curriculum guidelines for undergraduate degree programs in computer science published by the IEEE Computing Society and the Association for Com puting Machinery (ACM).

What's New in This Edition

For the Tenth Edition, we focused on revisions and enhancements aimed at lowering costs to the students, better engaging them in the learning process, and providing increased support for instructors.

According to the publishing industry's most trusted market research firm, Outsell, 2015 represented a turning point in text usage: for the first time, student preference for digital learning materials was higher than for print, and the increase in preference for digital has been accelerating since.

While print remains important for many students as a pedagogical tool, the Tenth Edition is being delivered in forms that emphasize support for learning from digital materials. All forms we are providing dramatically reduce the cost to students compared to the Ninth Edition. These forms are:

- **Stand-alone e-text now with significan enhancements**. The e-text format for the Tenth Edition adds exercises with solutions at the ends of main sections, hide/reveal definitions for key terms, and a number of animated figures. It also includes additional "Practice Exercises" with solutions for each chapter, extra exercises, programming problems and projects, "Fur ther Reading" sections, a complete glossary, and four appendices for legacy operating systems.

- **E-text with print companion bundle**. For a nominal additional cost, the e-text also is available with an abridged print companion that includes a loose-leaf copy of the main chapter text, end-of-chapter "Practice Exer cises" (solutions available online), and "Further Reading" sections. Instruc tors may also order bound print companions for the bundled package by contacting their Wiley account representative.

Although we highly encourage all instructors and students to take advantage of the cost, content, and learning advantages of the e-text edition, it is possible for instructors to work with their Wiley Account Manager to create a custom print edition.

To explore these options further or to discuss other options, contact your Wiley account manager (http://www.wiley.com/go/whosmyrep) or visit the product information page for this text on wiley.com

Book Material

The book consists of 21 chapters and 4 appendices. Each chapter and appendix contains the text, as well as the following enhancements:

- A set of practice exercises, including solutions
- A set of regular exercises
- A set of programming problems

- A set of programming projects
- A Further Reading section
- Pop-up definitions of important (blue) terms
- A glossary of important terms
- Animations that describe specific key concepts

A hard copy of the text is available in book stores and online. That version has the same text chapters as the electronic version. It does not, however, include the appendices, the regular exercises, the solutions to the practice exercises, the programming problems, the programming projects, and some of the other enhancements found in this ePub electronic book.

Content of This Book

The text is organized in ten major parts:

- **Overview**. Chapters 1 and 2 explain what operating systems are, what they do, and how they are designed and constructed. These chapters discuss what the common features of an operating system are and what an operating system does for the user. We include coverage of both traditional PC and server operating systems and operating systems for mobile devices. The presentation is motivational and explanatory in nature. We have avoided a discussion of how things are done internally in these chap ters. Therefore, they are suitable for individual readers or for students in lower-level classes who want to learn what an operating system is without getting into the details of the internal algorithms.

- **Process management**. Chapters 3 through 5 describe the process concept and concurrency as the heart of modern operating systems. A ***process*** is the unit of work in a system. Such a system consists of a collection of ***concurrently*** executing processes, some executing operating-system code and others executing user code. These chapters cover methods for process scheduling and interprocess communication. Also included is a detailed discussion of threads, as well as an examination of issues related to multi core systems and parallel programming.

- **Process synchronization**. Chapters 6 through 8 cover methods for process synchronization and deadlock handling. Because we have increased the coverage of process synchronization, we have divided the former Chapter 5 (Process Synchronization) into two separate chapters: Chapter 6, Syn chronization Tools, and Chapter 7, Synchronization Examples.

- **Memory management**. Chapters 9 and 10 deal with the management of main memory during the execution of a process. To improve both the utilization of the CPU and the speed of its response to its users, the computer must keep several processes in memory. There are many different memory-management schemes, reflecting various approaches to memory management, and the effectiveness of a particular algorithm depends on the situation.

- **Storage management**. Chapters 11 and 12 describe how mass storage and I/O are handled in a modern computer system. The I/O devices that attach to a computer vary widely, and the operating system needs to provide a wide range of functionality to applications to allow them to control all aspects of these devices. We discuss system I/O in depth, including I/O system design, interfaces, and internal system structures and functions. In many ways, I/O devices are the slowest major components of the com puter. Because they represent a performance bottleneck, we also examine performance issues associated with I/O devices.

- **File systems**. Chapters 13 through 15 discuss how file systems are handled in a modern computer system. File systems provide the mechanism for on line storage of and access to both data and programs. We describe the clas sic internal algorithms and structures of storage management and provide a firm practical understanding of the algorithms used— their properties, advantages, and disadvantages.

- **Security and protection**. Chapters 16 and 17 discuss the mechanisms necessary for the security and protection of computer systems. The processes in an operating system must be protected from one another's activities. To provide such protection, we must ensure that only processes that have gained proper authorization from the operating system can operate on the files, memory, CPU, and other resources of the system. Protection is a mechanism for controlling the access of programs, processes, or users to computer-system resources. This mechanism must provide a means of specifying the controls to be imposed, as well as a means of enforce ment. Security protects the integrity of the information stored in the system (both data and code), as well as the physical resources of the system, from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency.

- **Advanced topics**. Chapters 18 and 19 discuss virtual machines and networks/distributed systems. Chapter 18 provides an overview of virtual machines and their relationship to contemporary operating systems. Included is a general description of the hardware and software techniques that make virtualization possible. Chapter 19 provides an overview of computer networks and distributed systems, with a focus on the Internet and TCP/IP.

- **Case studies**. Chapter 20 and 21 present detailed case studies of two real operating systems—Linux and Windows 10.

- **Appendices**. Appendix A discusses several old influential operating systems that are no longer in use. Appendices B through D cover in great detaisl three older operating systems— Windows 7, BSD, and Mach.

Programming Environments

The text provides several example programs written in C and Java. These programs are intended to run in the following programming environments:

- **POSIX**. POSIX (which stands for *Portable Operating System Interface*) represents a set of standards implemented primarily for UNIX-based operating systems. Although Windows systems can also run certain POSIX programs, our coverage of POSIX focuses on Linux and UNIX systems. POSIX compliant systems must implement the POSIX core standard (POSIX.1); Linux and macOS are examples of POSIX-compliant systems. POSIX also defines several extensions to the standards, including real-time extensions (POSIX.1b) and an extension for a threads library (POSIX.1c, better known as Pthreads). We provide several programming examples written in C illustrating the POSIX base API, as well as Pthreads and the extensions for real-time programming. These example programs were tested on Linux 4.4 and macOS 10.11 systems using the gcc compiler.

- **Java**. Java is a widely used programming language with a rich API and built-in language support for concurrent and parallel programming. Java programs run on any operating system supporting a Java virtual machine (or JVM). We illustrate various operating-system and networking concepts with Java programs tested using Version 1.8 of the Java Development Kit (JDK).

- **Windows systems**. The primary programming environment for Windows systems is the Windows API, which provides a comprehensive set of functions for managing processes, threads, memory, and peripheral devices. We supply a modest number of C programs illustrating the use of this API. Programs were tested on a system running Windows 10.

We have chosen these three programming environments because we believe that they best represent the two most popular operating-system models—Linux/UNIX and Windows—along with the widely used Java environment. Most programming examples are written in C, and we expect readers to be comfortable with this language. Readers familiar with both the C and Java languages should easily understand most programs provided in this text.

In some instances—such as thread creation—we illustrate a specific concept using all three programming environments, allowing the reader to contrast the three different libraries as they address the same task. In other situations, we may use just one of the APIs to demonstrate a concept. For example, we illustrate shared memory using just the POSIX API; socket programming in TCP/IP is highlighted using the Java API.

Linux Virtual Machine

To help students gain a better understanding of the Linux system, we provide a Linux virtual machine running the Ubuntu distribution with this text. The virtual machine, which is available for download from the text website

(http://www.os-book.com), also provides development environments includ ing the gcc and Java compilers. Most of the programming assignments in the book can be completed using this virtual machine, with the exception of assign ments that require the Windows API. The virtual machine can be installed and run on any host operating system that can run the VirtualBox virtualization software, which currently includes Windows 10 Linux, and macOS.

The Tenth Edition

As we wrote this Tenth Edition of *Operating System Concepts*, we were guided by the sustained growth in four fundamental areas that affect operating systems:

1. Mobile operating systems
2. Multicore systems
3. Virtualization

4. Nonvolatile memory secondary storage

To emphasize these topics, we have integrated relevant coverage throughout this new edition. For example, we have greatly increased our coverage of the Android and iOS mobile operating systems, as well as our coverage of the ARMv8 architecture that dominates mobile devices. We have also increased our coverage of multicore systems, including increased coverage of APIs that provide support for concurrency and parallelism. Nonvolatile memory devices like SSDs are now treated as the equals of hard-disk drives in the chapters that discuss I/O, mass storage, and file systems.

Several of our readers have expressed support for an increase in Java coverage, and we have provided additional Java examples throughout this edition.

Additionally, we have rewritten material in almost every chapter by bring ing older material up to date and removing material that is no longer interest ing or relevant. We have reordered many chapters and have, in some instances, moved sections from one chapter to another. We have also greatly revised the artwork, creating several new figures as well as modifying many existing figures.

Major Changes

The Tenth Edition update encompasses much more material than previous updates, in terms of both content and new supporting material. Next, we provide a brief outline of the major content changes in each chapter:

- **Chapter 1: Introduction** includes updated coverage of multicore systems, as well as new coverage of NUMA systems and Hadoop clusters. Old material has been updated, and new motivation has been added for the study of operating systems.

- **Chapter 2: Operating-System Structures** provides a significantly revised discussion of the design and implementation of operating systems. We have updated our treatment of Android and iOS and have revised our

coverage of the system boot process with a focus on GRUB for Linux systems. New coverage of the Windows subsystem for Linux is included as well. We have added new sections on linkers and loaders, and we now discuss why applications are often operating-system specific. Finally, we have added a discussion of the BCC debugging toolset.

- **Chapter 3: Processes** simplifies the discussion of scheduling so that it now includes only CPU scheduling issues. New coverage describes the memory layout of a C program, the Android process hierarchy, Mach message passing, and Android RPCs. We have also replaced coverage of the traditional UNIX/Linux init process with coverage of systemd.

- **Chapter 4: Threads and Concurrency** (previously Threads) increases the coverage of support for concurrent and parallel programming at the API and library level. We have revised the section on Java threads so that it now includes futures and have updated the coverage of Apple's Grand Central Dispatch so that it now includes Swift. New sections discuss fork join parallelism using the fork-join framework in Java, as well as Intel thread building blocks.

- **Chapter 5: CPU Scheduling** (previously Chapter 6) revises the coverage of multilevel queue and multicore processing scheduling. We have integrated coverage of NUMA-aware scheduling issues throughout, including how this scheduling affects load balancing. We also discuss related modifica tions to the Linux CFS scheduler. New coverage combines discussions of round-robin and priority scheduling, heterogeneous multiprocessing, and Windows 10 scheduling.

- **Chapter 6: Synchronization Tools** (previously part of Chapter 5, Process Synchronization) focuses on various tools for synchronizing processes. Significant new coverage discusses architectural issues such as instruction reordering and delayed writes to buffers. The chapter also introduces lock free algorithms using compare-and-swap (CAS) instructions. No specific APIs are presented; rather, the chapter provides an introduction to race conditions and general tools that can be used to prevent data races. Details include new coverage of memory models, memory barriers, and liveness issues.

- **Chapter 7: Synchronization Examples** (previously part of Chapter 5, Process Synchronization) introduces classical synchronization problems and discusses specific API support for designing solutions that solve these problems. The chapter includes new coverage of POSIX named and unnamed semaphores, as well as condition variables. A new section on Java synchronization is included as well.

- **Chapter 8: Deadlocks** (previously Chapter 7) provides minor updates, including a new section on livelock and a discussion of deadlock as an example of a liveness hazard. The chapter includes new coverage of the Linux lockdep and the BCC deadlock detector tools, as well as coverage of Java deadlock detection using thread dumps.

- **Chapter 9: Main Memory** (previously Chapter 8) includes several revisions that bring the chapter up to date with respect to memory management on modern computer systems. We have added new coverage of the ARMv8 64-bit architecture, updated the coverage of dynamic link libraries, and changed swapping coverage so that it now focuses on swapping pages rather than processes. We have also eliminated coverage of segmentation.

- **Chapter 10: Virtual Memory** (previously Chapter 9) contains several revisions, including updated coverage of memory allocation on NUMAsystems and global allocation using a free-frame list. New coverage includes com pressed memory, major/minor page faults, and memory management in Linux and Windows 10.

- **Chapter 11: Mass-Storage Structure** (previously Chapter 10) adds coverage of nonvolatile memory devices, such as flash and solid-state disks. Hard-drive scheduling is simplified to show only currently used algorithms. Also included are a new section on cloud storage, updated RAID coverage, and a new discussion of object storage.

- **Chapter 12, I/O** (previously Chapter 13) updates the coverage of technologies and performance numbers, expands the coverage of synchronous/asynchronous and blocking/nonblocking I/O, and adds a section on vectored I/O. It also expands coverage of power management for mobile operating systems.

- **Chapter 13: File-System Interface** (previously Chapter 11) has been updated with information about current technologies. In particular, the coverage of directory structures has been improved, and the coverage of protection has been updated. The memory-mapped files section has been expanded, and a Windows API example has been added to the discussion of shared memory. The ordering of topics is refactored in Chapter 13 and 14.

- **Chapter 14: File-System Implementation** (previously Chapter 12) has been updated with coverage of current technologies. The chapter now includes discussions of TRIM and the Apple File System. In addition, the discussion of performance has been updated, and the coverage of journaling has been expanded.

- **Chapter 15: File System Internals** is new and contains updated information from previous Chapters 11 and 12.

- **Chapter 16: Security** (previously Chapter 15) now precedes the protection chapter. It includes revised and updated terms for current security threats and solutions, including ransomware and remote access tools. The principle of least privilege is emphasized. Coverage of code-injection vul nerabilities and attacks has been revised and now includes code samples. Discussion of encryption technologies has been updated to focus on the technologies currently used. Coverage of authentication (by passwords and other methods) has been updated and expanded with helpful hints. Additions include a discussion of address-space layout randomization and a new summary of security defenses. The Windows 7 example has been updated to Windows 10.

- **Chapter 17: Protection** (previously Chapter 14) contains major changes. The discussion of protection rings and layers has been updated and now

refers to the Bell–LaPadula model and explores the ARM model of Trust Zones and Secure Monitor Calls. Coverage of the need-to-know principle has been expanded, as has coverage of mandatory access control. Subsections on Linux capabilities, Darwin entitlements, security integrity protection, system-call filtering, sandboxing, and code signing have been added. Coverage of run-time-based enforcement in Java has also been added, including the stack inspection technique.

- **Chapter 18: Virtual Machines** (previously Chapter 16) includes added details about hardware assistance technologies. Also expanded is the topic of application containment, now including containers, zones, docker, and Kubernetes. A new section discusses ongoing virtualization research, including unikernels, library operating systems, partitioning hypervisors, and separation hypervisors.

- **Chapter 19, Networks and Distributed Systems** (previously Chapter 17) has been substantially updated and now combines coverage of computer networks and distributed systems. The material has been revised to bring it up to date with respect to contemporary computer networks and distributed systems. The TCP/IP model receives added emphasis, and a discussion of cloud storage has been added. The section on network topologies has been removed. Coverage of name resolution has been expanded and a Java example added. The chapter also includes new coverage of dis tributed file systems, including MapReduce on top of Google file system, Hadoop, GPFS, and Lustre.

- **Chapter 20: The Linux System** (previously Chapter 18) has been updated to cover the Linux 4.*i* kernel.

- **Chapter 21: The Windows 10 System** is a new chapter that covers the internals of Windows 10.
- **Appendix A: Influentia Operating Systems** has been updated to include material from chapters that are no longer covered in the text.

Supporting Website

When you visit the website supporting this text at http://www.os-book.com, you can download the following resources:

- Linux virtual machine
- C and Java source code
- The complete set of figures and illustrations
- FreeBSD, Mach, and Windows 7 case studies
- Errata
- Bibliography

Notes to Instructors

On the website for this text, we provide several sample syllabi that suggest var ious approaches for using the text in both introductory and advanced courses.

As a general rule, we encourage instructors to progress sequentially through the chapters, as this strategy provides the most thorough study of operat ing systems. However, by using the sample syllabi, an instructor can select a different ordering of chapters (or subsections of chapters).

In this edition, we have added many new written exercises and pro gramming problems and projects. Most of the new programming assignments involve processes, threads, process scheduling, process synchronization, and memory management. Some involve adding kernel modules to the Linux sys tem, which requires using either the Linux virtual machine that accompanies this text or another suitable Linux distribution.

Solutions to written exercises and programming assignments are avail able to instructors who have adopted this text for their operating-system class. To obtain these restricted supplements, contact your local John Wiley & Sons sales representative. You can find your Wiley representative by going to http://www.wiley.com/college and clicking “Who’s my rep?”

Notes to Students

We encourage you to take advantage of the practice exercises that appear at the end of each chapter. We also encourage you to read through the study guide, which was prepared by one of our students. Finally, for students who are unfa miliar with UNIX and Linux systems, we recommend that you download and install the Linux virtual machine that we include on the supporting website. Not only will this provide you with a new computing experience, but the open source nature of Linux will allow you to easily examine the inner details of this popular operating system. We wish you the very best of luck in your study of operating systems!

Contacting Us

We have endeavored to eliminate typos, bugs, and the like from the text. But, as in new releases of software, bugs almost surely remain. An up-to-date errata list is accessible from the book's website. We would be grateful if you would notify us of any errors or omissions in the book that are not on the current list of errata.

We would be glad to receive suggestions on improvements to the book. We also welcome any contributions to the book website that could be of use to other readers, such as programming exercises, project suggestions, on-line labs and tutorials, and teaching tips. E-mail should be addressed to os-book authors@cs.yale.edu.

Acknowledgments

Many people have helped us with this Tenth Edition, as well as with the previous nine editions from which it is derived.

Tenth Edition

- Rick Farrow provided expert advice as a technical editor.
- Jonathan Levin helped out with coverage of mobile systems, protection, and security.
- Alex Ionescu updated the previous Windows 7 chapter to provide Chapter 21: Windows 10.
- Sarah Diesburg revised Chapter 19: Networks and Distributed Systems.
- Brendan Gregg provided guidance on the BCC toolset.
- Richard Stallman (RMS) supplied feedback on the description of free and open-source software.
- Robert Love provided updates to Chapter 20: The Linux System.
- Michael Shapiro helped with storage and I/O technology details.
- Richard West provided insight on areas of virtualization research.
- Clay Breshears helped with coverage of Intel thread-building blocks.
- Gerry Howser gave feedback on motivating the study of operating systems and also tried out new material in his class.
- Judi Paige helped with generating figures and presentation of slides. •

Jay Gagne and Audra Rissmeyer prepared new artwork for this edition. •

Owen Galvin provided technical editing for Chapter 11 and Chapter 12.

- Mark Wogahn has made sure that the software to produce this book (LaTeX and fonts) works properly.
- Ranjan Kumar Meher rewrote some of the LaTeX software used in the production of this new text.

Previous Editions

- **First three editions**. This book is derived from the previous editions, the first three of which were coauthored by James Peterson.

- **General contributions**. Others who helped us with previous editions include Hamid Arabnia, Rida Bazzi, Randy Bentson, David Black, Joseph Boykin, Jeff Brumfield, Gael Buckley, Roy Campbell, P. C. Capon, John Carpenter, Gil Carrick, Thomas Casavant, Bart Childs, Ajoy Kumar Datta, Joe Deck, Sudarshan K. Dhall, Thomas Doeppner, Caleb Drake, M. Rasit Eskicioglu, Hans Flack, Robert Fowler, G. Scott Graham, Richard Guy, ˘ Max Hailperin, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Don Heller, Bruce Hillyer, Mark Holliday, Dean Hougen, Michael Huang, Ahmed Kamel, Morty Kewstel, Richard Kieburtz, Carol Kroll, Morty Kwestel, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller, Michael Molloy, Euripides Montagne, Yoichi Muraoka, Jim M. Ng, Banu Ozden, Ed Posnak, Boris Putanec, Charles ¨

Qualline, John Quarterman, Mike Reiter, Gustavo Rodriguez-Rivera, Carolyn J. C. Schauble, Thomas P. Skinner, Yannis Smaragdakis, Jesse St. Laurent, John Stankovic, Adam Stauffer, Steven Stepanek, John Sterling, Hal Stern, Louis Stevens, Pete Thomas, David Umbaugh, Steve Vinoski, Tommy Wagner, Larry L. Wear, John Werth, James M. Westall, J. S. Weston, and Yang Xiang

- **Specifi Contributions**
  - Robert Love updated both Chapter 20 and the Linux coverage through out the text, as well as answering many of our Android-related ques tions.
  - Appendix B was written by Dave Probert and was derived from Chap ter 22 of the Eighth Edition of *Operating System Concepts.*
  - Jonathan Katz contributed to Chapter 16. Richard West provided input into Chapter 18. Salahuddin Khan updated Section 16.7 to provide new coverage of Windows 7 security.
  - Parts of Chapter 19 were derived from a paper by Levy and Silberschatz [1990].
  - Chapter 20 was derived from an unpublished manuscript by Stephen Tweedie.
  - Cliff Martin helped with updating the UNIX appendix to cover FreeBSD.
  - Some of the exercises and accompanying solutions were supplied by Arvind Krishnamurthy.
  - Andrew DeNicola prepared the student study guide that is available on our website. Some of the slides were prepared by Marilyn Turnamian.
  - Mike Shapiro, Bryan Cantrill, and Jim Mauro answered several Solaris related questions, and Bryan Cantrill from Sun Microsystems helped with the ZFS coverage. Josh Dees and Rob Reynolds contributed cover age of Microsoft’s NET.
  - Owen Galvin helped copy-edit Chapter 18 edition.

Book Production

The Executive Editor was Don Fowley. The Senior Production Editor was Ken Santor. The Freelance Developmental Editor was Chris Nelson. The Assistant Developmental Editor was Ryann Dannelly. The cover designer was Tom Nery. The copyeditor was Beverly Peavler. The freelance proofreader was Katrina Avery. The freelance indexer was WordCo, Inc. The Aptara LaTex team con sisted of Neeraj Saxena and Lav kush.

Personal Notes

Avi would like to acknowledge Valerie for her love, patience, and support during the revision of this book.

Peter would like to thank his wife Carla and his children, Gwen, Owen, and Maddie.

Greg would like to acknowledge the continued support of his family: his wife Pat and sons Thomas and Jay.

Abraham Silberschatz, New Haven, CT
Peter Baer Galvin, Boston, MA
Greg Gagne, Salt Lake City, UT

Contents

PART ONE OVERVIEW

Chapter 1 Introduction

Chapter 2 Operating-System Structures

PART TWO PROCESS MANAGEMENT

Chapter 3 Processes

Chapter 4 Threads & Concurrency

Chapter 5 CPU Scheduling

PART THREE PROCESS SYNCHRONIZATION

Chapter 6 Synchronization Tools

Chapter 7 Synchronization Examples

Chapter 8 Deadlocks

Contents

PART FOUR MEMORY MANAGEMENT

Chapter 9 Main Memory

Chapter 10 Virtual Memory

PART FIVE STORAGE MANAGEMENT

Chapter 11 Mass-Storage Structure

Chapter 12 I/O Systems

Contents

PART SIX FILE SYSTEM

Chapter 13 File-System Interface

Chapter 14 File-System Implementation

Chapter 15 File-System Internals

PART SEVEN SECURITY AND PROTECTION

Chapter 16 Security

Chapter 17 Protection

PART EIGHT ADVANCED TOPICS

Chapter 18 Virtual Machines

Chapter 19 Networks and Distributed Systems

PART NINE CASE STUDIES

Chapter 20 The Linux System

Chapter 21 Windows 10

PART TEN APPENDICES Chapter A

Influentia Operating Systems

Chapter B Windows 7

Part One

Overview

An *operating system* acts as an intermediary between the user of a com puter and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a *convenient* and *efficient* manner.

An operating system is software that manages the computer hard ware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent programs from interfering with the proper operation of the system.

Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task, and it is important that the goals of the system be well defined before the design begins.

Because an operating system is large and complex, it must be cre ated piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and func tions.

1 CHAPTER

Introduction

An **operating system** is software that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how they vary in accomplishing these tasks in a wide variety of computing environments. Operating systems are everywhere, from cars and home appliances that include "Internet of Things" devices, to smart phones, personal computers, enterprise computers, and cloud computing envi ronments.

In order to explore the role of an operating system in a modern computing environment, it is important first to understand the organization and architec ture of computer hardware. This includes the CPU, memory, and I/O devices, as well as storage. A fundamental responsibility of an operating system is to allocate these resources to programs.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. In this chapter, we provide a general overview of the major components of a contemporary computer system as well as the functions provided by the operating system. Additionally, we cover several topics to help set the stage for the remainder of the text: data structures used in operating systems, computing environments, and open-source and free operating systems.

CHAPTER OBJECTIVES

- Describe the general organization of a computer system and the role of interrupts.
- Describe the components in a modern multiprocessor computer system.
- Illustrate the transition from user mode to kernel mode.
- Discuss how operating systems are used in various computing environ ments.
- Provide examples of free and open-source operating systems.

1.1 What Operating Systems Do

We begin our discussion by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into four components: the ***hardware,*** the ***operating system,*** the ***application programs,*** and a ***user*** (Figure 1.1).

The **hardware**— the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system. The **application programs**—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an ***environment*** within which other programs can do useful work.

To understand more fully the operating system's role, we next explore operating systems from two viewpoints: that of the user and that of the system.

1.1.1 User View

The user's view of the computer varies according to the interface being used. Many computer users sit with a laptop or in front of a PC consisting of a monitor, keyboard, and mouse. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and security and none paid to **resource utilization**—how various hardware and software resources are shared.

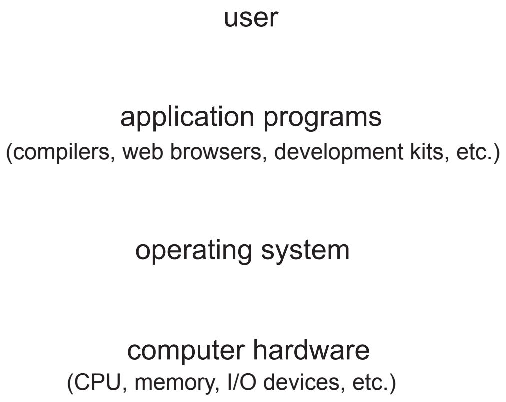

user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

Figure 1.1 Abstract view of the components of a computer system.

Increasingly, many users interact with mobile devices such as smartphones and tablets—devices that are replacing desktop and laptop computer systems for some users. These devices are typically connected to networks through cellular or other wireless technologies. The user interface for mobile computers generally features a **touch screen**, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse. Many mobile devices also allow users to interact through a **voice recognition** interface, such as Apple's **Siri**.

Some computers have little or no user view. For example, **embedded computers** in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating sys tems and applications are designed primarily to run without user intervention.

1.1.2 System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

1.1.3 Defining Operating Systems

By now, you can probably see that the term ***operating system*** covers many roles and functions. That is the case, at least in part, because of the myriad designs and uses of computers. Computers are present within toasters, cars, ships, spacecraft, homes, and businesses. They are the basis for game machines, cable TV tuners, and industrial control systems.

To explain this diversity, we can turn to the history of computers. Although computers have a relatively short history, they have evolved rapidly. Comput ing started as an experiment to determine what could be done and quickly moved to fixed-purpose systems for military uses, such as code breaking and trajectory plotting, and governmental uses, such as census calculation. Those early computers evolved into general-purpose, multifunction mainframes, and that's when operating systems were born. In the 1960s, **Moore's Law** predicted that the number of transistors on an integrated circuit would double every 18 months, and that prediction has held true. Computers gained in functionality and shrank in size, leading to a vast number of uses and a vast number and variety of operating systems. (See Appendix A for more details on the history of operating systems.)

How, then, can we define what an operating system is? In general, we have no completely adequate definition of an operating system. Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute programs and to make solving user problems easier. Computer hardware is constructed toward this goal. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

In addition, we have no universally accepted definition of what is part of the operating system. A simple viewpoint is that it includes everything a ven dor ships when you order “the operating system.” The features included, how ever, vary greatly across systems. Some systems take up less than a megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are based entirely on graphical windowing systems. A more com mon definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer—usually called the **kernel**. Along with the kernel, there are two other types of programs: **system programs**, which are associated with the operating system but are not neces sarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.

The matter of what constitutes an operating system became increasingly important as personal computers became more widespread and operating sys tems grew increasingly sophisticated. In 1998, the United States Department of Justice filed suit against Microsoft, in essence claiming that Microsoft included too much functionality in its operating systems and thus prevented application vendors from competing. (For example, a web browser was an integral part of Microsoft’s operating systems.) As a result, Microsoft was found guilty of using its operating-system monopoly to limit competition.

Today, however, if we look at operating systems for mobile devices, we see that once again the number of features constituting the operating system is increasing. Mobile operating systems often include not only a core kernel but also **middleware**—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems—Apple’s iOS and Google’s Android—features a core kernel along with middleware that supports databases, multimedia, and graphics (to name only a few).

WHY STUDY OPERATING SYSTEMS?

Although there are many practitioners of computer science, only a small per centage of them will be involved in the creation or modification of an operat ing system. Why, then, study operating systems and how they work? Simply because, as almost all code runs on top of an operating system, knowledge of how operating systems work is crucial to proper, efficient, effective, and secure programming. Understanding the fundamentals of operating systems, how they drive computer hardware, and what they provide to applications is not only essential to those who program them but also highly useful to those who write programs on them and use them.

In summary, for our purposes, the operating system includes the always running kernel, middleware frameworks that ease application development and provide features, and system programs that aid in managing the system while it is running. Most of this text is concerned with the kernel of general purpose operating systems, but other components are discussed as needed to fully explain operating system design and operation.

1.2 Computer-System Organization

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common **bus** that provides access between components and shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, a disk drive, audio device, or graphics display). Depending on the controller, more than one device may be attached. For instance, one system USB port can connect to a USB hub, to which several devices can connect. A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.

Typically, operating systems have a **device driver** for each device con troller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device. The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

In the following subsections, we describe some basics of how such a system operates, focusing on three key aspects of the system. We start with interrupts, which alert the CPU to events that require attention. We then discuss storage structure and I/O structure.

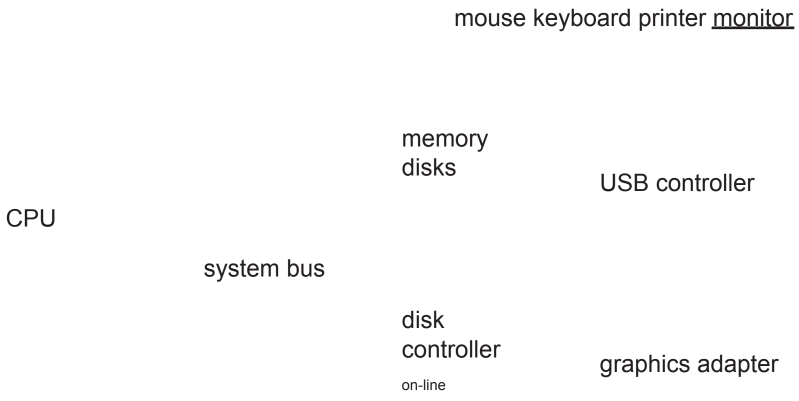


Figure 1.2 A typical PC computer system.

1.2.1 Interrupts

Consider a typical computer operation: a program performing I/O. To start an I/O operation, the device driver loads the appropriate registers in the device controller. The device controller, in turn, examines the contents of these reg isters to determine what action to take (such as “read a character from the keyboard”). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver that it has finished its operation. The device driver then gives control to other parts of the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information such as “write completed successfully” or “device busy”. But how does the controller inform the device driver that it has finished its operation? This is accomplished via an **interrupt**.

1.2.1.1 Overview

Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. (There may be many buses within a computer system, but the system bus is the main communications path between the major components.) Interrupts are used for many other purposes as well and are a key part of how operating systems and hardware interact.

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A timeline of this operation is shown in Figure 1.3. To run the animation assicated with this figure please click **here**.

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for managing this transfer would be to invoke a generic routine to examine the interrupt information. The routine, in turn,

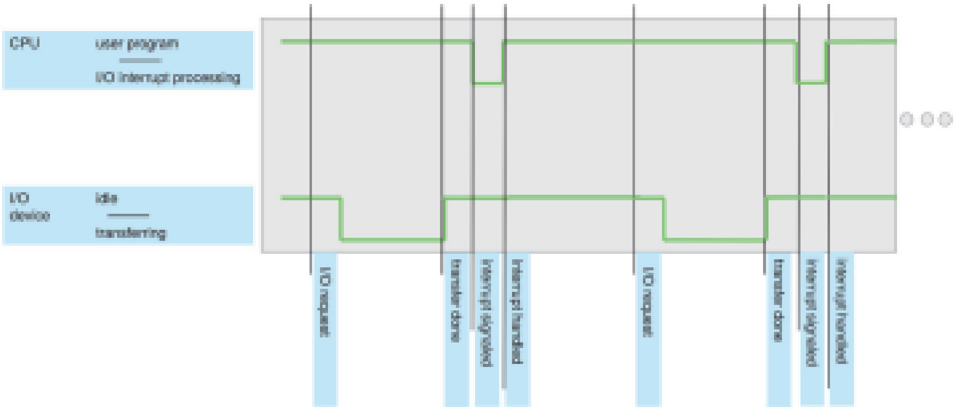


Figure 1.3 Interrupt timeline for a single program doing output.

would call the interrupt-specific handler. However, interrupts must be handled quickly, as they occur very frequently. A table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector**, of addresses is then indexed by a unique number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as Windows and UNIX dispatch interrupts in this manner.

The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt. If the interrupt routine needs to modify the processor state— for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

1.2.1.2 Implementation

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-request line, it reads the interrupt number and jumps to the **interrupt-handler routine** by using that interrupt number as an index into the interrupt vector. It then starts execution at the address associated with that index. The interrupt handler saves any state it will be changing during its operation, determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller ***raises*** an interrupt by asserting a signal on the interrupt request line, the CPU ***catches*** the interrupt and ***dispatches*** it to the interrupt handler, and the handler ***clears*** the interrupt by servicing the device. Figure 1.4 summarizes the interrupt-driven I/O cycle.

The basic interrupt mechanism just described enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. In a modern operating system, however, we need more sophisticated interrupt handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

In modern computer hardware, these three features are provided by the CPU and the **interrupt-controller hardware**.

CPU

1

device driver initiates I/O

2

4
transfers control to
interrupt handler

CPU executing checks for
interrupts between instructions

7

5

interrupt handler
processes data,
returns from interrupt

CPU receiving interrupt,

6

CPU resumes
processing of
interrupted task

3

I/O controller

input ready, output
complete, or error
generates interrupt signal

initiates I/O

Figure 1.4 Interrupt-driven I/O cycle.

Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors. The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

Recall that the purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Figure 1.5 illustrates the design of the interrupt vector for Intel processors. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

The interrupt mechanism also implements a system of **interrupt priority levels**. These levels enable the CPU to defer the handling of low-priority inter-

vector number description

0
1
2
3
4

5
6
7
8
9
10

11	device not available
12	double fault
13	coprocessor segment overrun
14	(reserved) invalid task state
15	segment
16	segment not present
17	stack fault
18	general protection
19–31	page fault
32–255	(Intel reserved, do not use)
divide error	floating-point error
debug exception	alignment check
null interrupt	machine check
breakpoint	(Intel reserved, do not use)
INTO-detected overflow	maskable interrupts
bound range exception	
invalid opcode	

Figure 1.5 Intel processor event-vector table.

rupts without masking all interrupts and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt. In summary, interrupts are used throughout modern operating systems to handle asynchronous events (and for other purposes we will discuss through out the text). Device controllers and hardware faults raise interrupts. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system perfor mance.

1.2.2 Storage Structure

The CPU can load instructions only from memory, so any programs must first be loaded into memory to run. General-purpose computers run most of their programs from rewritable memory, called main memory (also called **random-access memory**, or **RAM**). Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory** (**DRAM**).

Computers use other forms of memory as well. For example, the first pro gram to run on computer power-on is a **bootstrap program**, which then loads the operating system. Since RAM is **volatile**—loses its content when power is turned off or otherwise lost—we cannot trust it to hold the bootstrap pro gram. Instead, for this and some other purposes, the computer uses electri cally erasable programmable read-only memory (EEPROM) and other forms of **firmwar** —storage that is infrequently written to and is nonvolatile. EEPROM

STORAGE DEFINITIONS AND NOTATION

The basic unit of computer storage is the **bit**. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A **byte** is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes. For example, a computer that has 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words. A computer executes many operations in its native word size rather than a byte at a time.

Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A **kilobyte**, or **KB**, is 1,024 bytes; a **megabyte**, or **MB**, is $1{,}024^2$ bytes; a **gigabyte**, or **GB**, is $1{,}024^3$ bytes; a **terabyte**, or **TB**, is $1{,}024^4$ bytes; and a **petabyte**, or **PB**, is $1{,}024^5$ bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes. Networking measurements are an exception to this general rule; they are given in bits (because networks move data a bit at a time).

can be changed but cannot be changed frequently. In addition, it is low speed, and so it contains mostly static programs and data that aren't frequently used. For example, the iPhone uses EEPROM to store serial numbers and hardware information about the device.

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a byte or word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution from the location stored in the program counter.

A typical instruction–execution cycle, as executed on a system with a **von Neumann architecture**, first fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses. It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible on most systems for two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.

2. Main memory, as mentioned, is volatile—it loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage devices are **hard-disk drives** (**HDDs**) and **nonvolatile memory** (**NVM**) **devices**, which provide storage for both programs and data. Most programs (system and application) are stored in secondary storage until they are loaded into memory. Many programs then use secondary storage as both the source and the destination of their processing. Secondary storage is also much slower than main memory. Hence, the proper management of secondary storage is of central importance to a computer sys tem, as we discuss in Chapter 11.

In a larger sense, however, the storage structure that we have described —consisting of registers, main memory, and secondary storage—is only one of many possible storage system designs. Other possible components include cache memory, CD-ROM or blu-ray, magnetic tapes, and so on. Those that are slow enough and large enough that they are used only for special purposes — to store backup copies of material stored on other devices, for example— are called **tertiary storage**. Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, size, and volatility.

The wide variety of storage systems can be organized in a hierarchy (Figure 1.6) according to storage capacity and access time. As a general rule, there is a trade-off between size and speed, with smaller and faster memory closer to the CPU. As shown in the figure, in addition to differing in speed and capacity, the various storage systems are either volatile or nonvolatile. Volatile storage, as mentioned earlier, loses its contents when the power to the device is removed, so data must be written to nonvolatile storage for safekeeping.

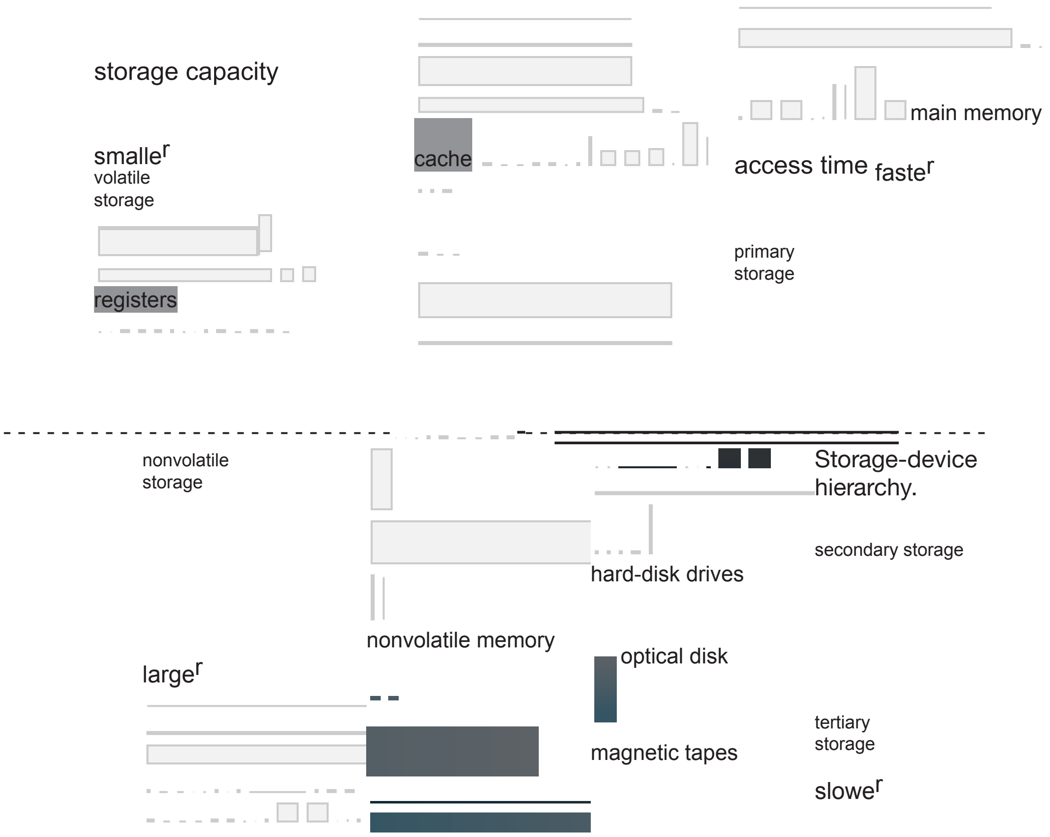


Figure 1.6 Storage-device hierarchy.

The top four levels of memory in the figure are constructed using **semiconductor memory**, which consists of semiconductor-based electronic circuits. NVM devices, at the fourth level, have several variants but in general are faster than hard disks. The most common form of NVM device is flash memory, which is popular in mobile devices such as smartphones and tablets. Increasingly, flash memory is being used for long-term storage on laptops, desktops, and servers as well.

Since storage plays an important role in operating-system structure, we will refer to it frequently in the text. In general, we will use the following terminology:

- Volatile storage will be referred to simply as **memory**. If we need to empha size a particular type of storage device (for example, a register),we will do so explicitly.
- Nonvolatile storage retains its contents when power is lost. It will be referred to as **NVS**. The vast majority of the time we spend on NVS will be on secondary storage. This type of storage can be classified into two distinct types:
    - **Mechanical**. A few examples of such storage systems are HDDs, optical disks, holographic storage, and magnetic tape. If we need to emphasize a particular type of mechanical storage device (for example, magnetic tape), we will do so explicitly.
    - **Electrical**. A few examples of such storage systems are flash memory, FRAM, NRAM, and SSD. Electrical storage will be referred to as **NVM**. If we need to emphasize a particular type of electrical storage device (for example, SSD), we will do so explicitly.

  Mechanical storage is generally larger and less expensive per byte than electrical storage. Conversely, electrical storage is typically costly, smaller, and faster than mechanical storage.

The design of a complete storage system must balance all the factors just discussed: it must use only as much expensive memory as necessary while providing as much inexpensive, nonvolatile storage as possible. Caches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components.

1.2.3 I/O Structure

A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

Recall from the beginning of this section that a general-purpose computer system consists of multiple devices, all of which exchange data via a common

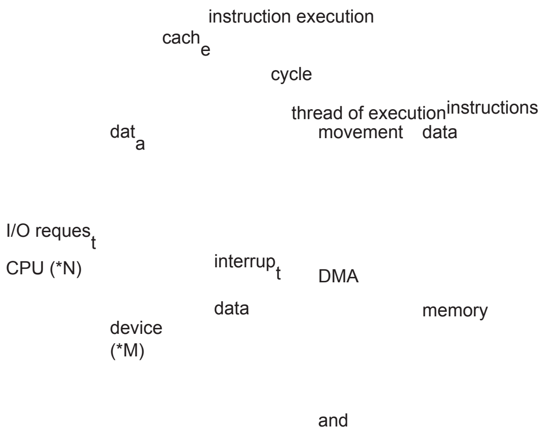


Figure 1.7 How a modern computer system works.

bus. The form of interrupt-driven I/O described in Section 1.2.1 is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as NVS I/O. To solve this problem, **direct memory access** (**DMA**) is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from the device and main memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective. Figure 1.7 shows the interplay of all components of a computer system.

1.3 Computer-System Architecture

In Section 1.2, we introduced the general structure of a typical computer system. A computer system can be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

1.3.1 Single-Processor Systems

Many years ago, most computer systems used a single processor containing one CPU with a single processing core. The **core** is the component that executes instructions and registers for storing data locally. The one main CPU with its core is capable of executing a general-purpose instruction set, including instructions from processes. These systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers.

All of these special-purpose processors run a limited instruction set and do not run processes. Sometimes, they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status. For example, a disk-controller microprocessor receives a sequence of requests from the main CPU core and implements its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In other systems or circumstances, special-purpose processors are low-level components built into the hardware. The operating system cannot communicate with these processors; they do their jobs autonomously. The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiproces sor. If there is only one general-purpose CPU with a single processing core, then the system is a single-processor system. According to this definition, however, very few contemporary computer systems are single-processor systems.

1.3.2 Multiprocessor Systems

On modern computers, from mobile devices to servers, **multiprocessor systems** now dominate the landscape of computing. Traditionally, such systems have two (or more) processors, each with a single-core CPU. The proces sors share the computer bus and sometimes the clock, memory, and periph eral devices. The primary advantage of multiprocessor systems is increased throughput. That is, by increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with *N* processors is not *N*, however; it is less than *N*. When multiple processors cooperate on a task, a cer tain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors.

The most common multiprocessor systems use **symmetric multiprocessing** (**SMP**), in which each peer CPU processor performs all tasks, including operating-system functions and user processes. Figure 1.8 illustrates a typical SMP architecture with two processors, each with its own CPU. Notice that each CPU processor has its own set of registers, as well as a private—or local— cache. However, all processors share physical memory over the system bus.

The benefit of this model is that many processes can run simultaneously —*N* processes can run if there are *N* CPUs—without causing performance to deteriorate significantly. However, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures. A multiprocessor system of this form will allow processes and resources—such as memory— to be shared dynamically among the various processors and can lower the workload variance among the processors. Such a system must be written carefully, as we shall see in Chapter 5 and Chapter 6.

The definition of ***multiprocessor*** has evolved over time and now includes **multicore** systems, in which multiple computing cores reside on a single chip. Multicore systems can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication.

Figure 1.8 Symmetric multiprocessing architecture.

In addition, one chip with multiple cores uses significantly less power than multiple single-core chips, an important issue for mobile devices as well as laptops.

In Figure 1.9, we show a dual-core design with two cores on the same processor chip. In this design, each core has its own register set, as well as its own local cache, often known as a level 1, or L1, cache. Notice, too, that a level 2 (L2) cache is local to the chip but is shared by the two processing cores. Most archi tectures adopt this approach, combining local and shared caches, where local, lower-level caches are generally smaller and faster than higher-level shared

Figure 1.9 A dual-core design with two cores on the same chip.

DEFINITIONS OF COMPUTER SYSTEM COMPONENTS

- **CPU**—The hardware that executes instructions.
- **Processor**—A physical chip that contains one or more CPUs.
- **Core**—The basic computation unit of the CPU.
- **Multicore**— Including multiple computing cores on the same CPU.
- **Multiprocessor**— Including multiple processors.

Although virtually all systems are now multicore, we use the general term ***CPU*** when referring to a single computational unit of a computer system and ***core*** as well as ***multicore*** when specifically referring to one or more cores on a CPU.

caches. Aside from architectural considerations, such as cache, memory, and bus contention, a multicore processor with N cores appears to the operating sys tem as N standard CPUs. This characteristic puts pressure on operating-system designers—and application programmers— to make efficient use of these pro cessing cores, an issue we pursue in Chapter 4. Virtually all modern operating systems—including Windows, macOS, and Linux, as well as Android and iOS mobile systems—support multicore SMP systems.

Adding additional CPUs to a multiprocessor system will increase computing power; however, as suggested earlier, the concept does not scale very well, and once we add too many CPUs, contention for the system bus becomes a bottleneck and performance begins to degrade. An alternative approach is instead to provide each CPU (or group of CPUs) with its own local memory that is accessed via a small, fast local bus. The CPUs are connected by a **shared system interconnect**, so that all CPUs share one physical address space. This approach—known as **non-uniform memory access**, or **NUMA**—is illustrated in Figure 1.10. The advantage is that, when a CPU accesses its local memory, not only is it fast, but there is also no contention over the system interconnect. Thus, NUMA systems can scale more effectively as more processors are added.

A potential drawback with a NUMA system is increased latency when a CPU must access remote memory across the system interconnect, creating a possible performance penalty. In other words, for example, CPU$_0$ cannot access the local memory of CPU$_3$ as quickly as it can access its own local memory, slowing down performance. Operating systems can minimize this NUMA penalty through careful CPU scheduling and memory management, as discussed in Section 5.5.2 and Section 10.5.4. Because NUMA systems can scale to accommodate a large number of processors, they are becoming increasingly popular on servers as well as high-performance computing systems.

Finally, **blade servers** are systems in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade processor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between

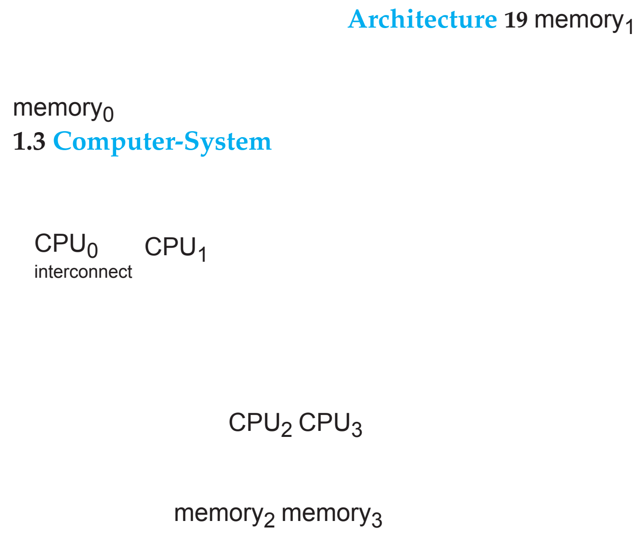


Figure 1.10 NUMA multiprocessing architecture.

types of computers. In essence, these servers consist of multiple independent multiprocessor systems.

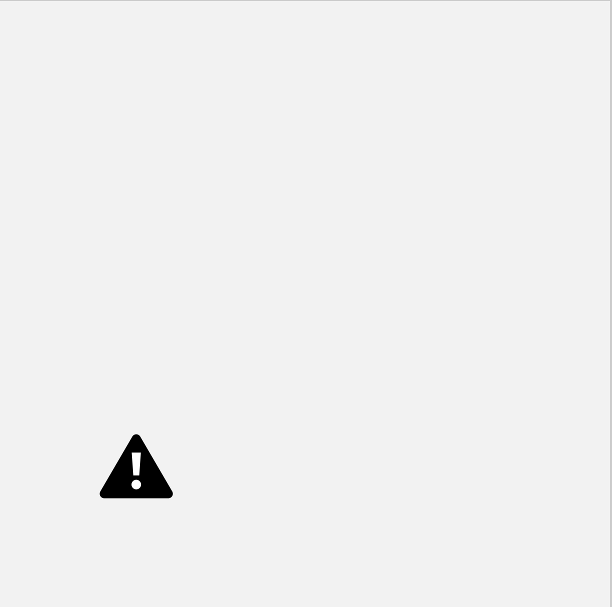
1.3.3 Clustered Systems

Another type of multiprocessor system is a **clustered system**, which gath ers together multiple CPUs. Clustered systems differ from the multiprocessor systems described in Section 1.3.2 in that they are composed of two or more individual systems—or nodes—joined together; each node is typically a mul ticore system. Such systems are considered **loosely coupled**. We should note that the definition of ***clustered*** is not concrete; many commercial and open source packages wrestle to define what a clustered system is and why one form is better than another. The generally accepted definition is that clustered computers share storage and are closely linked via a local-area network LAN (as described in Chapter 19) or a faster interconnect, such as InfiniBand.

Clustering is usually used to provide **high-availability service**— that is, service that will continue even if one or more systems in the cluster fail. Generally, we obtain high availability by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the network). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.

High availability provides increased reliability, which is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.

Clustering can be structured asymmetrically or symmetrically. In **asym metric clustering**, one machine is in **hot-standby mode** while the other is run ning the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active



PC MOTHERBOARD

Consider the desktop PC motherboard with a

processor socket shown below:

This board is a fully functioning computer, once its slots are populated. It consists of a processor socket containing a CPU, DRAM sockets, PCIe bus slots, and I/O connectors of various types. Even the lowest-cost general purpose CPU contains multiple cores. Some motherboards contain multiple processor sockets. More advanced computers allow more than one system board, creating NUMA systems.

server. In **symmetric clustering**, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware. However, it does require that more than one application be available to run.

Since a cluster consists of several computer systems connected via a network, clusters can also be used to provide **high-performance computing** environments. Such systems can supply significantly greater computational power than single-processor or even SMP systems because they can run an application concurrently on all computers in the cluster. The application must have been written specifically to take advantage of the cluster, however. This involves a technique known as **parallelization**, which divides a program into separate components that run in parallel on individual cores in a computer or computers in a cluster. Typically, these applications are designed so that once each computing node in the cluster has solved its portion of the problem, the results from all the nodes are combined into a final solution.

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN) (as described in Chapter 19). Parallel clusters allow multiple hosts to access the same data on shared storage. Because most oper-

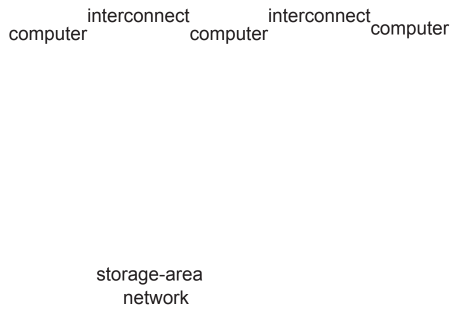


Figure 1.11 General structure of a clustered system.

ating systems lack support for simultaneous data access by multiple hosts, parallel clusters usually require the use of special versions of software and special releases of applications. For example, Oracle Real Application Cluster is a version of Oracle's database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. To provide this shared access, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a **distributed lock manager** (**DLM**), is included in some cluster technology.

Cluster technology is changing rapidly. Some cluster products support thousands of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by **storage-area networks** (**SANs**), as described in Section 11.7.4, which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability. Figure 1.11 depicts the general structure of a clustered system.

1.4 Operating-System Operations

Now that we have discussed basic information about computer-system organi zation and architecture, we are ready to talk about operating systems. An oper ating system provides the environment within which programs are executed. Internally, operating systems vary greatly, since they are organized along many different lines. There are, however, many commonalities, which we consider in this section.

For a computer to start running— for instance, when it is powered up or rebooted—it needs to have an initial program to run. As noted earlier, this initial program, or bootstrap program, tends to be simple. Typically, it is stored within the computer hardware in firmware. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to

HADOOP

Hadoop is an open-source software framework that is used for distributed processing of large data sets (known as **big data**) in a clustered system con taining simple, low-cost hardware components. Hadoop is designed to scale from a single system to a cluster containing thousands of computing nodes. Tasks are assigned to a node in the cluster, and Hadoop arranges communica tion between nodes to manage parallel computations to process and coalesce results. Hadoop also detects and manages failures in nodes, providing an efficient and highly reliable distributed computing service.

Hadoop is organized around the following three components:

1. A distributed file system that manages data and files across distributed computing nodes.
2. The YARN ("Yet Another Resource Negotiator") framework, which manages resources within the cluster as well as scheduling tasks on nodes in the cluster.
3. The **MapReduce** system, which allows parallel processing of data across nodes in the cluster.

Hadoop is designed to run on Linux systems, and Hadoop applications can be written using several programming languages, including scripting languages such as PHP, Perl, and Python. Java is a popular choice for developing Hadoop applications, as Hadoop has several Java libraries that support MapReduce. More information on MapReduce and Hadoop can be found at https://hadoop.apache.org/docs/r1.2.1/mapred tutorial.html and https://hadoop.apache.org

start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory. Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel by system programs that are loaded into memory at boot time to become **system daemons**, which run the entire time the kernel is running. On Linux, the first system program is "systemd," and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt. In Section 1.2.1 we described hardware interrupts. Another form of interrupt is a **trap** (or an **exception**), which is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed by executing a special operation called a **system call**.

1.4.1 Multiprogramming and Multitasking

One of the most important aspects of operating systems is the ability to run multiple programs, as a single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Furthermore, users typically *want* to run more than one program at a time as well. **Multiprogramming** increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute. In a multiprogrammed system, a program in execution is termed a **process**.

The idea is as follows: The operating system keeps several processes in memory simultaneously (Figure 1.12). The operating system picks and begins to execute one of these processes. Eventually, the process may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another process. When ***that*** process needs to wait, the CPU switches to ***another*** process, and so on. Eventually, the first process finishes waiting and gets the CPU back. As long as at least one process needs to execute, the CPU is never idle.

This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If she has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.)

Multitasking is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast **response time**. Consider that when a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or touch screen. Since interactive I/O typically runs at "peo ple speeds," it may take a long time to complete. Input, for example, may be

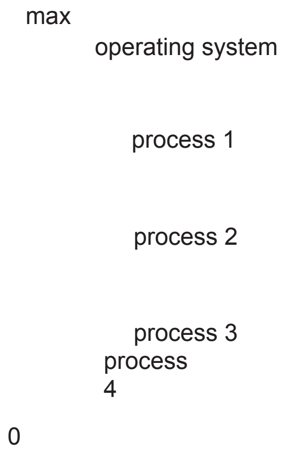


Figure 1.12 Memory layout for a multiprogramming system.

bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to another process.

Having several processes in memory at the same time requires some form of memory management, which we cover in Chapter 9 and Chapter 10. In addition, if several processes are ready to run at the same time, the system must choose which process will run next. Making this decision is **CPU scheduling**, which is discussed in Chapter 5. Finally, running multiple processes concur rently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. We discuss these considerations throughout the text.

In a multitasking system, the operating system must ensure reasonable response time. A common method for doing so is **virtual memory**, a technique that allows the execution of a process that is not completely in memory (Chapter 10). The main advantage of this scheme is that it enables users to run programs that are larger than actual **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical mem ory** as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

Multiprogramming and multitasking systems must also provide a file sys tem (Chapter 13, Chapter 14, and Chapter 15). The file system resides on a secondary storage; hence, storage management must be provided (Chapter 11). In addition, a system must protect resources from inappropriate use (Chapter 17). To ensure orderly execution, the system must also provide mechanisms for process synchronization and communication (Chapter 6 and Chapter 7), and it may ensure that processes do not get stuck in a deadlock, forever waiting for one another (Chapter 8).

1.4.2 Dual-Mode and Multimode Operation

Since the operating system and its users share the hardware and software resources of the computer system, a properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs —or the operating system itself— to execute incorrectly. In order to ensure the proper execution of the system, we must be able to distinguish between the execution of operating-system code and user-defined code. The approach taken by most computer systems is to provide hardware support that allows differentiation among various modes of execution.

At the very least, we need two separate ***modes*** of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request. This is shown in Figure 1.13. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well. At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to

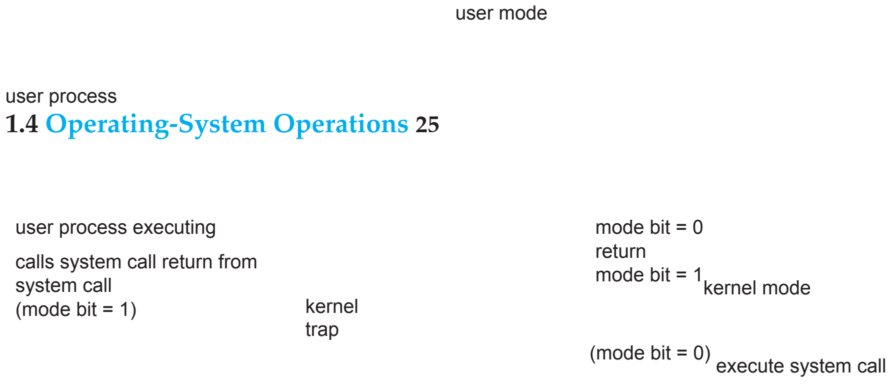


Figure 1.13 Transition from user to kernel mode.

a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management. Many additional privileged instructions are discussed throughout the text.

The concept of modes can be extended beyond two modes. For example, Intel processors have four separate **protection rings**, where ring 0 is kernel mode and ring 3 is user mode. (Although rings 1 and 2 could be used for various operating-system services, in practice they are rarely used.) ARMv8 systems have seven modes. CPUs that support virtualization (Section 18.1) frequently have a separate mode to indicate when the **virtual machine manager** (**VMM**) is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel. It needs that level of privilege so it can create and manage virtual machines, changing the CPU state to do so.

We can now better understand the life cycle of instruction execution in a computer system. Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call. Most contemporary operating systems—such as Microsoft Windows, Unix, and Linux—

take advantage of this dual-mode feature and provide greater protection for the operating system.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic `trap` instruction, although some systems have a specific `syscall` instruction to invoke a system call.

When a system call is executed, it is typically treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call. We describe system calls more fully in Section 2.3.

Once hardware protection is in place, it detects errors that violate modes. These errors are normally handled by the operating system. If a user program fails in some way—such as by making an attempt either to execute an illegal instruction or to access memory that is not in the user's address space— then the hardware traps to the operating system. The trap transfers control through the interrupt vector to the operating system, just as an interrupt does. When a program error occurs, the operating system must terminate the program abnormally. This situation is handled by the same code as a user-requested abnormal termination. An appropriate error message is given, and the memory of the program may be dumped. The memory dump is usually written to a file so that the user or programmer can examine it and perhaps correct it and restart the program.

1.4.3 Timer

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

LINUX TIMERS

On Linux systems, the kernel configuration parameter HZ specifies the fre quency of timer interrupts. An HZ value of 250 means that the timer generates 250 interrupts per second, or one interrupt every 4 milliseconds. The value of HZ depends upon how the kernel is configured, as well the machine type and architecture on which it is running. A related kernel variable is jiffies, which represent the number of timer interrupts that have occurred since the system was booted. A programming project in Chapter 2 further explores timing in the Linux kernel.

1.5 Resource Management

As we have seen, an operating system is a **resource manager**. The system's CPU, memory space, file-storage space, and I/O devices are among the resources that the operating system must manage.

1.5.1 Process Management

A program can do nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A program such as a compiler is a process, and a word-processing program being run by an individual user on a PC is a process. Similarly, a social media app on a mobile device is a process. For now, you can consider a process to be an instance of a program in execution, but later you will see that the concept is more general. As described in Chapter 3, it is possible to provide system calls that allow processes to create subprocesses to execute concurrently.

A process needs certain resources—including CPU time, memory, files, and I/O devices— to accomplish its task. These resources are typically allocated to the process while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along. For example, consider a process running a web browser whose function is to display the contents of a web page on a screen. The process will be given the URL as an input and will execute the appropriate instructions and system calls to obtain and display the desired information on the screen. When the process terminates, the operating system will reclaim any reusable resources.

We emphasize that a program by itself is not a process. A program is a ***passive*** entity, like the contents of a file stored on disk, whereas a process is an ***active*** entity. A single-threaded process has one **program counter** specifying the next instruction to execute. (Threads are covered in Chapter 4.) The exe cution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, one instruction at most is executed on behalf of the process. Thus, although

two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

A process is the unit of work in a system. A system consists of a collec tion of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that exe cute user code). All these processes can potentially execute concurrently—by multiplexing on a single CPU core—or in parallel across multiple CPU cores.

The operating system is responsible for the following activities in connec tion with process management:

- Creating and deleting both user and system processes
- Scheduling processes and threads on the CPUs
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

We discuss process-management techniques in Chapter 3 through Chapter 7.

1.5.2 Memory Management

As discussed in Section 1.2.2, the main memory is central to the operation of a modern computer system. Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. Each byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The CPU reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on a von Neumann architecture). As noted earlier, the main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from disk, those data must first be transferred to main memory by CPU-generated I/O calls. In the same way, instructions must be in memory for the CPU to execute them.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management. Many different memory management schemes are used. These schemes reflect various approaches, and the effectiveness of any given algorithm depends on the situation. In selecting a memory-management scheme for a specific system, we must take into account many factors—especially the ***hardware*** design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and which process is using them
- Allocating and deallocating memory space as needed
- Deciding which processes (or parts of processes) and data to move into and out of memory

Memory-management techniques are discussed in Chapter 9 and Chapter 10.

1.5.3 File-System Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. The operating system maps files onto physical media and accesses these files via the storage devices.

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Secondary storage is the most common, but tertiary storage is also possible. Each of these media has its own characteristics and physical organization. Most are controlled by a device, such as a disk drive, that also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form (for example, text files), or they may be formatted rigidly (for example, fixed fields such as an mp3 music file). Clearly, the concept of a file is an extremely general one.

The operating system implements the abstract concept of a file by managing mass storage media and the devices that control them. In addition, files are normally organized into directories to make them easier to use. Finally, when multiple users have access to files, it may be desirable to control which user may access a file and how that user may access it (for example, read, write, append).

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto mass storage
- Backing up files on stable (nonvolatile) storage media

File-management techniques are discussed in Chapter 13, Chapter 14, and Chapter 15.

1.5.4 Mass-Storage Management

As we have already seen, the computer system must provide secondary storage to back up main memory. Most modern computer systems use HDDs and NVM devices as the principal on-line storage media for both programs and data. Most programs—including compilers, web browsers, word processors, and games—are stored on these devices until loaded into memory. The programs then use the devices as both the source and the destination of their processing. Hence, the proper management of secondary storage is of central importance to a computer system. The operating system is responsible for the following activities in connection with secondary storage management:

- Mounting and unmounting
- Free-space management
- Storage allocation
- Disk scheduling
- Partitioning
- Protection

Because secondary storage is used frequently and extensively, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the secondary storage subsystem and the algorithms that manipulate that subsystem.

At the same time, there are many uses for storage that is slower and lower in cost (and sometimes higher in capacity) than secondary storage. Backups of disk data, storage of seldom-used data, and long-term archival storage are some examples. Magnetic tape drives and their tapes and CD DVD and Blu-ray drives and platters are typical tertiary storage devices.

Tertiary storage is not crucial to system performance, but it still must be managed. Some operating systems take on this task, while others leave tertiary-storage management to application programs. Some of the functions that operating systems can provide include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

Techniques for secondary storage and tertiary storage management are discussed in Chapter 11.

1.5.5 Cache Management

Caching is an important principle of computer systems. Here's how it works. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system— the cache—on a tem porary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache.

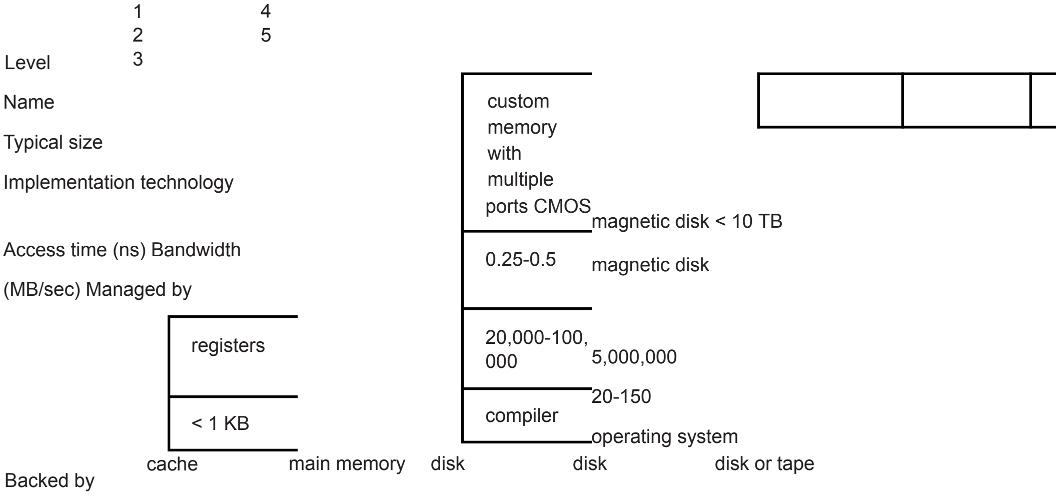

1
2
3
4
5
Level
Name
Typical size
Implementation technology
Access time (ns) Bandwidth
(MB/sec) Managed by
Backed by
registers
< 1 KB
custom memory with multiple ports CMOS
0.25-0.5
20,000-100, 000
compiler
cache
main memory
disk
magnetic disk < 10 TB
magnetic disk
5,000,000
20-150
operating system
disk
disk or tape

Figure 1.14 Characteristics of various types of storage.

If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

In addition, internal programmable registers provide a high-speed cache for main memory. The programmer (or compiler) implements the register allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.

Other caches are implemented totally in hardware. For instance, most systems have an instruction cache to hold the instructions expected to be executed next. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy. We are not concerned with these hardware-only caches in this text, since they are outside the control of the operating system.

Because caches have limited size, cache management is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance, as you can see by examining Figure 1.14. Replacement algorithms for software-controlled caches are discussed in Chapter 10.

The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the control ling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system inter vention. In contrast, transfer of data from disk to memory is usually controlled by the operating system.

In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on hard disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the hard disk, in main memory, in the cache, and in an internal register (see Figure 1.15). Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A

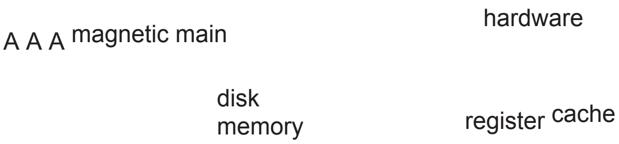


Figure 1.15 Migration of integer A from disk to register.

becomes the same only after the new value of A is written from the internal register back to the hard disk.

In a computing environment where only one process executes at a time, this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy. However, in a multitasking environment, where the CPU is switched back and forth among various pro cesses, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.

The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache (refer back to Figure 1.8). In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute in parallel, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**, and it is usually a hardware issue (handled below the operating-system level).

In a distributed environment, the situation becomes even more complex. In this environment, several copies (or replicas) of the same file can be kept on different computers. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible. There are various ways to achieve this guarantee, as we discuss in Chapter 19.

1.5.6 I/O System Management

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the **I/O subsystem**. The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

We discussed earlier in this chapter how interrupt handlers and device drivers are used in the construction of efficient I/O subsystems. In Chapter 12, we discuss how the I/O subsystem interfaces to the other system components, manages devices, transfers data, and detects I/O completion.

1.6 Security and Protection

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authoriza tion from the operating system. For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU without eventually relinquishing control. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

Protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage, as we discuss in Chapter 17.

A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information (her means of identifying herself to the system) is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system). Prevention of some of these attacks is considered an operating system function on some systems, while other systems leave it to policy or additional software. Due to the alarming rise in security incidents, operating system security features are a fast-growing area of research and implementa tion. We discuss security in Chapter 16.

Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and asso ciated **user identifier** (**user IDs**). In Windows parlance, this is a **security ID** (**SID**). These numerical IDs are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When an ID needs to be readable by a user, it is translated back to the user name via the user name list.

In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may be allowed only to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group. Group functionality can be implemented as a system-wide list of group names and **group identifier** . A user can be in one or more groups, depending on operating-system design

decisions. The user's group IDs are also included in every associated process and thread.

In the course of normal system use, the user ID and group ID for a user are sufficient. However, a user sometimes needs to **escalate privileges** to gain extra permissions for an activity. The user may need access to a device that is restricted, for example. Operating systems provide various methods to allow privilege escalation. On UNIX, for instance, the *setuid* attribute on a program causes that program to run with the user ID of the owner of the file, rather than the current user's ID. The process runs with this **effective UID** until it turns off the extra privileges or terminates.

1.7 Virtualization

Virtualization is a technology that allows us to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer. These environments can be viewed as different individual operating systems (for example, Windows and UNIX) that may be running at the same time and may interact with each other. A user of a **virtual machine** can switch among the various operating systems in the same way a user can switch among the various processes running concurrently in a single operating system.

Virtualization allows operating systems to run as applications within other operating systems. At first blush, there seems to be little reason for such func tionality. But the virtualization industry is vast and growing, which is a testa ment to its utility and importance.

Broadly speaking, virtualization software is one member of a class that also includes emulation. **Emulation**, which involves simulating computer hard ware in software, is typically used when the source CPU type is different from the target CPU type. For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its desktop and laptop computers, it included an emulation facility called "Rosetta," which allowed applications compiled for the IBM CPU to run on the Intel CPU. That same concept can be extended to allow an entire operating system written for one platform to run on another. Emula tion comes at a heavy price, however. Every machine-level instruction that runs natively on the source system must be translated to the equivalent function on the target system, frequently resulting in several target instructions. If the source and target CPUs have similar performance levels, the emulated code may run much more slowly than the native code.

With virtualization, in contrast, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU. Virtualization first came about on IBM mainframes as a method for multiple users to run tasks concurrently. Running multiple vir tual machines allowed (and still allows) many users to run tasks on a system designed for a single user. Later, in response to problems with running multiple Microsoft Windows applications on the Intel x86 CPU, VMware created a new virtualization technology in the form of an application that ran on Windows. That application ran one or more **guest** copies of Windows or other native x86 operating systems, each running its own applications. (See Figure 1.16.)

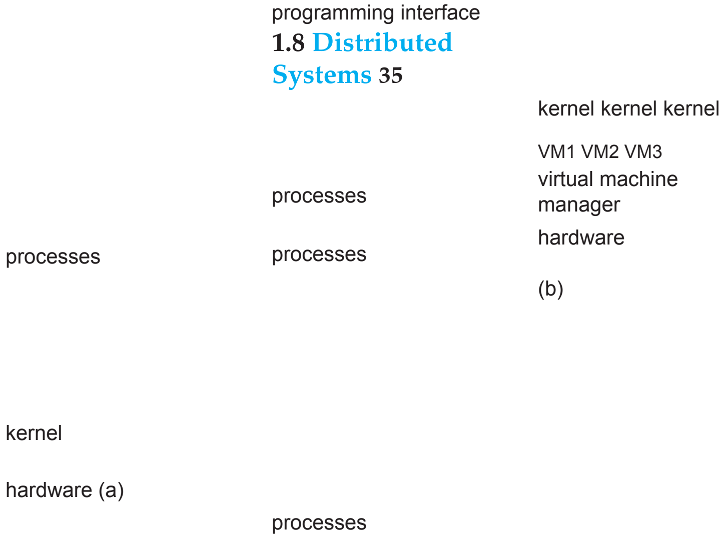


Figure 1.16 A computer running (a) a single operating system and (b) three virtual machines.

Windows was the **host** operating system, and the VMware application was the **virtual machine manager** (**VMM**). The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others.

Even though modern operating systems are fully capable of running multi ple applications reliably, the use of virtualization continues to grow. On laptops and desktops, a VMM allows the user to install multiple operating systems for exploration or to run applications written for operating systems other than the native host. For example, an Apple laptop running macOS on the x86 CPU can run a Windows 10 guest to allow execution of Windows applications. Com panies writing software for multiple operating systems can use virtualization to run all of those operating systems on a single physical server for develop ment, testing, and debugging. Within data centers, virtualization has become a common method of executing and managing computing environments. VMMs like VMware ESXand Citrix XenServer no longer run on host operating systems but rather *are* the host operating systems, providing services and resource management to virtual machine processes.

With this text, we provide a Linux virtual machine that allows you to run Linux—as well as the development tools we provide—on your personal system regardless of your host operating system. Full details of the features and implementation of virtualization can be found in Chapter 18.

1.8 Distributed Systems

A distributed system is a collection of physically separate, possibly heteroge neous computer systems that are networked to provide users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability.

Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface's device driver.

Others make users specifically invoke network functions. Generally, systems contain a mix of the two modes— for example FTP and NFS. The protocols that create a distributed system can greatly affect that system's utility and popularity.

A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functional ity. Networks vary by the protocols used, the distances between nodes, and the transport media. **TCP/IP** is the most common network protocol, and it provides the fundamental architecture of the Internet. Most operating systems support TCP/IP, including all general-purpose ones. Some systems support proprietary protocols to suit their needs. For an operating system, it is necessary only that a network protocol have an interface device—a network adapter, for example —with a device driver to manage it, as well as software to handle data. These concepts are discussed throughout this book.

Networks are characterized based on the distances between their nodes. A **local-area network** (**LAN**) connects computers within a room, a building, or a campus. A **wide-area network** (**WAN**) usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide, for example. These networks may run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks. For example, a **metropolitan-area network** (**MAN**) could link buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **personal-area network** (**PAN**) between a phone and a headset or a smartphone and a desktop computer.

The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network. Even very short-range infrared communication can be used for networking. At a rudimentary level, whenever computers communicate, they use or create a network. These networks also vary in their performance and reliability.

Some operating systems have taken the concept of networks and dis tributed systems further than the notion of providing network connectivity. A **network operating system** is an operating system that provides features such as file sharing across the network, along with a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers. A distributed operat ing system provides a less autonomous environment. The different computers communicate closely enough to provide the illusion that only a single operat ing system controls the network. We cover computer networks and distributed systems in Chapter 19.

1.9 Kernel Data Structures

We turn next to a topic central to operating-system implementation: the way data are structured in the system. In this section, we briefly describe several `data data data null ...`

Figure 1.17 Singly linked list.

fundamental data structures used extensively in operating systems. Readers who require further details on these structures, as well as others, should consult the bibliography at the end of the chapter.

1.9.1 Lists, Stacks, and Queues

An array is a simple data structure in which each element can be accessed directly. For example, main memory is constructed as an array. If the data item being stored is larger than one byte, then multiple bytes can be allocated to the item, and the item is addressed as "item number × item size." But what about storing an item whose size may vary? And what about removing an item if the relative positions of the remaining items must be preserved? In such situations, arrays give way to other data structures.

After arrays, lists are perhaps the most fundamental data structures in computer science. Whereas each item in an array can be accessed directly, the items in a list must be accessed in a particular order. That is, a **list** represents a collection of data values as a sequence. The most common method for implementing this structure is a **linked list**, in which items are linked to one another. Linked lists are of several types:

- In a ***singly linked list,*** each item points to its successor, as illustrated in Figure 1.17.
- In a ***doubly linked list,*** a given item can refer either to its predecessor or to its successor, as illustrated in Figure 1.18.
- In a ***circularly linked list,*** the last element in the list refers to the first element, rather than to null, as illustrated in Figure 1.19.

Linked lists accommodate items of varying sizes and allow easy insertion and deletion of items. One potential disadvantage of using a list is that performance for retrieving a specified item in a list of size n is linear—$O(n)$, as it requires potentially traversing all n elements in the worst case. Lists are sometimes used directly by kernel algorithms. Frequently, though, they are used for constructing more powerful data structures, such as stacks and queues.

A **stack** is a sequentially ordered data structure that uses the last in, first out (**LIFO**) principle for adding and removing items, meaning that the last item

`data null data data data null ...`

Figure 1.18 Doubly linked list.

`data data data data ...`

Figure 1.19 Circularly linked list.

placed onto a stack is the first item removed. The operations for inserting and removing items from a stack are known as ***push*** and ***pop***, respectively. An operating system often uses a stack when invoking function calls. Parameters, local variables, and the return address are pushed onto the stack when a function is called; returning from the function call pops those items off the stack.

A **queue**, in contrast, is a sequentially ordered data structure that uses the first in, first out (**FIFO**) principle: items are removed from a queue in the order in which they were inserted. There are many everyday examples of queues, including shoppers waiting in a checkout line at a store and cars waiting in line at a traffic signal. Queues are also quite common in operating systems—jobs that are sent to a printer are typically printed in the order in which they were submitted, for example. As we shall see in Chapter 5, tasks that are waiting to be run on an available CPU are often organized in queues.

1.9.2 Trees

A **tree** is a data structure that can be used to represent data hierarchically. Data values in a tree structure are linked through parent–child relationships. In a **general tree**, a parent may have an unlimited number of children. In a **binary tree**, a parent may have at most two children, which we term the ***left child*** and the ***right child***. A **binary search tree** additionally requires an ordering between the parent's two children in which *left child* <= *right child*. Figure 1.20 provides an example of a binary search tree. When we search for an item in a binary search tree, the worst-case performance is $O(n)$ (consider how this can occur). To remedy this situation, we can use an algorithm to create a **balanced binary search tree**. Here, a tree containing n items has at most $lg\ n$ levels, thus ensuring worst-case performance of $O(lg\ n)$. We shall see in Section 5.7.1 that Linux uses a balanced binary search tree (known as a **red-black tree**) as part its CPU-scheduling algorithm.

1.9.3 Hash Functions and Maps

A **hash function** takes data as its input, performs a numeric operation on the data, and returns a numeric value. This numeric value can then be used as an index into a table (typically an array) to quickly retrieve the data. Whereas searching for a data item through a list of size n can require up to $O(n)$ compar isons, using a hash function for retrieving data from a table can be as good as $O(1)$, depending on implementation details. Because of this performance, hash functions are used extensively in operating systems.

One potential difficulty with hash functions is that two unique inputs can result in the same output value— that is, they can link to the same table

Figure 1.20 Binary search tree.

location. We can accommodate this ***hash collision*** by having a linked list at the table location that contains all of the items with the same hash value. Of course, the more collisions there are, the less efficient the hash function is.

One use of a hash function is to implement a **hash map**, which associates (or ***maps***) [key:value] pairs using a hash function. Once the mapping is established, we can apply the hash function to the key to obtain the value from the hash map (Figure 1.21). For example, suppose that a user name is mapped to a password. Password authentication then proceeds as follows: a user enters her user name and password. The hash function is applied to the user name, which is then used to retrieve the password. The retrieved password is then compared with the password entered by the user for authentication.

1.9.4 Bitmaps

A**bitmap** is a string of *n* binary digits that can be used to represent the status of *n* items. For example, suppose we have several resources, and the availability of each resource is indicated by the value of a binary digit: 0 means that the resource is available, while 1 indicates that it is unavailable (or vice versa). The

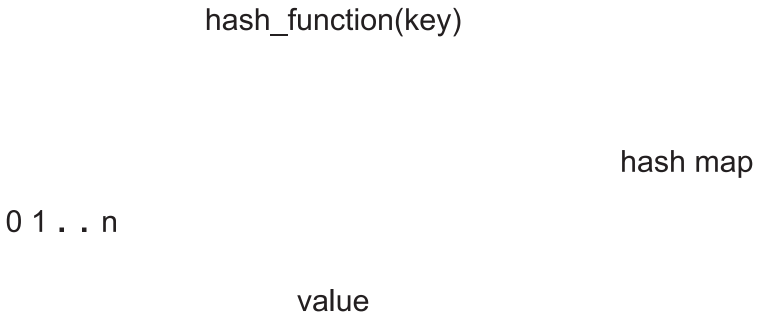


Figure 1.21 Hash map.

LINUX KERNEL DATA STRUCTURES

The data structures used in the Linux kernel are available in the kernel source code. The ***include*** file `<linux/list.h>` provides details of the linked-list data structure used throughout the kernel. A queue in Linux is known as a `kfifo`, and its implementation can be found in the `kfifo.c` file in the `kernel` directory of the source code. Linux also provides a balanced binary search tree implementation using ***red-black trees***. Details can be found in the include file <`linux/rbtree.h`>.

value of the i^{th} position in the bitmap is associated with the i^{th} resource. As an example, consider the bitmap shown below:

001011101

Resources 2, 4, 5, 6, and 8 are unavailable; resources 0, 1, 3, and 7 are available. The power of bitmaps becomes apparent when we consider their space efficiency. If we were to use an eight-bit Boolean value instead of a single bit, the resulting data structure would be eight times larger. Thus, bitmaps are commonly used when there is a need to represent the availability of a large number of resources. Disk drives provide a nice illustration. A medium-sized disk drive might be divided into several thousand individual units, called **disk blocks**. A bitmap can be used to indicate the availability of each disk block. In summary, data structures are pervasive in operating system implementations. Thus, we will see the structures discussed here, along with others, throughout this text as we explore kernel algorithms and their implementations.

1.10 Computing Environments

So far, we have briefly described several aspects of computer systems and the operating systems that manage them. We turn now to a discussion of how operating systems are used in a variety of computing environments.

1.10.1 Traditional Computing

As computing has matured, the lines separating many of the traditional computing environments have blurred. Consider the "typical office environment." Just a few years ago, this environment consisted of PCs connected to a network, with servers providing file and print services. Remote access was awkward, and portability was achieved by use of laptop computers.

Today, web technologies and increasing WAN bandwidth are stretching the boundaries of traditional computing. Companies establish **portals**, which pro vide web accessibility to their internal servers. **Network computers** (or **thin clients**)—which are essentially terminals that understand web-based comput ing—are used in place of traditional workstations where more security or easier maintenance is desired. Mobile computers can synchronize with PCs to allow very portable use of company information. Mobile devices can also

connect to **wireless networks** and cellular data networks to use the company's web portal (as well as the myriad other web resources).

At home, most users once had a single computer with a slow modem connection to the office, the Internet, or both. Today, network-connection speeds once available only at great cost are relatively inexpensive in many places, giving home users more access to more data. These fast data connections are allowing home computers to serve up web pages and to run networks that include printers, client PCs, and servers. Many homes use **firewall** to pro tect their networks from security breaches. Firewalls limit the communications between devices on a network.

In the latter half of the 20th century, computing resources were relatively scarce. (Before that, they were nonexistent!) For a period of time, systems were either batch or interactive. Batch systems processed jobs in bulk, with prede termined input from files or other data sources. Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. These time-sharing systems used a timer and scheduling algorithms to cycle processes rapidly through the CPU, giving each user a share of the resources.

Traditional time-sharing systems are rare today. The same scheduling tech nique is still in use on desktop computers, laptops, servers, and even mobile computers, but frequently all the processes are owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time. Consider the windows created while a user is working on a PC, for example, and the fact that they may be performing different tasks at the same time. Even a web browser can be composed of multiple processes, one for each website currently being visited, with time sharing applied to each web browser process.

1.10.2 Mobile Computing

Mobile computing refers to computing on handheld smartphones and tablet computers. These devices share the distinguishing physical features of being portable and lightweight. Historically, compared with desktop and laptop computers, mobile systems gave up screen size, memory capacity, and overall functionality in return for handheld mobile access to services such as e-mail and web browsing. Over the past few years, however, features on mobile devices have become so rich that the distinction in functionality between, say, a consumer laptop and a tablet computer may be difficult to discern. In fact, we might argue that the features of a contemporary mobile device allow it to provide functionality that is either unavailable or impractical on a desktop or laptop computer.

Today, mobile systems are used not only for e-mail and web browsing but also for playing music and video, reading digital books, taking photos, and recording and editing high-definition video. Accordingly, tremendous growth continues in the wide range of applications that run on such devices. Many developers are now designing applications that take advantage of the unique features of mobile devices, such as global positioning system (GPS) chips, accelerometers, and gyroscopes. An embedded GPS chip allows a mobile device to use satellites to determine its precise location on Earth. That functionality is

especially useful in designing applications that provide navigation— for exam ple, telling users which way to walk or drive or perhaps directing them to nearby services, such as restaurants. An accelerometer allows a mobile device to detect its orientation with respect to the ground and to detect certain other forces, such as tilting and shaking. In several computer games that employ accelerometers, players interface with the system not by using a mouse or a keyboard but rather by tilting, rotating, and shaking the mobile device! Perhaps more a practical use of these features is found in ***augmented-reality*** appli cations, which overlay information on a display of the current environment. It is difficult to imagine how equivalent applications could be developed on traditional laptop or desktop computer systems.

To provide access to on-line services, mobile devices typically use either IEEE standard 802.11 wireless or cellular data networks. The memory capacity and processing speed of mobile devices, however, are more limited than those of PCs. Whereas a smartphone or tablet may have 256 GB in storage, it is not uncommon to find 8 TB in storage on a desktop computer. Similarly, because power consumption is such a concern, mobile devices often use processors that are smaller, are slower, and offer fewer processing cores than processors found on traditional desktop and laptop computers.

Two operating systems currently dominate mobile computing: **Apple iOS** and **Google Android**. iOS was designed to run on Apple iPhone and iPad mobile devices. Android powers smartphones and tablet computers available from many manufacturers. We examine these two mobile operating systems in further detail in Chapter 2.

1.10.3 Client –Server Computing

Contemporary network architecture features arrangements in which **server systems** satisfy requests generated by **client systems**. This form of specialized distributed system, called a **client–server** system, has the general structure depicted in Figure 1.22.

Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server running a database that responds to client requests for data is an example of such a system.

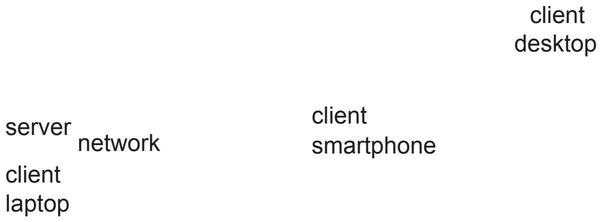


Figure 1.22 General structure of a client – server system.

- The **file-serve system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers. The actual contents of the files can vary greatly, ranging from traditional web pages to rich multimedia content such as high-definition video.

1.10.4 Peer-to-Peer Computing

Another structure for a distributed system is the peer-to-peer (P2P) system model. In this model, clients and servers are not distinguished from one another. Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client–server systems. In a client–server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network. Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- An alternative scheme uses no centralized lookup service. Instead, a peer acting as a client must discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a ***discovery protocol*** must be pro vided that allows peers to discover services provided by other peers in the network. Figure 1.23 illustrates such a scenario.

Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella, that enabled peers to exchange files with one another. The Napster system used an approach simi lar to the first type described above: a centralized server maintained an index of all files stored on peer nodes in the Napster network, and the actual exchange of files took place between the peer nodes. The Gnutella system used a tech nique similar to the second type: a client broadcast file requests to other nodes in the system, and nodes that could service the request responded directly to the client. Peer-to-peer networks can be used to exchange copyrighted mate rials (music, for example) anonymously, and there are laws governing the distribution of copyrighted material. Notably, Napster ran into legal trouble for copyright infringement, and its services were shut down in 2001. For this reason, the future of exchanging files remains uncertain.

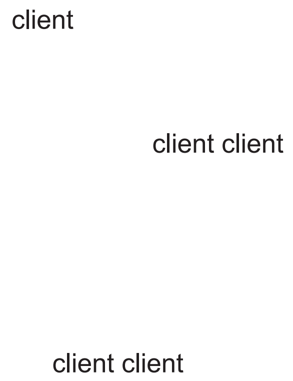


Figure 1.23 Peer-to-peer system with no centralized service.

Skype is another example of peer-to-peer computing. It allows clients to make voice calls and video calls and to send text messages over the Internet using a technology known as **voice over IP** (**VoIP**). Skype uses a hybrid peer to-peer approach. It includes a centralized login server, but it also incorporates decentralized peers and allows two peers to communicate.

1.10.5 Cloud Computing

Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it's a logical extension of virtualization, because it uses virtualization as a base for its functionality. For example, the Amazon Elastic Compute Cloud (**ec2**) facility has thousands of servers, millions of virtual machines, and petabytes of storage available for use by anyone on the Internet. Users pay per month based on how much of those resources they use. There are actually many types of cloud computing, including the following:

- **Public cloud**—a cloud available via the Internet to anyone willing to pay for the services
- **Private cloud**—a cloud run by a company for that company's own use
- **Hybrid cloud**—a cloud that includes both public and private cloud com ponents
- Software as a service (**SaaS**)—one or more applications (such as word processors or spreadsheets) available via the Internet
- Platform as a service (**PaaS**)—a software stack ready for application use via the Internet (for example, a database server)
- Infrastructure as a service (**IaaS**)—servers or storage available over the Internet (for example, storage available for making backup copies of pro duction data)

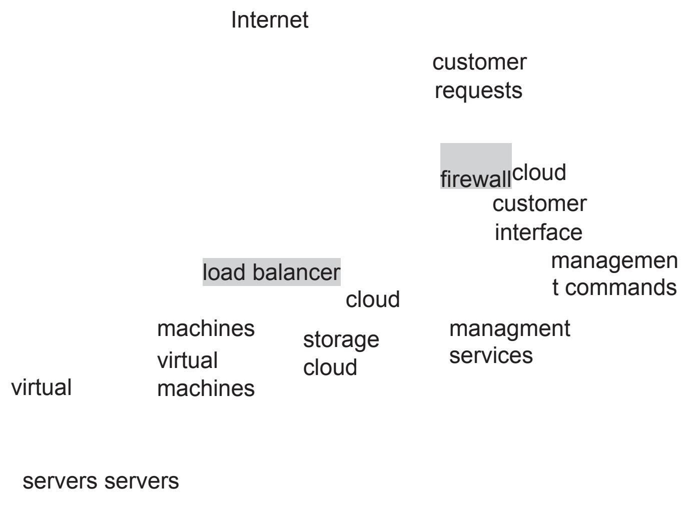


Figure 1.24 Cloud computing.

These cloud-computing types are not discrete, as a cloud computing environ ment may provide a combination of several types. For example, an organiza tion may provide both SaaS and IaaS as publicly available services.

Certainly, there are traditional operating systems within many of the types of cloud infrastructure. Beyond those are the VMMs that manage the virtual machines in which the user processes run. At a higher level, the VMMs them selves are managed by cloud management tools, such as VMware vCloud Director and the open-source Eucalyptus toolset. These tools manage the resources within a given cloud and provide interfaces to the cloud components, making a good argument for considering them a new type of operating system.

Figure 1.24 illustrates a public cloud providing IaaS. Notice that both the cloud services and the cloud user interface are protected by a firewall.

1.10.6 Real-Time Embedded Systems

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to optical drives and microwave ovens. They tend to have very specific tasks. The systems they run on are usually primitive, and so the operating systems provide limited features. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

These embedded systems vary considerably. Some are general-purpose computers, running standard operating systems—such as Linux—with special-purpose applications to implement the functionality. Others are hardware devices with a special-purpose embedded operating system providing just the functionality desired. Yet others are hardware devices

with application-specific integrated circuits (**ASICs**) that perform their tasks without an operating system.

The use of embedded systems continues to expand. The power of these devices, both as standalone units and as elements of networks and the web, is sure to increase as well. Even now, entire houses can be computerized, so that a central computer—either a general-purpose computer or an embedded system —can control heating and lighting, alarm systems, and even coffee makers. Web access can enable a home owner to tell the house to heat up before she arrives home. Someday, the refrigerator will be able to notify the grocery store when it notices the milk is gone.

Embedded systems almost always run **real-time operating systems**. A real time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The com puter must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing ***must*** be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt ***after*** it had smashed into the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this sys tem with a traditional laptop system where it is desirable (but not mandatory) to respond quickly.

In Chapter 5, we consider the scheduling facility needed to implement real time functionality in an operating system, and in Chapter 20 we describe the real-time components of Linux.

1.11 Free and Open-Source Operating Systems

The study of operating systems has been made easier by the avail ability of a vast number of free software and open-source releases. Both **free operating systems** and **open-source operating systems** are available in source-code format rather than as compiled binary code. Note, though, that free software and open-source software are two different ideas championed by different groups of people (see http://gnu.org/philosophy/open-source-misses-the-point.html/ for a discussion on the topic). Free software (sometimes referred to as ***free/libre software***) not only makes source code available but also is licensed to allow no-cost use, redistribution, and modification. Open-source software does not necessarily offer such licensing. Thus, although all free software is open source, some open-source software is not "free." GNU/Linux is the most famous open-source operating system, with some distributions free and others open source only (http://www.gnu.org/distros/). Microsoft Windows is a well-known example of the opposite **closed-source** approach. Windows is **proprietary** software—Microsoft owns it, restricts its use, and carefully protects its source code. Apple's macOS operating system comprises a hybrid

approach. It contains an open-source kernel named Darwin but includes proprietary, closed-source components as well.

Starting with the source code allows the programmer to produce binary code that can be executed on a system. Doing the opposite—**reverse engineering** the source code from the binaries—is quite a lot of work, and useful items such as comments are never recovered. Learning operating systems by examining the source code has other benefits as well. With the source code in hand, a student can modify the operating system and then compile and run the code to try out those changes, which is an excellent learning tool. This text includes projects that involve modifying operating-system source code, while also describing algorithms at a high level to be sure all important operating-system topics are covered. Throughout the text, we provide pointers to examples of open-source code for deeper study.

There are many benefits to open-source operating systems, including a community of interested (and usually unpaid) programmers who contribute to the code by helping to write it, debug it, analyze it, provide support, and sug gest changes. Arguably, open-source code is more secure than closed-source code because many more eyes are viewing the code. Certainly, open-source code has bugs, but open-source advocates argue that bugs tend to be found and fixed faster owing to the number of people using and viewing the code. Companies that earn revenue from selling their programs often hesitate to open-source their code, but Red Hat and a myriad of other companies are doing just that and showing that commercial companies benefit, rather than suffer, when they open-source their code. Revenue can be generated through support contracts and the sale of hardware on which the software runs, for example.

1.11.1 History

In the early days of modern computing (that is, the 1950s), software generally came with source code. The original hackers (computer enthusiasts) at MIT's Tech Model Railroad Club left their programs in drawers for others to work on. "Homebrew" user groups exchanged code during their meetings. Company specific user groups, such as Digital Equipment Corporation's DECUS, accepted contributions of source-code programs, collected them onto tapes, and dis tributed the tapes to interested members. In 1970, Digital's operating systems were distributed as source code with no restrictions or copyright notice.

Computer and software companies eventually sought to limit the use of their software to authorized computers and paying customers. Releasing only the binary files compiled from the source code, rather than the source code itself, helped them to achieve this goal, as well as protecting their code and their ideas from their competitors. Although the Homebrew user groups of the 1970s exchanged code during their meetings, the operating systems for hobbyist machines (such as CPM) were proprietary. By 1980, proprietary software was the usual case.

1.11.2 Free Operating Systems

To counter the move to limit software use and redistribution, Richard Stallman in 1984 started developing a free, UNIX-compatible operating system called GNU(which is a recursive acronym for "GNU's Not Unix!"). To Stallman, "free" refers to freedom of use, not price. The free-software movement does not object

to trading a copy for an amount of money but holds that users are entitled to four certain freedoms: (1) to freely run the program, (2) to study and change the source code, and to give or sell copies either (3) with or (4) without changes. In 1985, Stallman published the GNU Manifesto, which argues that all software should be free. He also formed the **Free Software Foundation** (**FSF**) with the goal of encouraging the use and development of free software.

The FSF uses the copyrights on its programs to implement "copyleft," a form of licensing invented by Stallman. Copylefting a work gives anyone that possesses a copy of the work the four essential freedoms that make the work free, with the condition that redistribution must preserve these freedoms. The **GNU General Public License** (**GPL**) is a common license under which free software is released. Fundamentally, the GPL requires that the source code be distributed with any binaries and that all copies (including modified versions) be released under the same GPL license. The Creative Commons "Attribution Sharealike" license is also a copyleft license; "sharealike" is another way of stating the idea of copyleft.

1.11.3 GNU/Linux

As an example of a free and open-source operating system, consider **GNU/Linux**. By 1991, the GNU operating system was nearly complete. The GNU Project had developed compilers, editors, utilities, libraries, and games — whatever parts it could not find elsewhere. However, the GNU kernel never became ready for prime time. In 1991, a student in Finland, Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide. The advent of the Internet meant that anyone interested could download the source code, modify it, and submit changes to Torvalds. Releasing updates once a week allowed this so-called "Linux" operating system to grow rapidly, enhanced by several thousand programmers. In 1991, Linux was not free software, as its license permitted only noncommercial redistribution. In 1992, however, Torvalds rereleased Linux under the GPL, making it free software (and also, to use a term coined later, "open source").

The resulting GNU/Linux operating system (with the kernel properly called Linux but the full operating system including GNU tools called GNU/Linux) has spawned hundreds of unique **distributions**, or custom builds, of the system. Major distributions include Red Hat, SUSE, Fedora, Debian, Slackware, and Ubuntu. Distributions vary in function, utility, installed applications, hardware support, user interface, and purpose. For example, Red Hat Enterprise Linux is geared to large commercial use. PCLinuxOS is a **live CD**—an operating system that can be booted and run from a CD-ROM without being installed on a system's boot disk. A variant of PCLinuxOS—called PCLinuxOS Supergamer DVD—is a **live DVD** that includes graphics drivers and games. A gamer can run it on any compatible system simply by booting from the DVD. When the gamer is finished, a reboot of the system resets it to its installed operating system.

You can run Linux on a Windows (or other) system using the following simple, free approach:

1. Download the free Virtualbox VMM tool from

   https://www.virtualbox.org/

   and install it on your system.

2. Choose to install an operating system from scratch, based on an installation image like a CD, or choose pre-built operating-system images that can be installed and run more quickly from a site like

   http://virtualboxes.org/images/

   These images are preinstalled with operating systems and applications and include many flavors of GNU/Linux.

3. Boot the virtual machine within Virtualbox.

An alternative to using Virtualbox is to use the free program Qemu (http://wiki.qemu.org/Download/), which includes the qemu-img command for converting Virtualbox images to Qemu images to easily import them.

With this text, we provide a virtual machine image of GNU/Linux running the Ubuntu release. This image contains the GNU/Linux source code as well as tools for software development. We cover examples involving the GNU/Linux image throughout this text, as well as in a detailed case study in Chapter 20.

1.11.4 BSD UNIX

BSD UNIX has a longer and more complicated history than Linux. It started in 1978 as a derivative of AT&T's UNIX. Releases from the University of California at Berkeley (UCB) came in source and binary form, but they were not open source because a license from AT&T was required. BSD UNIX's development was slowed by a lawsuit by AT&T, but eventually a fully functional, open-source version, 4.4BSD-lite, was released in 1994.

Just as with Linux, there are many distributions of BSD UNIX, including FreeBSD, NetBSD, OpenBSD, and DragonflyBSD. To explore the source code of FreeBSD, simply download the virtual machine image of the version of interest and boot it within Virtualbox, as described above for Linux. The source code comes with the distribution and is stored in /usr/src/. The kernel source code is in /usr/src/sys. For example, to examine the vir tual memory implementation code in the FreeBSD kernel, see the files in /usr/src/sys/vm. Alternatively, you can simply view the source code online at https://svnweb.freebsd.org.

As with many open-source projects, this source code is contained in and controlled by a **version control system**—in this case, "subversion" (https://subversion.apache.org/source-code). Version control systems allow a user to "pull" an entire source code tree to his computer and "push" any changes back into the repository for others to then pull. These systems also provide other features, including an entire history of each file and a conflict resolution feature in case the same file is changed concurrently. Another

version control system is **git**, which is used for GNU/Linux, as well as other programs (http://www.git-scm.com).

Darwin, the core kernel component of macOS, is based on BSD UNIX and is open-sourced as well. That source code is available from http://www.opensource.apple.com/. Every macOS release has its open-source components posted at that site. The name of the package that contains the kernel begins with "xnu." Apple also provides extensive developer tools, documentation, and support at http://developer.apple.com.

THE STUDY OF OPERATING SYSTEMS

There has never been a more interesting time to study operating systems, and it has never been easier. The open-source movement has overtaken operating systems, causing many of them to be made available in both source and binary (executable) format. The list of operating systems available in both formats includes Linux, BSD UNIX, Solaris, and part of macOS. The availability of source code allows us to study operating systems from the inside out. Questions that we could once answer only by looking at documentation or the behavior of an operating system we can now answer by examining the code itself.

Operating systems that are no longer commercially viable have been open-sourced as well, enabling us to study how systems operated in a time of fewer CPU, memory, and storage resources. An extensive but incomplete list of open-source operating-system projects is available from http://dmoz.org/Computers/Software/Operating Systems/Open Source/.

In addition, the rise of virtualization as a mainstream (and frequently free) computer function makes it possible to run many operating systems on top of one core system. For example, VMware (http://www.vmware.com) provides a free "player" for Windows on which hundreds of free "virtual appliances" can run. Virtualbox (http://www.virtualbox.com) provides a free, open-source virtual machine manager on many operating systems. Using such tools, students can try out hundreds of operating systems without dedicated hardware.

In some cases, simulators of specific hardware are also available, allowing the operating system to run on "native" hardware, all within the confines of a modern computer and modern operating system. For example, a DECSYSTEM-20 simulator running on macOS can boot TOPS-20, load the source tapes, and modify and compile a new TOPS-20 kernel. An interested student can search the Internet to find the original papers that describe the operating system, as well as the original manuals.

The advent of open-source operating systems has also made it easier to make the move from student to operating-system developer. With some knowledge, some effort, and an Internet connection, a student can even create a new operating-system distribution. Not so many years ago, it was difficult or impossible to get access to source code. Now, such access is limited only by how much interest, time, and disk space a student has.

1.11.5 Solaris

Solaris is the commercial UNIX-based operating system of Sun Microsystems. Originally, Sun's **SunOS** operating system was based on BSD UNIX. Sun moved to AT&T's System V UNIX as its base in 1991. In 2005, Sun open-sourced most of the Solaris code as the OpenSolaris project. The purchase of Sun by Oracle in 2009, however, left the state of this project unclear.

Several groups interested in using OpenSolaris have expanded its features, and their working set is Project Illumos, which has expanded from the Open Solaris base to include more features and to be the basis for several products. Illumos is available at http://wiki.illumos.org.

1.11.6 Open-Source Systems as Learning Tools

The free-software movement is driving legions of programmers to create thousands of open-source projects, including operating systems. Sites like http://freshmeat.net/ and http://distrowatch.com/ provide portals to many of these projects. As we stated earlier, open-source projects enable students to use source code as a learning tool. They can modify programs and test them, help find and fix bugs, and otherwise explore mature, full-featured operating systems, compilers, tools, user interfaces, and other types of programs. The availability of source code for historic projects, such as Multics, can help students to understand those projects and to build knowledge that will help in the implementation of new projects.

Another advantage of working with open-source operating systems is their diversity. GNU/Linux and BSD UNIX are both open-source operating systems, for instance, but each has its own goals, utility, licensing, and purpose. Some times, licenses are not mutually exclusive and cross-pollination occurs, allow ing rapid improvements in operating-system projects. For example, several major components of OpenSolaris have been ported to BSD UNIX. The advan tages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

1.12 Summary

- An operating system is software that manages the computer hardware, as well as providing an environment for application programs to run.

- Interrupts are a key way in which hardware interacts with the operating system. A hardware device triggers an interrupt by sending a signal to the CPU to alert the CPU that some event requires attention. The interrupt is managed by the interrupt handler.

- For a computer to do its job of executing programs, the programs must be in main memory, which is the only large storage area that the processor can access directly.

- The main memory is usually a volatile storage device that loses its contents when power is turned off or lost.

- Nonvolatile storage is an extension of main memory and is capable of holding large quantities of data permanently.

- The most common nonvolatile storage device is a hard disk, which can provide storage of both programs and data.

- The wide variety of storage systems in a computer system can be organized in a hierarchy according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.

- Modern computer architectures are multiprocessor systems in which each CPU contains several computing cores.

- To best utilize the CPU, modern operating systems employ multiprogram ming, which allows several jobs to be in memory at the same time, thus ensuring that the CPU always has a job to execute.

- Multitasking is an extension of multiprogramming wherein CPU schedul ing algorithms rapidly switch between processes, providing users with a fast response time.

- To prevent user programs from interfering with the proper operation of the system, the system hardware has two modes: user mode and kernel mode.

- Various instructions are privileged and can be executed only in kernel mode. Examples include the instruction to switch to kernel mode, I/O control, timer management, and interrupt management.

- A process is the fundamental unit of work in an operating system. Pro cess management includes creating and deleting processes and providing mechanisms for processes to communicate and synchronize with each other.

- An operating system manages memory by keeping track of what parts of memory are being used and by whom. It is also responsible for dynami cally allocating and freeing memory space.

- Storage space is managed by the operating system; this includes providing file systems for representing files and directories and managing space on mass-storage devices.

- Operating systems provide mechanisms for protecting and securing the operating system and users. Protection measures control the access of processes or users to the resources made available by the computer system.

- Virtualization involves abstracting a computer's hardware into several different execution environments.
- Data structures that are used in an operating system include lists, stacks, queues, trees, and maps.
- Computing takes place in a variety of environments, including traditional computing, mobile computing, client–server systems, peer-to-peer systems, cloud computing, and real-time embedded systems.