Operating System Concepts

TENTH EDITION

ABRAHAM SILBERSCHATZ • PETER BAER GALVIN • GREG GAGNE

WILEY

OPERATING SYSTEM CONCEPTS

TENTH EDITION

OPERATING SYSTEM CONCEPTS

ABRAHAM SILBERSCHATZ
Yale University

PETER BAER GALVIN
Cambridge Computer and Starfish Storage

GREG GAGNE
Westminster College

TENTH EDITION

WILEY

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 automobiles 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 predominantly 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 guidelines 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 Computing 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, "Further 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 Exercises" (solutions available online), and "Further Reading" sections. Instructors 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 chapters. 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 multicore 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, Synchronization 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 computer. 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 online storage of and access to both data and programs. We describe the classic 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 enforcement. 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 including the gcc and Java compilers. Most of the programming assignments in the book can be completed using this virtual machine, with the exception of assignments 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 bringing older material up to date and removing material that is no longer interesting 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 modifications 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 manage-

ment 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 NUMA systems and global allocation using a free-frame list. New coverage includes compressed 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 vulnerabilities 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 TrustZones 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 distributed 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 various 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 operating 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 programming 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 system, 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 available 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 unfamiliar 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 Eskicioğlu, 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 Özden, 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 throughout the text, as well as answering many of our Android-related questions.
  - Appendix B was written by Dave Probert and was derived from Chapter 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 coverage 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 consisted 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

Chapter 4 Threads & Concurrency

Chapter 5 CPU Scheduling

PART THREE ■ PROCESS SYNCHRONIZATION

Chapter 6 Synchronization Tools

Chapter 7 Synchronization Examples

Chapter 8 Deadlocks

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

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

Part One

Overview

An *operating system* acts as an intermediary between the user of a computer 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 hardware. 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 created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions.

CHAPTER 1

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 environments.

In order to explore the role of an operating system in a modern computing environment, it is important first to understand the organization and architecture 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 environments.
- 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.

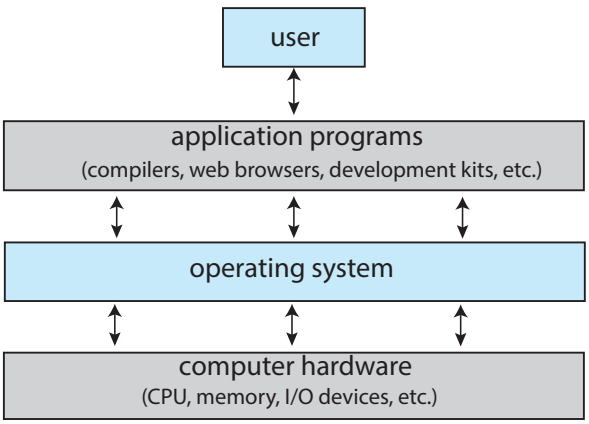

Figure 1.1 Abstract view of the components of a computer system.

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 systems 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. Computing 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 vendor ships when you order "the operating system." The features included, however, 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 common 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 necessarily 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 systems 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

WHY STUDY OPERATING SYSTEMS?

Although there are many practitioners of computer science, only a small percentage of them will be involved in the creation or modification of an operating 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.

a core kernel along with middleware that supports databases, multimedia, and graphics (to name only a few).

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 controller. 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.

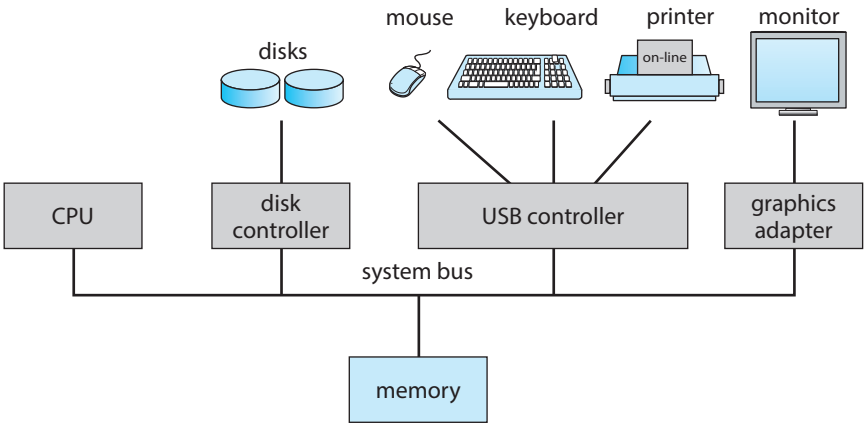

Figure 1.2 A typical PC computer system.

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 registers 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,

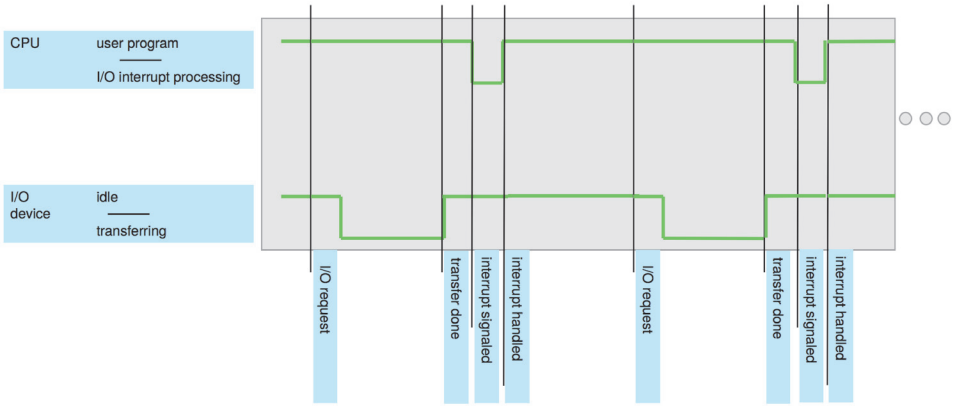

Figure 1.3 Interrupt timeline for a single program doing output.

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**.