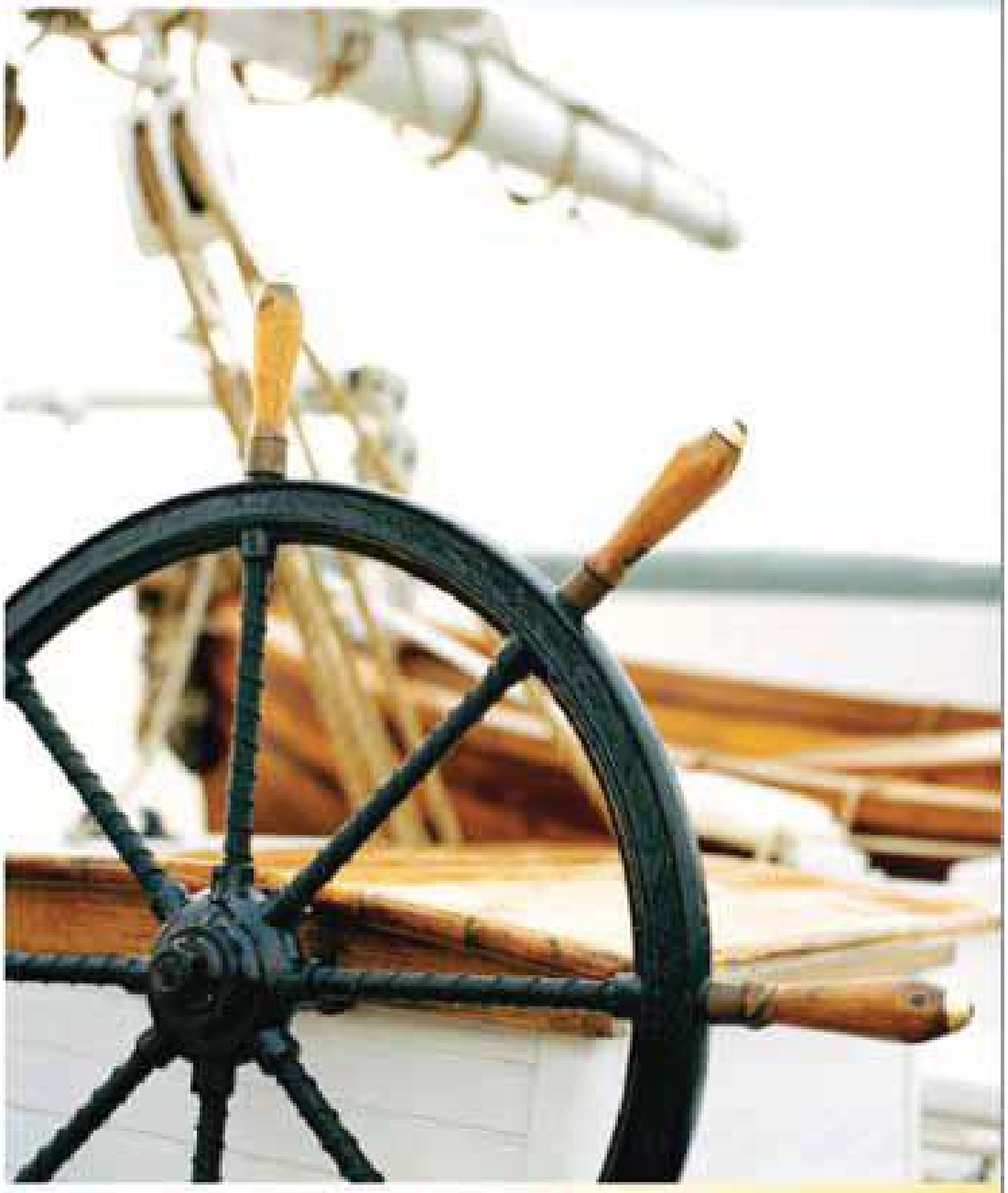
OPERATING SYSTEMS

Internals and Design Principles

Seventh Edition



William Stallings

Operating Systems
Internals and Design Principles
Seventh Edition

William Stallings

ISBN 1-256-52023-3

Prentice Hall

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director: Marcia Horton
Editor in Chief: Michael Hirsch
Executive Editor: Tracy Dunkelberger
Assistant Editor: Melinda Haggerty
Editorial Assistant: Allison Michael
Director of Marketing: Patrice Jones
Marketing Manager: Yezan Alayan
SenioMarketing Coordinator: Kathryn Ferranti
Production Manager: Pat Brown

Art Director: Jayne Conte
Cover Designer: Bruce Kenselaar
Media Director: Daniel Sandin
Media Project Manager: Wanda Rockwell
Full-Service Project Management/Composition: Shiny Rajesh/Integra Software Service Pvt. Ltd.
Interior Printer/Bindery: Edwards Brothers
Cover Printer: Lehigh-Phoenix Color

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on appropriate page within text.

Library of Congress Cataloging-in-Publication Data

Stallings, William.
Operating systems : internals and design principles / William Stallings. — 7th ed.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-13-230998-1 (alk. paper)
ISBN-10: 0-13-230998-X (alk. paper)
1. Operating systems (Computers) I. Title.
QA76.76.O63S733 2011
005.4'3 dc22

2010048597

10 9 8 7 6 5 4 3 2 1—EB—15 14 13 12 11

Prentice Hall
is an imprint of

PEARSON

www.pearsonhighered.com

ISBN 10: 0-13-230998-X
ISBN 13: 978-0-13-230998-1

ISBN 1-256-52023-3

To my brilliant and brave wife,
Antigone Tricia, who has survived
the worst horrors imaginable.

ISBN 1-256-52023-3

CONTENTS

ISBN 1-256-52023-3

ISBN 1-256-52023-3

ISBN 1-256-52023-3

ISBN 1-256-52023-3

ONLINE CHAPTERS AND APPENDICES[1]

[1]Online chapters, appendices, and other documents are Premium Content, available via the access card at the front of this book.

ISBN 1-256-52023-3

ISBN 1-256-52023-3

ONLINE RESOURCES

Site	Location	Description
Companion Web Site	williamstallings.com/OS/OS7e.html www.pearsonhighered.com/stallings/	*Student Resources* button: Useful links and documents for students *Instructor Resources* button: Useful links and documents for instructors
Premium Web Content	www.pearsonhighered.com/stallings/, click on Premium Web Content button and enter the student access code found on the card in the front of the book.	Online chapters, appendices, and other documents that supplement the book
Instructor Resource Center (IRC)	Pearsonhighered.com/Stallings/, click on Instructor Resource button.	Solutions manual, projects manual, slides, and other useful documents
Computer Science Student Resource Site	computersciencestudent.com	Useful links and documents for computer science students

ISBN 1-256-52023-3

PREFACE

This book does not pretend to be a comprehensive record; but it aims at helping to disentangle from an immense mass of material the crucial issues and cardinal decisions. Throughout I have set myself to explain faithfully and to the best of my ability.

—*THE WORLD CRISIS*, WINSTON CHURCHILL

OBJECTIVES

This book is about the concepts, structure, and mechanisms of operating systems. Its purpose is to present, as clearly and completely as possible, the nature and characteristics of modern-day operating systems.

This task is challenging for several reasons. First, there is a tremendous range and variety of computer systems for which operating systems are designed. These include embedded systems, smart phones, single-user workstations and personal computers, medium-sized shared systems, large mainframe and supercomputers, and specialized machines such as real-time systems. The variety is not just in the capacity and speed of machines, but in applications and system support requirements as well. Second, the rapid pace of change that has always characterized computer systems continues with no letup. A number of key areas in operating system design are of recent origin, and research into these and other new areas continues.

In spite of this variety and pace of change, certain fundamental concepts apply consistently throughout. To be sure, the application of these concepts depends on the current state of technology and the particular application requirements. The intent of this book is to provide a thorough discussion of the fundamentals of operating system design and to relate these to contemporary design issues and to current directions in the development of operating systems.

EXAMPLE SYSTEMS

This text is intended to acquaint the reader with the design principles and implementation issues of contemporary operating systems. Accordingly, a purely conceptual or theoretical treatment would be inadequate. To illustrate the concepts and to tie them to real-world design choices that must be made, three operating systems have been chosen as running examples:

- **Windows 7:** A multitasking operating system for personal computers, workstations, and servers. This operating system incorporates many of the latest developments in operating system technology. In addition, Windows is one of the first important commercial operating systems to rely heavily on

ISBN 1-256-52023-3

object-oriented design principles. This book covers the technology used in the most recent version of Windows, known as Windows 7.

- **UNIX:** A multiuser operating system, originally intended for minicomputers, but implemented on a wide range of machines from powerful microcomputers to supercomputers. Several flavors of UNIX are included as examples. FreeBSD is a widely used system that incorporates many state-of-the-art features. Solaris is a widely used commercial version of UNIX.
- **Linux:** An open-source version of UNIX that is now widely used.

These systems were chosen because of their relevance and representativeness. The discussion of the example systems is distributed throughout the text rather than assembled as a single chapter or appendix. Thus, during the discussion of concurrency, the concurrency mechanisms of each example system are described, and the motivation for the individual design choices is discussed. With this approach, the design concepts discussed in a given chapter are immediately reinforced with real-world examples.

INTENDED AUDIENCE

The book is intended for both an academic and a professional audience. As a textbook, it is intended as a one-semester undergraduate course in operating systems for computer science, computer engineering, and electrical engineering majors. It covers all of the core topics and most of the elective topics recommended in *Computer Science Curriculum 2008*, from the Joint Task Force on Computing Curricula of the IEEE Computer Society and the ACM, for the Undergraduate Program in Computer Science. The book also covers the operating systems topics recommended in the *Guidelines for Associate-Degree Curricula in Computer Science 2002*, also from the Joint Task Force on Computing Curricula of the IEEE Computer Society and the ACM. The book also serves as a basic reference volume and is suitable for self-study.

PLAN OF THE TEXT

The book is divided into eight parts (see Chapter 0 for an overview):

- Background
- Processes
- Memory
- Scheduling
- Input/output and files
- Embedded systems
- Security
- Distributed systems

ISBN 1-256-52023-3

The book includes a number of pedagogic features, including the use of animations and numerous figures and tables to clarify the discussion. Each chapter includes a list of key words, review questions, homework problems, suggestions for further reading, and recommended Web sites. The book also includes an extensive glossary, a list of frequently used acronyms, and a bibliography. In addition, a test bank is available to instructors.

WHAT'S NEW IN THE SEVENTH EDITION

In the 3 years since the sixth edition of this book was published, the field has seen continued innovations and improvements. In this new edition, I try to capture these changes while maintaining a broad and comprehensive coverage of the entire field. To begin the process of revision, the sixth edition of this book was extensively reviewed by a number of professors who teach the subject and by professionals working in the field. The result is that, in many places, the narrative has been clarified and tightened, and illustrations have been improved. Also, a number of new "field-tested" homework problems have been added.

Beyond these refinements to improve pedagogy and user friendliness, the technical content of the book has been updated throughout, to reflect the ongoing changes in this exciting field, and the instructor and student support has been expanded. The most noteworthy changes are as follows:

- **Windows 7:** Windows 7 is Microsoft's latest OS offering for PCs, workstations, and servers. The seventh edition provides details on Windows 7 internals in all of the key technology areas covered in this book, including process/thread management, scheduling, memory management, security, file systems, and I/O.
- **Multicore operating system issues:** The seventh edition now includes coverage of what has become the most prevalent new development in computer systems: the use of multiple processors on a single chip. At appropriate points in the book, operating system issues related to the use of a multicore organization are explored.
- **Virtual machines:** Chapter 2 now includes a section on virtual machines, which outlines the various approaches that have been implemented commercially.
- **New scheduling examples:** Chapter 10 now includes a discussion of the FreeBSD scheduling algorithm, designed for use with multiprocessor and multicore systems, and Linux VServer scheduling for a virtual machine environment.
- **Service-oriented architecture (SOA):** SOA is a form of client/server architecture that now enjoys widespread use in enterprise systems. SOA is now covered in Chapter 16.
- **Probability, statistics, and queueing analysis:** Two new chapters review key topics in these areas to provide background for OS performance analysis.
- **B-trees:** This is a technique for organizing indexes into files and databases that is commonly used in OS file systems, including those supported by

ISBN 1-256-52023-3

Mac OS X, Windows, and several Linux file systems. B-trees are now covered in Chapter 12.

- **Student study aids:** Each chapter now begins with a list of learning objectives. In addition, a chapter-by-chapter set of **review outlines** highlights key concepts that the student should concentrate on in each chapter.
- **OS/161:** OS/161 is an educational operating system that is becoming increasingly recognized as the teaching platform of choice. This new edition provides support for using OS/161 as an active learning component. See later in this Preface for details.
- **Sample syllabus:** The text contains more material than can be conveniently covered in one semester. Accordingly, instructors are provided with several sample syllabi that guide the use of the text within limited time (e.g., 16 weeks or 12 weeks). These samples are based on real-world experience by professors with the sixth edition.

With each new edition, it is a struggle to maintain a reasonable page count while adding new material. In part, this objective is realized by eliminating obsolete material and tightening the narrative. For this edition, chapters and appendices that are of less general interest have been moved online, as individual PDF files. This has allowed an expansion of material without the corresponding increase in size and price.

STUDENT RESOURCES

For this new edition, a tremendous amount of original supporting material has been made available online, in the following categories
The Companion Web site and student resource material can be reached through the Publisher's Web site www.pearsonhighered.com/stallings or by clicking on the button labeled "Book Info and More Instructor Resources" at the book's Companion Web site WilliamStallings.com/OS/OS7e.html. For this new edition, a tremendous amount of original supporting material has been made available online, in the following categories:

- **Homework problems and solutions:** To aid the student in understanding the material, a separate set of homework problems with solutions are available. These enable the students to test their understanding of the text.
- **Programming projects:** Two major programming projects, one to build a shell (or command line interpreter) and one to build a process dispatcher, are described.
- **Key papers:** Several dozen papers from the professional literature, many hard to find, are provided for further reading.
- **Supporting documents:** A variety of other useful documents are referenced in the text and provided online.

Premium Web Content
Purchasing this textbook new grants the reader 6 months of access to this online material. See the access card in the front of this book for details.

ISBN 1-256-52023-3

- **Online chapters:** To limit the size and cost of the book, four chapters of the book are provided in PDF format. The chapters are listed in this book's table of contents.
- **Online appendices:** There are numerous interesting topics that support material found in the text but whose inclusion is not warranted in the printed text. A total of 13 appendices cover these topics for the interested student. The appendices are listed in this book's table of contents.

INSTRUCTOR SUPPORT MATERIALS

Support materials are available at the Instructor Resource Center (IRC) for this textbook, which can be reached through the Publisher's Web site www.pearsonhighered.com/stallings or by clicking on the button labeled "Book Info and More Instructor Resources" at this book's Companion Web site WilliamStallings.com/OS/OS7e.html. To gain access to the IRC, please contact your local Pearson sales representative via pearsonhighered.com/educator/replocator/requestSalesRep.page or call Pearson Faculty Services at 1-800-526-0485. To support instructors, the following materials are provided:

- **Solutions manual:** Solutions to end-of-chapter Review Questions and Problems.
- **Projects manual:** Suggested project assignments for all of the project categories listed in the next section.
- **PowerPoint slides:** A set of slides covering all chapters, suitable for use in lecturing.
- **PDF files:** Reproductions of all figures and tables from the book.
- **Test bank:** A chapter-by-chapter set of questions.
- Links to Web sites for other courses being taught using this book.
- An Internet mailing list has been set up so that instructors using this book can exchange information, suggestions, and questions with each other and with the author. As soon as typos or other errors are discovered, an errata list for this book will be available at WilliamStallings.com. Sign-up information for this Internet mailing list.
- **Computer science student resource list:** A list of helpful links for computer science students and professionals is provided at ComputerScienceStudent.com, which provides documents, information, and useful links for computer science students and professionals.
- **Programming projects:** Two major programming projects, one to build a shell (or command line interpreter) and one to build a process dispatcher, are described in the online portion of this textbook. The IRC provides further information and step-by-step exercises for developing the programs. As an alternative, the instructor can assign a more extensive series of projects that cover many of the principles in the book. The student is provided with

ISBN 1-256-52023-3

detailed instructions for doing each of the projects. In addition, there is a set of homework problems, which involve questions related to each project for the student to answer.

Projects and Other Student Exercises

For many instructors, an important component of an OS course is a project or set of projects by which the student gets hands-on experience to reinforce concepts from the text. This book provides an unparalleled degree of support for including a projects component in the course. In the online portion of the text, two major programming projects are defined. In addition, the instructor support materials available through Pearson not only include guidance on how to assign and structure the various projects but also includes a set of user's manuals for various project types plus specific assignments, all written especially for this book. Instructors can assign work in the following areas:

- **OS/161 projects:** Described below.
- **Simulation projects:** Described below.
- **Programming projects:** Described below.
- **Research projects:** A series of research assignments that instruct the student to research a particular topic on the Internet and write a report.
- **Reading/report assignments:** A list of papers that can be assigned for reading and writing a report, plus suggested assignment wording.
- **Writing assignments:** A list of writing assignments to facilitate learning the material.
- **Discussion topics:** These topics can be used in a classroom, chat room, or message board environment to explore certain areas in greater depth and to foster student collaboration.

In addition, information is provided on a software package known as BACI that serves as a framework for studying concurrency mechanisms.

This diverse set of projects and other student exercises enables the instructor to use the book as one component in a rich and varied learning experience and to tailor a course plan to meet the specific needs of the instructor and students. See Appendix B in this book for details.

OS/161

New to this edition is support for an active learning component based on OS/161. OS/161 is an educational operating system that is becoming increasingly recognized as the preferred teaching platform for OS internals. It aims to strike a balance between giving students experience in working on a real operating system and potentially overwhelming students with the complexity that exists in a fully fledged operating system, such as Linux. Compared to most deployed operating systems, OS/161 is quite small (approximately 20,000 lines of code and comments), and therefore it is much easier to develop an understanding of the entire code base.

ISBN 1-256-52023-3

The IRC includes:

1. A packaged set of html files that the instructor can upload to a course server for student access.
2. A getting-started manual to be handed out to students to help them begin using OS/161.
3. A set of exercises using OS/161, to be handed out to students.
4. Model solutions to each exercise for the instructor's use.
5. All of this will be cross-referenced with appropriate sections in the book, so that the student can read the textbook material and then do the corresponding OS/161 project.

Simulations for Students and Instructors

The IRC provides support for assigning projects based on a set of seven **simulations** that cover key areas of OS design. The student can use a set of simulation packages to analyze OS design features. The simulators are all written in Java and can be run either locally as a Java application or online through a browser. The IRC includes specific assignments to give to students, telling them specifically what they are to do and what results are expected.

Animations for Students and Instructors

This edition also incorporates animations. Animations provide a powerful tool for understanding the complex mechanisms of a modern OS. A total of 53 animations are used to illustrate key functions and algorithms in OS design. The animations are used for Chapters 3, 5, 6, 7, 8, 9, and 11. For access to the animations, click on the rotating globe at this book's Web site at WilliamStallings.com/OS/OS7e.html.

ACKNOWLEDGMENTS

This new edition has benefited from review by a number of people, who gave generously of their time and expertise. These include Samir Chettri (The University of Maryland, Baltimore County), Michael Rogers (Tennessee Technological University), Glenn Booker (Drexel University), Jeongkyu Lee (University of Bridgeport), Sanjiv Bhatia (University of Missouri, Baltimore County), Martin Barrett (East Tennessee State University), Lubomir Ivanov (Iona College), Bina Ramamurthy (University at Buffalo), Dean Kelley (Minnesota State University), Joel Weinstein (Northeastern University), all of whom reviewed most or all of the book.

Thanks also to the people who provided detailed reviews of a one or more chapters: John South (University of Dallas), Kevin Sanchez-Cherry (IT Security Specilist), Adri Jovin (PG Scholar,Department of IT, Anna University of Technology, Coimbatore), Thriveni Venkatesh (Professor Thriveni T K from GcMAT, Bangalore, India), Fernando Lichtschein (Instituto de Tecnología ORT Argentina), C. Annamala (Indian Institute of Technology Kharagpur),

ISBN 1-256-52023-3

Abdul-Rahman Mahmood (Independent IT & security consultant & Creator of AlphaPeeler crypto tool), and Abhilash V R (VVDN Technologies).

I would also like to thank Dave Probert, Architect in the Windows Core Kernel & Architecture team at Microsoft, for the review of the material on Windows Vista and for providing the comparisons of Linux and Vista; Tigran Aivazian, author of the Linux Kernel Internals document, which is part of the Linux Documentation Project, for the review of the material on Linux 2.6; Nick Garnett of eCosCentric, for the review of the material on eCos; and Philip Levis, one of the developers of TinyOS, for the review of the material on TinyOS.

Professor Andrew Peterson, Ioan Stefanovici, and OS instructors at the University of Toronto prepared the OS/161 supplements for the IRC.

Adam Critchley (University of Texas at San Antonio) developed the simulation exercises. Matt Sparks (University of Illinois at Urbana-Champaign) adapted a set of programming problems for use with this textbook.

Lawrie Brown of the Australian Defence Force Academy produced the material on buffer overflow attacks. Ching-Kuang Shene (Michigan Tech University) provided the examples used in the section on race conditions and reviewed the section. Tracy Camp and Keith Hellman, both at the Colorado School of Mines, developed a new set of homework problems. In addition, Fernando Ariel Gont contributed a number of homework problems; he also provided detailed reviews of all of the chapters.

I would also like to thank Bill Bynum (College of William and Mary) and Tracy Camp (Colorado School of Mines) for contributing Appendix O; Steve Taylor (Worcester Polytechnic Institute) for contributing the programming projects and reading/report assignments in the instructor's manual; and Professor Tan N. Nguyen (George Mason University) for contributing the research projects in the instruction manual. Ian G. Graham (Griffith University) contributed the two programming projects in the textbook. Oskars Rieksts (Kutztown University) generously allowed me to make use of his lecture notes, quizzes, and projects.

Finally, I would like to thank the many people responsible for the publication of the book, all of whom did their usual excellent job. This includes my editor Tracy Dunkelberger, her assistants Carole Snyder, Melinda Hagerty, and Allison Michael, and production manager Pat Brown. I also thank Shiny Rajesh and the production staff at Integra for another excellent and rapid job. Thanks also to the marketing and sales staffs at Pearson, without whose efforts this book would not be in your hands.

With all this assistance, little remains for which I can take full credit. However, I am proud to say that, with no help whatsoever, I selected all of the quotations.

ISBN 1-256-52023-3

About the Author

William Stallings has made a unique contribution to understanding the broad sweep of technical developments in computer security, computer networking, and computer architecture. He has authored 17 titles, and, counting revised editions, a total of 42 books on various aspects of these subjects. His writings have appeared in numerous ACM and IEEE publications, including the *Proceedings of the IEEE* and *ACM Computing Reviews*.

He has 11 times received the award for the best Computer Science textbook of the year from the Text and Academic Authors Association.

In over 30 years in the field, he has been a technical contributor, technical manager, and an executive with several high-technology firms. He has designed and implemented both TCP/IP-based and OSI-based protocol suites on a variety of computers and operating systems, ranging from microcomputers to mainframes. As a consultant, he has advised government agencies, computer and software vendors, and major users on the design, selection, and use of networking software and products.

He has created and maintains the **Computer Science Student Resource Site** at http://www.computersciencestudent.com/. This site provides documents and links on a variety of subjects of general interest to computer science students (and professionals). He is a member of the editorial board of *Cryptologia*, a scholarly journal devoted to all aspects of cryptology.

Dr. Stallings holds a PhD from M.I.T. in Computer Science and a B.S. from Notre Dame in electrical engineering.

ISBN 1-256-52023-3

ISBN 1-256-52023-3

CHAPTER 0

READER'S AND INSTRUCTOR'S GUIDE

ISBN 1-256-52023-3

These delightful records should have been my constant study.

THE IMPORTANCE OF BEING EARNEST, OSCAR WILDE

This book, with its accompanying Web site, covers a lot of material. Here we give the reader some basic background information.

0.1 OUTLINE OF THIS BOOK

The book is organized in eight parts:

Part One. Background: Provides an overview of computer architecture and organization, with emphasis on topics that relate to operating system (OS) design, plus an overview of the OS topics in remainder of the book.

Part Two. Processes: Presents a detailed analysis of processes, multithreading, symmetric multiprocessing (SMP), and microkernels. This part also examines the key aspects of concurrency on a single system, with emphasis on issues of mutual exclusion and deadlock.

Part Three. Memory: Provides a comprehensive survey of techniques for memory management, including virtual memory.

Part Four. Scheduling: Provides a comparative discussion of various approaches to process scheduling. Thread scheduling, SMP scheduling, and real-time scheduling are also examined.

Part Five. Input/Output and Files: Examines the issues involved in OS control of the I/O function. Special attention is devoted to disk I/O, which is the key to system performance. Also provides an overview of file management.

Part Six. Embedded Systems: Embedded systems far outnumber general-purpose computing systems and present a number of unique OS challenges. The chapter includes a discussion of common principles plus coverage of two example systems: TinyOS and eCos.

Part Seven. Security: Provides a survey of threats and mechanisms for providing computer and network security.

Part Eight. Distributed Systems: Examines the major trends in the networking of computer systems, including TCP/IP, client/server computing, and clusters. Also describes some of the key design areas in the development of distributed operating systems.

A number of online chapters and appendices cover additional topics relevant to the book.

0.2 EXAMPLE SYSTEMS

This text is intended to acquaint the reader with the design principles and implementation issues of contemporary operating systems. Accordingly, a purely conceptual or theoretical treatment would be inadequate. To illustrate the concepts and to tie

ISBN 1-256-52023-3

them to real-world design choices that must be made, two operating systems have been chosen as running examples:

- **Windows:** A multitasking operating system designed to run on a variety of PCs, workstations, and servers. It is one of the few recent commercial operating systems that have essentially been designed from scratch. As such, it is in a position to incorporate in a clean fashion the latest developments in operating system technology. The current version, presented in this book, is Windows 7.
- **UNIX:** A multitasking operating system originally intended for minicomputers but implemented on a wide range of machines from powerful microcomputers to supercomputers. Included under this topic is Linux.

The discussion of the example systems is distributed throughout the text rather than assembled as a single chapter or appendix. Thus, during the discussion of concurrency, the concurrency mechanisms of each example system are described, and the motivation for the individual design choices is discussed. With this approach, the design concepts discussed in a given chapter are immediately reinforced with real-world examples.

The book also makes use of other example systems where appropriate, particularly in the chapter on embedded systems.

0.3 A ROADMAP FOR READERS AND INSTRUCTORS

It would be natural for the reader to question the particular ordering of topics presented in this book. For example, the topic of scheduling (Chapters 9 and 10) is closely related to those of concurrency (Chapters 5 and 6) and the general topic of processes (Chapter 3) and might reasonably be covered immediately after those topics.

The difficulty is that the various topics are highly interrelated. For example, in discussing virtual memory, it is useful to refer to the scheduling issues related to a page fault. Of course, it is also useful to refer to some memory management issues when discussing scheduling decisions. This type of example can be repeated endlessly: A discussion of scheduling requires some understanding of I/O management and vice versa.

Figure 0.1 suggests some of the important interrelationships between topics. The solid lines indicate very strong relationships, from the point of view of design and implementation decisions. Based on this diagram, it makes sense to begin with a basic discussion of processes, which we do in Chapter 3. After that, the order is somewhat arbitrary. Many treatments of operating systems bunch all of the material on processes at the beginning and then deal with other topics. This is certainly valid. However, the central significance of memory management, which I believe is of equal importance to process management, has led to a decision to present this material prior to an in-depth look at scheduling.

The ideal solution is for the student, after completing Chapters 1 through 3 in series, to read and absorb the following chapters in parallel: 4 followed by (optional) 5; 6 followed by 7; 8 followed by (optional) 9; 10. The remaining parts can

ISBN 1-256-52023-3

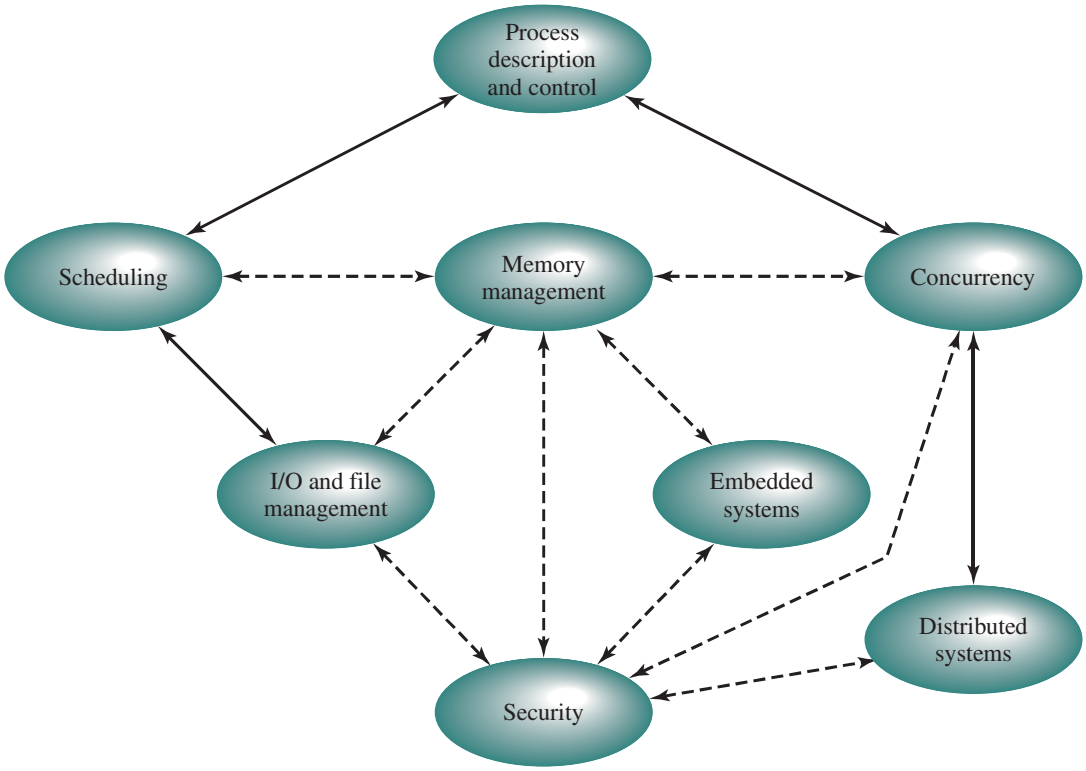


Figure 0.1 **OS Topics**

be done in any order. However, although the human brain may engage in parallel processing, the human student finds it impossible (and expensive) to work successfully with four copies of the same book simultaneously open to four different chapters. Given the necessity for a linear ordering, I think that the ordering used in this book is the most effective.

A final word. Chapter 2, especially Section 2.3, provides a top-level view of all of the key concepts covered in later chapters. Thus, after reading Chapter 2, there is considerable flexibility in choosing the order in which to read the remaining chapters.

0.4 INTERNET AND WEB RESOURCES

There are a number of resources available on the Internet and the Web to support this book and for keeping up with developments in this field.

Web Sites for This Book

Three Web sites provide additional resources for students and instructors. A special Web page for this book is maintained at **WilliamStallings.com/OS/OS7e.html**. For students, this Web site includes a list of relevant links, organized by chapter, an errata sheet for the book, and links to the animations used throughout the book. For access to the animations, click on the rotating globe. There are also documents that introduce the C programming language for students who are not familiar with

ISBN 1-256-52023-3

or need a refresher on this language For instructors, this Web site links to course pages by professors teaching from this book and provides a number of other useful documents and links.

There is also an access-controlled Web site, referred to as **Premium Content**, that provides a wealth of supporting material, including additional online chapters, additional online appendices, a set of homework problems with solutions, copies of a number of key papers in this field, and a number of other supporting documents. See the card at the front of this book for access information. Of particular note are the following online documents:

- **Pseudocode:** For those readers not comfortable with C, all of the algorithms are also reproduced in a Pascal-like pseudocode. This pseudocode language is intuitive and particularly easy to follow.
- **Windows 7, UNIX, and Linux descriptions:** As was mentioned, Windows and various flavors of UNIX are used as running case studies, with the discussion distributed throughout the text rather than assembled as a single chapter or appendix. Some readers would like to have all of this material in one place as a reference. Accordingly, all of the Windows, UNIX, and Linux material from the book is reproduced in three documents at the Web site.

Finally, additional material for instructors is available at the **Instructor Resource Center (IRC)** for this book. See Preface for details and access information.

As soon as any typos or other errors are discovered, an errata list for this book will be available at the Web site. Please report any errors that you spot. Errata sheets for my other books are at **WilliamStallings.com**.

I also maintain the Computer Science Student Resource Site, at **ComputerScienceStudent.com**. The purpose of this site is to provide documents, information, and links for computer science students and professionals. Links and documents are organized into six categories:

- **Math:** Includes a basic math refresher, a queueing analysis primer, a number system primer, and links to numerous math sites.
- **How-to:** Advice and guidance for solving homework problems, writing technical reports, and preparing technical presentations.
- **Research resources:** Links to important collections of papers, technical reports, and bibliographies.
- **Miscellaneous:** A variety of useful documents and links.
- **Computer science careers:** Useful links and documents for those considering a career in computer science.
- **Humor and other diversions:** You have to take your mind off your work once in a while.

Other Web Sites

There are numerous Web sites that provide information related to the topics of this book. In subsequent chapters, pointers to specific Web sites can be found

ISBN 1-256-52023-3

in the *Recommended Reading and Web Sites* section. Because the URL for a particular Web site may change, I have not included URLs in the book. For all of the Web sites listed in the book, the appropriate link can be found at this book's Web site. Other links not mentioned in this book will be added to the Web site over time.

USENET Newsgroups

A number of USENET newsgroups are devoted to some aspect of operating systems or to a particular operating system. As with virtually all USENET groups, there is a high noise-to-signal ratio, but it is worth experimenting to see if any meet your needs. The most relevant are as follows:

- **comp.os.research:** The best group to follow. This is a moderated newsgroup that deals with research topics.
- **comp.os.misc:** A general discussion of OS topics.
- **comp.os.linux.development.system:** Linux discussion for developers.

ISBN 1-256-52023-3

PART 1 Background

CHAPTER 1

COMPUTER SYSTEM OVERVIEW

ISBN 1-256-52023-3

No artifact designed by man is so convenient for this kind of functional description as a digital computer. Almost the only ones of its properties that are detectable in its behavior are the organizational properties. Almost no interesting statement that one can make about on operating computer bears any particular relation to the specific nature of the hardware. A computer is an organization of elementary functional components in which, to a high approximation, only the function performed by those components is relevant to the behavior of the whole system.

THE SCIENCES OF THE ARTIFICIAL, HERBERT SIMON

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Describe the basic elements of a computer system and their interrelationship.
- Explain the steps taken by a processor to execute an instruction.
- Understand the concept of interrupts and how and why a processor uses interrupts.
- List and describe the levels of a typical computer memory hierarchy.
- Explain the basic characteristics of multiprocessor and multicore organizations.
- Discuss the concept of locality and analyze the performance of a multilevel memory hierarchy.
- Understand the operation of a stack and its use to support procedure call and return.

An operating system (OS) exploits the hardware resources of one or more processors to provide a set of services to system users. The OS also manages secondary memory and I/O (input/output) devices on behalf of its users. Accordingly, it is important to have some understanding of the underlying computer system hardware before we begin our examination of operating systems.

This chapter provides an overview of computer system hardware. In most areas, the survey is brief, as it is assumed that the reader is familiar with this subject. However, several areas are covered in some detail because of their importance to topics covered later in the book. Further topics are covered in Appendix C.

1.1 BASIC ELEMENTS

At a top level, a computer consists of processor, memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the main function of the computer, which is to execute programs. Thus, there are four main structural elements:

- **Processor:** Controls the operation of the computer and performs its data processing functions. When there is only one processor, it is often referred to as the **central processing unit** (CPU).

ISBN 1-256-52023-3

- **Main memory:** Stores data and programs. This memory is typically volatile; that is, when the computer is shut down, the contents of the memory are lost. In contrast, the contents of disk memory are retained even when the computer system is shut down. Main memory is also referred to as *real memory* or *primary memory*.
- **I/O modules:** Move data between the computer and its external environment. The external environment consists of a variety of devices, including secondary memory devices (e.g., disks), communications equipment, and terminals.
- **System bus:** Provides for communication among processors, main memory, and I/O modules.

Figure 1.1 depicts these top-level components. One of the processor's functions is to exchange data with memory. For this purpose, it typically makes use of two internal (to the processor) registers: a memory address register (MAR), which specifies the address in memory for the next read or write; and a memory buffer register (MBR), which contains the data to be written into memory or which receives

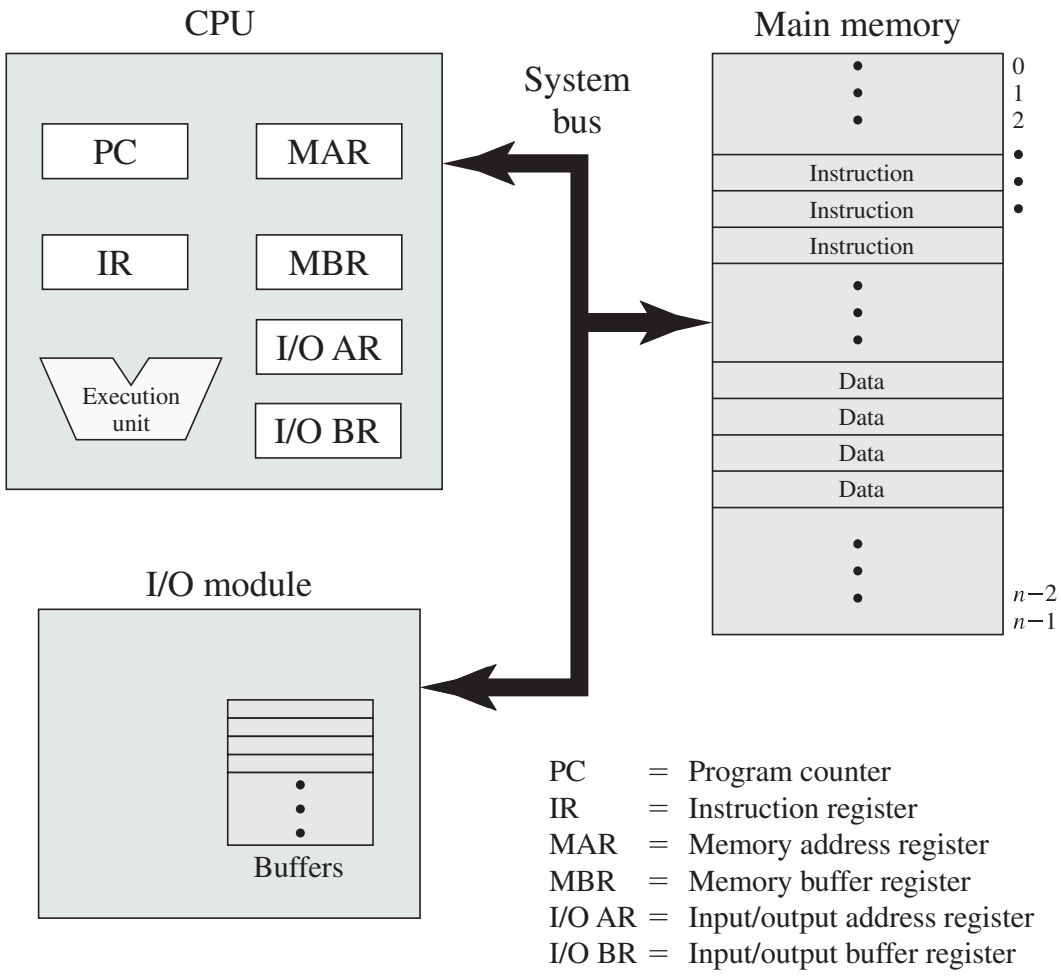


Figure 1.1 Computer Components: Top-Level View

ISBN 1-256-52023-3

the data read from memory. Similarly, an I/O address register (I/OAR) specifies a particular I/O device. An I/O buffer register (I/OBR) is used for the exchange of data between an I/O module and the processor.

A memory module consists of a set of locations, defined by sequentially numbered addresses. Each location contains a bit pattern that can be interpreted as either an instruction or data. An I/O module transfers data from external devices to processor and memory, and vice versa. It contains internal buffers for temporarily holding data until they can be sent on.

1.2 EVOLUTION OF THE MICROPROCESSOR

The hardware revolution that brought about desktop and handheld computing was the invention of the microprocessor, which contained a processor on a single chip. Though originally much slower than multichip processors, microprocessors have continually evolved to the point that they are now much faster for most computations due to the physics involved in moving information around in sub-nanosecond timeframes.

Not only have microprocessors become the fastest general purpose processors available, they are now multiprocessors; each chip (called a socket) contains multiple processors (called cores), each with multiple levels of large memory caches, and multiple logical processors sharing the execution units of each core. As of 2010, it is not unusual for even a laptop to have 2 or 4 cores, each with 2 hardware threads, for a total of 4 or 8 logical processors.

Although processors provide very good performance for most forms of computing, there is increasing demand for numerical computation. Graphical Processing Units (GPUs) provide efficient computation on arrays of data using Single-Instruction Multiple Data (SIMD) techniques pioneered in supercomputers. GPUs are no longer used just for rendering advanced graphics, but they are also used for general numerical processing, such as physics simulations for games or computations on large spreadsheets. Simultaneously, the CPUs themselves are gaining the capability of operating on arrays of data—with increasingly powerful vector units integrated into the processor architecture of the x86 and AMD64 families.

Processors and GPUs are not the end of the computational story for the modern PC. Digital Signal Processors (DSPs) are also present, for dealing with streaming signals—such as audio or video. DSPs used to be embedded in I/O devices, like modems, but they are now becoming first-class computational devices, especially in handhelds. Other specialized computational devices (fixed function units) co-exist with the CPU to support other standard computations, such as encoding/decoding speech and video (codecs), or providing support for encryption and security.

To satisfy the requirements of handheld devices, the classic microprocessor is giving way to the System on a Chip (SoC), where not just the CPUs and caches are on the same chip, but also many of the other components of the system, such as DSPs, GPUs, I/O devices (such as radios and codecs), and main memory.

ISBN 1-256-52023-3

1.3 INSTRUCTION EXECUTION

A program to be executed by a processor consists of a set of instructions stored in memory. In its simplest form, instruction processing consists of two steps: The processor reads (*fetches*) instructions from memory one at a time and executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution. Instruction execution may involve several operations and depends on the nature of the instruction.

The processing required for a single instruction is called an *instruction cycle.* Using a simplified two-step description, the instruction cycle is depicted in Figure 1.2. The two steps are referred to as the *fetch stage* and the *execute stage.* Program execution halts only if the processor is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the processor is encountered.

At the beginning of each instruction cycle, the processor fetches an instruction from memory. Typically, the program counter (PC) holds the address of the next instruction to be fetched. Unless instructed otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address). For example, consider a simplified computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to location 300. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained subsequently.

The fetched instruction is loaded into the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action. In general, these actions fall into four categories:

- **Processor-memory:** Data may be transferred from processor to memory or from memory to processor.
- **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- **Data processing:** The processor may perform some arithmetic or logic operation on data.
- **Control:** An instruction may specify that the sequence of execution be altered. For example, the processor may fetch an instruction from location 149, which

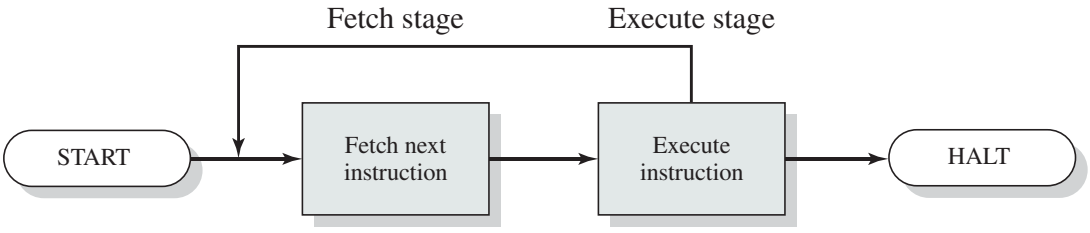
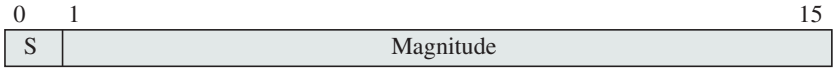


Figure 1.2 Basic Instruction Cycle

ISBN 1-256-52023-3

(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

Figure 1.3 Characteristics of a Hypothetical Machine

specifies that the next instruction will be from location 182. The processor sets the program counter to 182. Thus, on the next fetch stage, the instruction will be fetched from location 182 rather than 150.

An instruction's execution may involve a combination of these actions.

Consider a simple example using a hypothetical processor that includes the characteristics listed in Figure 1.3. The processor contains a single data register, called the accumulator (AC). Both instructions and data are 16 bits long, and memory is organized as a sequence of 16-bit words. The instruction format provides 4 bits for the opcode, allowing as many as $2^4 = 16$ different opcodes (represented by a single hexadecimal[1] digit). The opcode defines the operation the processor is to perform. With the remaining 12 bits of the instruction format, up to $2^{12} = 4{,}096$ (4K) words of memory (denoted by three hexadecimal digits) can be directly addressed.

Figure 1.4 illustrates a partial program execution, showing the relevant portions of memory and processor registers. The program fragment shown adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location. Three instructions, which can be described as three fetch and three execute stages, are required:

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the IR and the PC is incremented. Note that this process involves the use of a memory address register (MAR)

[1]A basic refresher on number systems (decimal, binary, hexadecimal) can be found at the Computer Science Student Resource Site at ComputerScienceStudent.com.

ISBN 1-256-52023-3

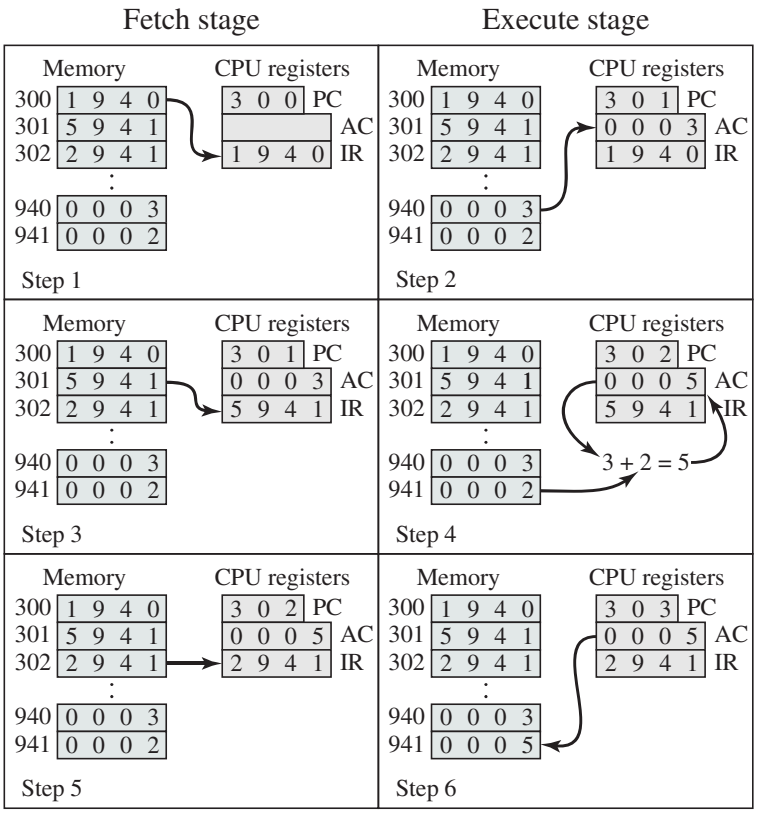


Figure 1.4 **Example of Program Execution** (contents of memory and registers in hexadecimal)

and a memory buffer register (MBR). For simplicity, these intermediate registers are not shown.

2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded from memory. The remaining 12 bits (three hexadecimal digits) specify the address, which is 940.
3. The next instruction (5941) is fetched from location 301 and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added and the result is stored in the AC.
5. The next instruction (2941) is fetched from location 302 and the PC is incremented.
6. The contents of the AC are stored in location 941.

In this example, three instruction cycles, each consisting of a fetch stage and an execute stage, are needed to add the contents of location 940 to the contents of 941. With a more complex set of instructions, fewer instruction cycles would be needed. Most modern processors include instructions that contain more than one address. Thus the execution stage for a particular instruction may involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation.

ISBN 1-256-52023-3

1.4 INTERRUPTS

Virtually all computers provide a mechanism by which other modules (I/O, memory) may interrupt the normal sequencing of the processor. Table 1.1 lists the most common classes of interrupts.

Interrupts are provided primarily as a way to improve processor utilization. For example, most I/O devices are much slower than the processor. Suppose that the processor is transferring data to a printer using the instruction cycle scheme of Figure 1.2. After each write operation, the processor must pause and remain idle until the printer catches up. The length of this pause may be on the order of many thousands or even millions of instruction cycles. Clearly, this is a very wasteful use of the processor.

To give a specific example, consider a PC that operates at 1 GHz, which would allow roughly 10^9 instructions per second.[2] A typical hard disk has a rotational speed of 7200 revolutions per minute for a half-track rotation time of 4 ms, which is 4 million times slower than the processor.

Figure 1.5a illustrates this state of affairs. The user program performs a series of WRITE calls interleaved with processing. The solid vertical lines represent segments of code in a program. Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O. The WRITE calls are to an I/O routine that is a system utility and that will perform the actual I/O operation. The I/O program consists of three sections:

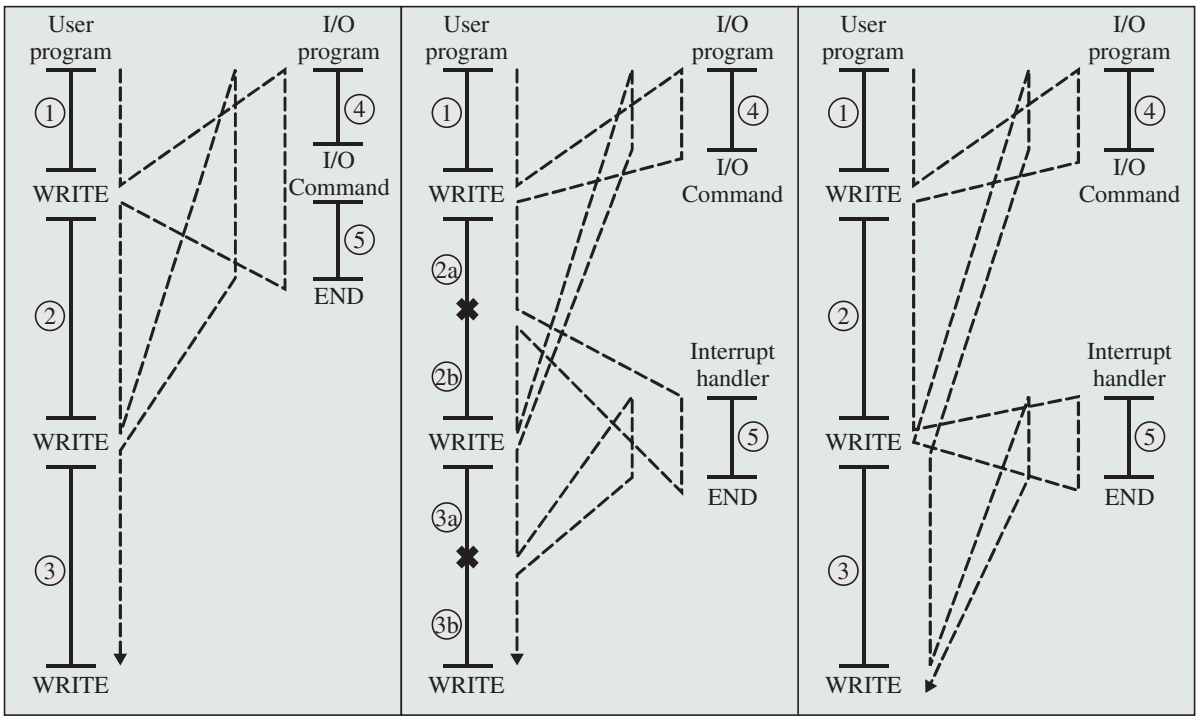
- A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.
- The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically check the status, or poll, the I/O device). The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.
- A sequence of instructions, labeled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.

Table 1.1 Classes of Interrupts

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

[2] A discussion of the uses of numerical prefixes, such as giga and tera, is contained in a supporting document at the Computer Science Student Resource Site at ComputerScienceStudent.com.

ISBN 1-256-52023-3



(a) No interrupts (b) Interrupts; short I/O wait (c) Interrupts; long I/O wait

Figure 1.5 Program Flow of Control without and with Interrupts

The dashed line represents the path of execution followed by the processor; that is, this line shows the sequence in which instructions are executed. Thus, after the first WRITE instruction is encountered, the user program is interrupted and execution continues with the I/O program. After the I/O program execution is complete, execution resumes in the user program immediately following the WRITE instruction.

Because the I/O operation may take a relatively long time to complete, the I/O program is hung up waiting for the operation to complete; hence, the user program is stopped at the point of the WRITE call for some considerable period of time.

Interrupts and the Instruction Cycle

With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in Figure 1.5b. As before, the user program reaches a point at which it makes a system call in the form of a WRITE call. The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user program.

When the external device becomes ready to be serviced, that is, when it is ready to accept more data from the processor, the I/O module for that external device sends an *interrupt request* signal to the processor. The processor responds by suspending operation of the current program; branching off to a routine to service

ISBN 1-256-52023-3

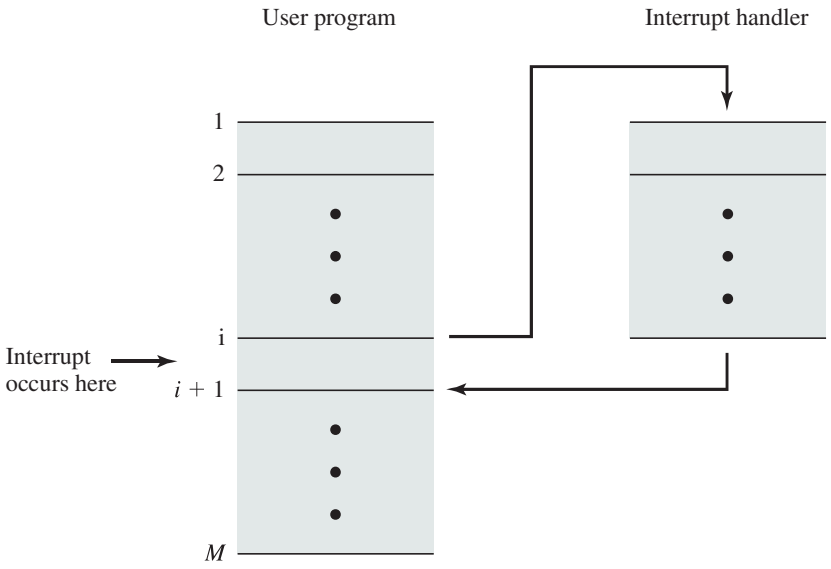


Figure 1.6 **Transfer of Control via Interrupts**

that particular I/O device, known as an interrupt handler; and resuming the original execution after the device is serviced. The points at which such interrupts occur are indicated by ✖ in Figure 1.5b. Note that an interrupt can occur at any point in the main program, not just at one specific instruction.

For the user program, an interrupt suspends the normal sequence of execution. When the interrupt processing is completed, execution resumes (Figure 1.6). Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the OS are responsible for suspending the user program and then resuming it at the same point.

To accommodate interrupts, an *interrupt stage* is added to the instruction cycle, as shown in Figure 1.7 (compare Figure 1.2). In the interrupt stage, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch stage and fetches the next instruction of the current program. If an interrupt is pending,

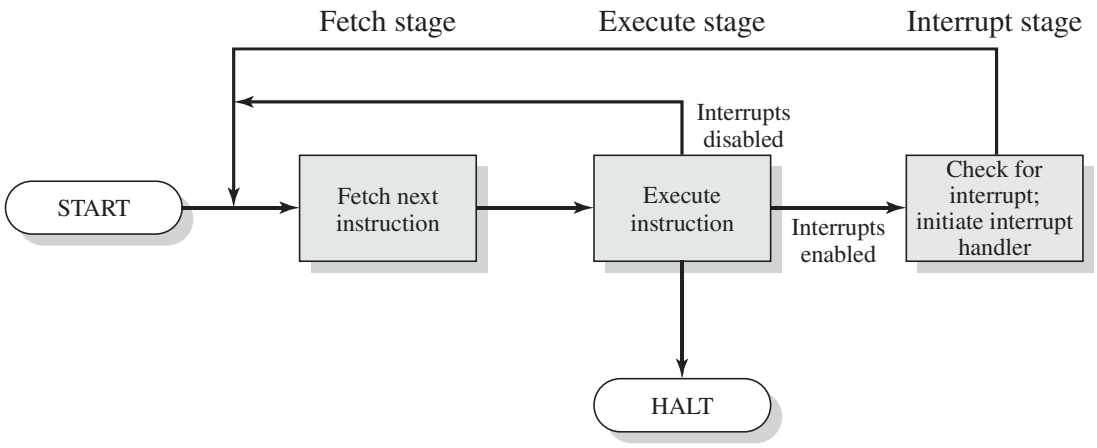


Figure 1.7 **Instruction Cycle with Interrupts**

ISBN 1-256-52023-3

the processor suspends execution of the current program and executes an *interrupt-handler* routine. The interrupt-handler routine is generally part of the OS. Typically, this routine determines the nature of the interrupt and performs whatever actions are needed. In the example we have been using, the handler determines which I/O module generated the interrupt and may branch to a program that will write more data out to that I/O module. When the interrupt-handler routine is completed, the processor can resume execution of the user program at the point of interruption.

It is clear that there is some overhead involved in this process. Extra instructions must be executed (in the interrupt handler) to determine the nature of the interrupt and to decide on the appropriate action. Nevertheless, because of the relatively large amount of time that would be wasted by simply waiting on an I/O operation, the processor can be employed much more efficiently with the use of interrupts.

To appreciate the gain in efficiency, consider Figure 1.8, which is a timing diagram based on the flow of control in Figures 1.5a and 1.5b. Figures 1.5b and 1.8

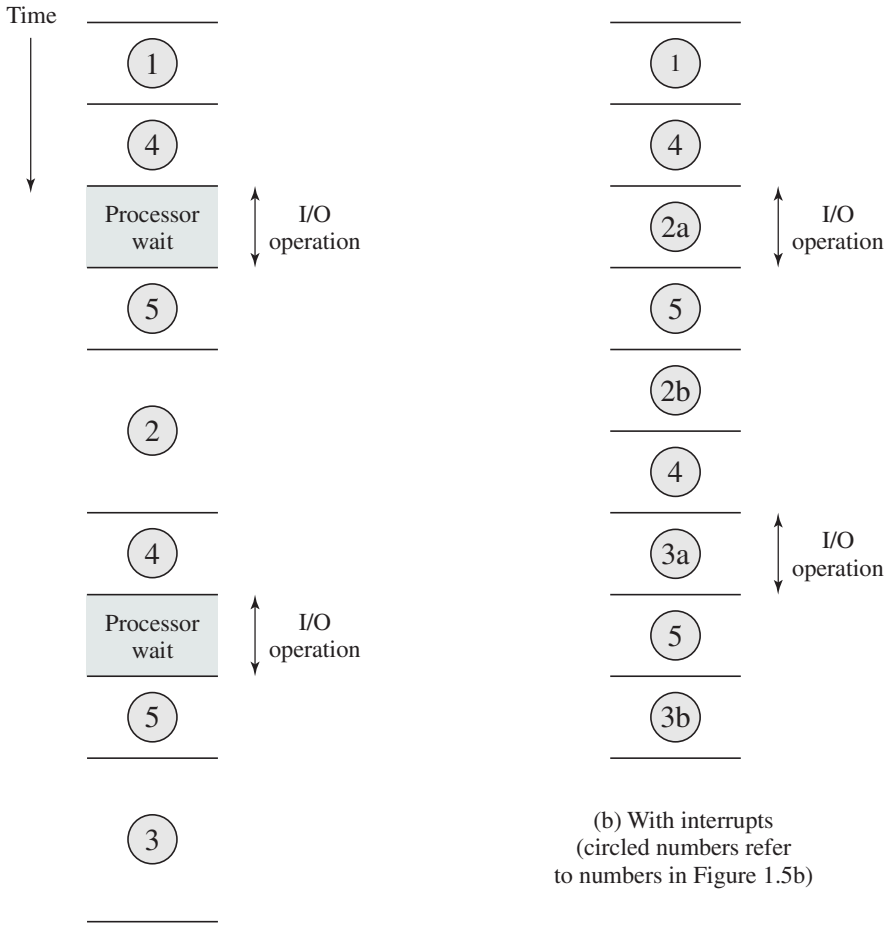
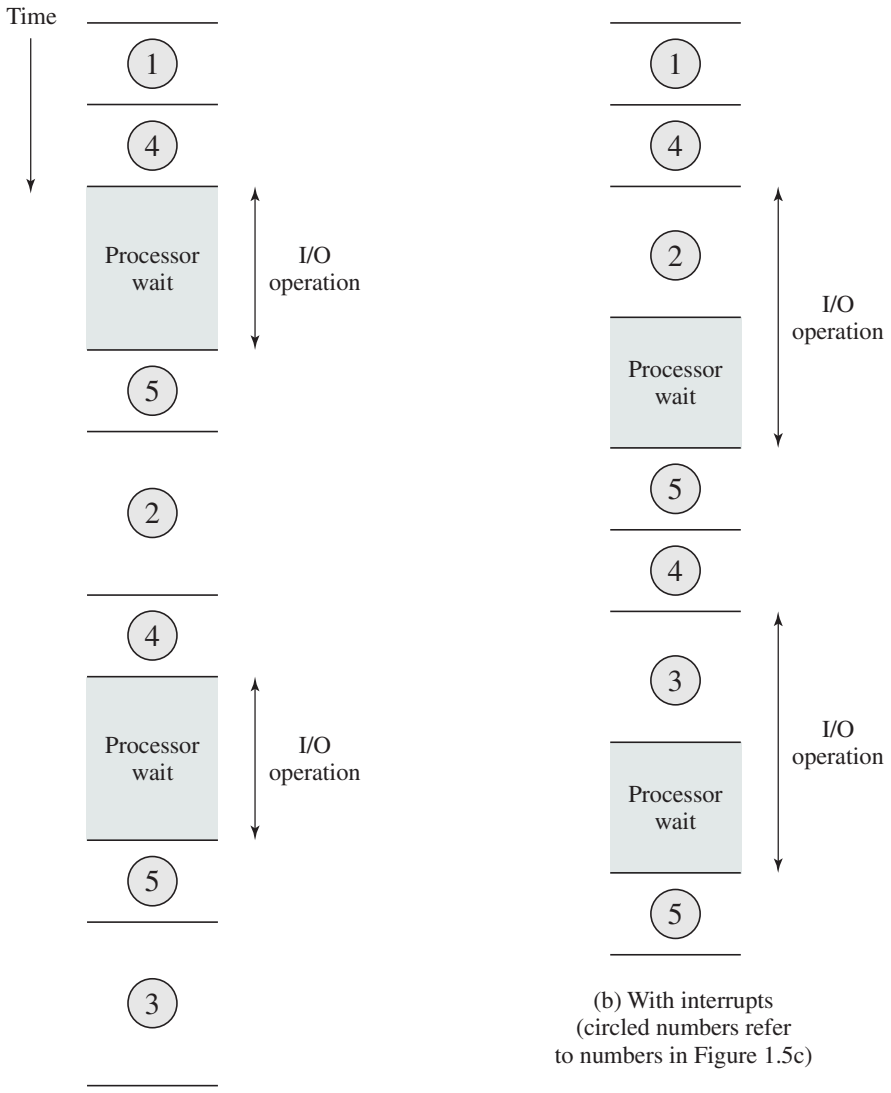


Figure 1.8 Program Timing: Short I/O Wait

ISBN 1-256-52023-3

assume that the time required for the I/O operation is relatively short: less than the time to complete the execution of instructions between write operations in the user program. The more typical case, especially for a slow device such as a printer, is that the I/O operation will take much more time than executing a sequence of user instructions. Figure 1.5c indicates this state of affairs. In this case, the user program reaches the second WRITE call before the I/O operation spawned by the first call is complete. The result is that the user program is hung up at that point. When the preceding I/O operation is completed, this new WRITE call may be processed, and a new I/O operation may be started. Figure 1.9 shows the timing for this situation with and without the use of interrupts. We can see that there is still a gain in efficiency because part of the time during which the I/O operation is underway overlaps with the execution of user instructions.



(a) Without interrupts (circled numbers refer to numbers in Figure 1.5a)

(b) With interrupts (circled numbers refer to numbers in Figure 1.5c)

Figure 1.9 Program Timing: Long I/O Wait

ISBN 1-256-52023-3

Interrupt Processing

An interrupt triggers a number of events, both in the processor hardware and in software. Figure 1.10 shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt, as indicated in Figure 1.7.
3. The processor tests for a pending interrupt request, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.
4. The processor next needs to prepare to transfer control to the interrupt routine. To begin, it saves information needed to resume the current program at the point of interrupt. The minimum information required is the program status word[3] (PSW) and the location of the next instruction to be executed, which

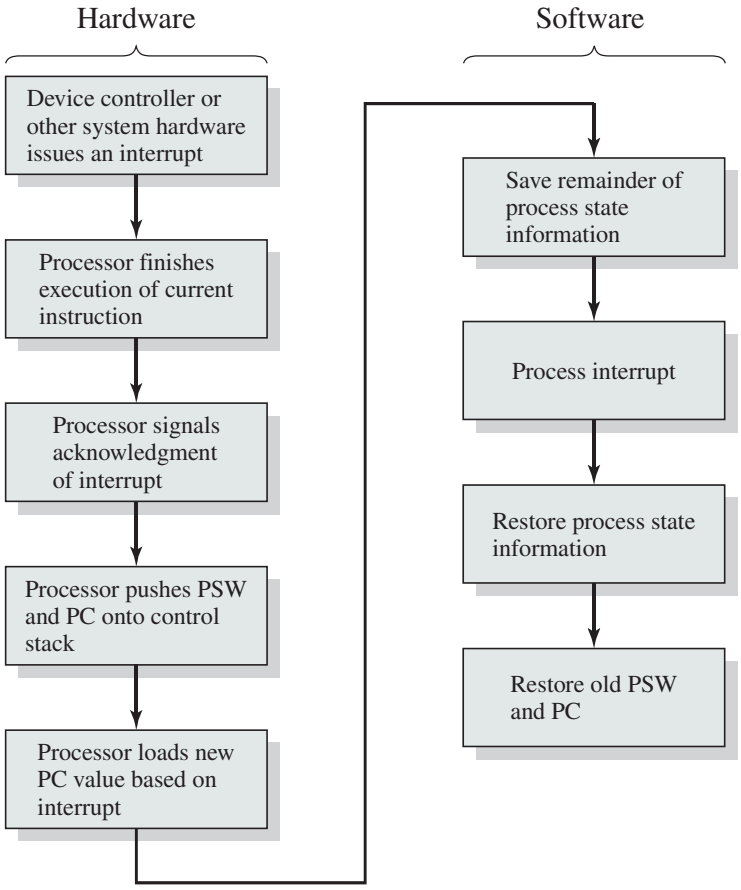


Figure 1.10 Simple Interrupt Processing

[3]The PSW contains status information about the currently running process, including memory usage information, condition codes, and other status information, such as an interrupt enable/disable bit and a kernel/user mode bit. See Appendix C for further discussion.

ISBN 1-256-52023-3

is contained in the program counter (PC). These can be pushed onto a control stack (see Appendix P).

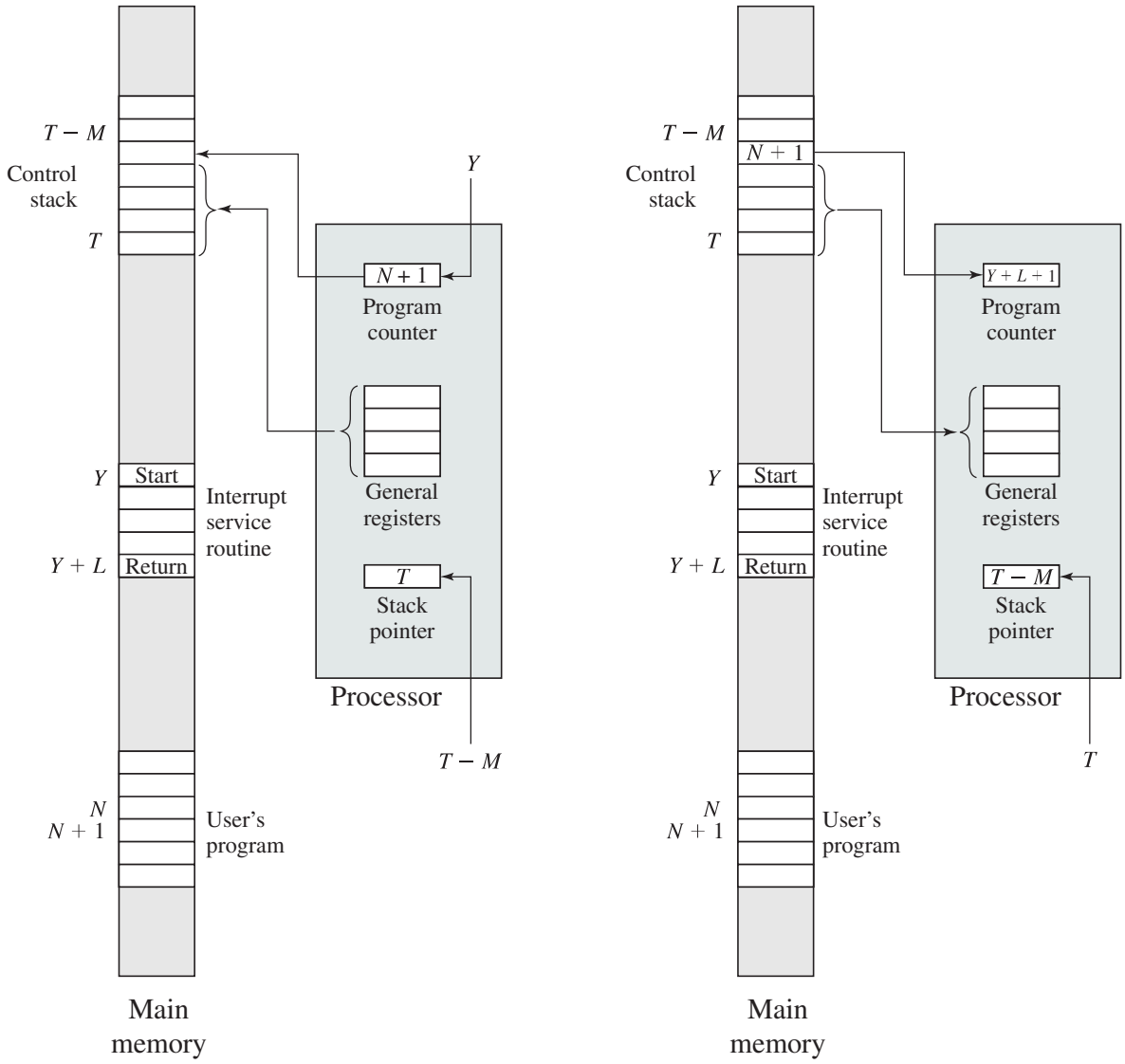
5. The processor then loads the program counter with the entry location of the interrupt-handling routine that will respond to this interrupt. Depending on the computer architecture and OS design, there may be a single program, one for each type of interrupt, or one for each device and each type of interrupt. If there is more than one interrupt-handling routine, the processor must determine which one to invoke. This information may have been included in the original interrupt signal, or the processor may have to issue a request to the device that issued the interrupt to get a response that contains the needed information.

Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an instruction fetch. Because the instruction fetch is determined by the contents of the program counter, control is transferred to the interrupt-handler program. The execution of this program results in the following operations:

6. At this point, the program counter and PSW relating to the interrupted program have been saved on the control stack. However, there is other information that is considered part of the state of the executing program. In particular, the contents of the processor registers need to be saved, because these registers may be used by the interrupt handler. So all of these values, plus any other state information, need to be saved. Typically, the interrupt handler will begin by saving the contents of all registers on the stack. Other state information that must be saved is discussed in Chapter 3. Figure 1.11a shows a simple example. In this case, a user program is interrupted after the instruction at location *N*. The contents of all of the registers plus the address of the next instruction ($N + 1$), a total of *M* words, are pushed onto the control stack. The stack pointer is updated to point to the new top of stack, and the program counter is updated to point to the beginning of the interrupt service routine.
7. The interrupt handler may now proceed to process the interrupt. This includes an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers (e.g., see Figure 1.11b).
9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

It is important to save all of the state information about the interrupted program for later resumption. This is because the interrupt is not a routine called from the program. Rather, the interrupt can occur at any time and therefore at any point in the execution of a user program. Its occurrence is unpredictable.

ISBN 1-256-52023-3



(a) Interrupt occurs after instruction at location N

(b) Return from interrupt

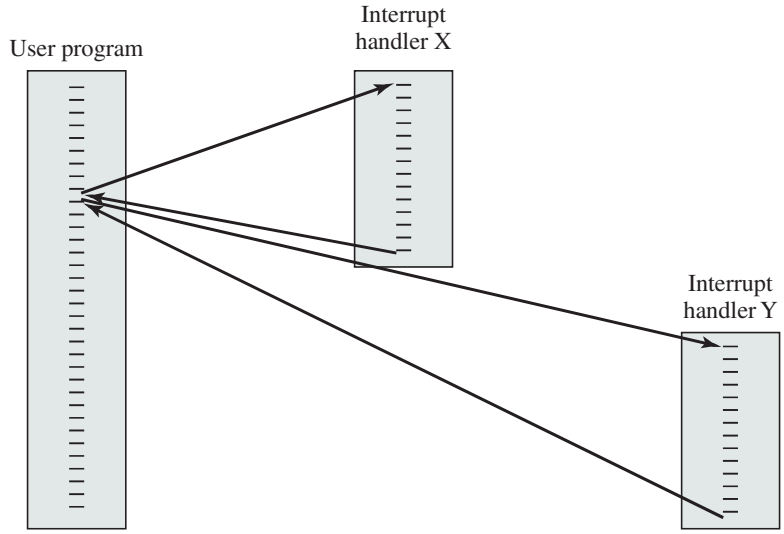
Figure 1.11 Changes in Memory and Registers for an Interrupt

Multiple Interrupts

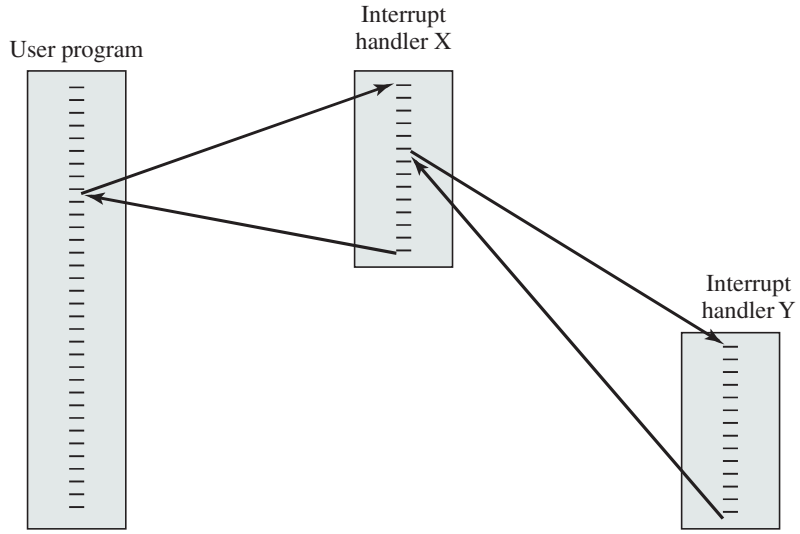
So far, we have discussed the occurrence of a single interrupt. Suppose, however, that one or more interrupts can occur while an interrupt is being processed. For example, a program may be receiving data from a communications line and printing results at the same time. The printer will generate an interrupt every time that it completes a print operation. The communication line controller will generate an interrupt every time a unit of data arrives. The unit could either be a single character or a block, depending on the nature of the communications discipline. In any case, it is possible for a communications interrupt to occur while a printer interrupt is being processed.

ISBN 1-256-52023-3

Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed. A *disabled interrupt* simply means that the processor ignores any new interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has reenabled interrupts. Thus, if an interrupt occurs when a user program is executing, then interrupts are disabled immediately. After the interrupt-handler routine completes, interrupts are reenabled before resuming the user program, and the processor checks to see if additional interrupts have occurred. This approach is simple, as interrupts are handled in strict sequential order (Figure 1.12a).



(a) Sequential interrupt processing



(b) Nested interrupt processing

Figure 1.12 Transfer of Control with Multiple Interrupts

ISBN 1-256-52023-3

The drawback to the preceding approach is that it does not take into account relative priority or time-critical needs. For example, when input arrives from the communications line, it may need to be absorbed rapidly to make room for more input. If the first batch of input has not been processed before the second batch arrives, data may be lost because the buffer on the I/O device may fill and overflow.

A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be interrupted (Figure 1.12b). As an example of this second approach, consider a system with three I/O devices: a printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively. Figure 1.13, based on an example in [TANE06], illustrates a possible sequence. A user program begins at $t = 0$. At $t = 10$, a printer interrupt occurs; user information is placed on the control stack and execution continues at the printer interrupt service routine (ISR). While this routine is still executing, at $t = 15$ a communications interrupt occurs. Because the communications line has higher priority than the printer, the interrupt request is honored. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues at the communications ISR. While this routine is executing, a disk interrupt occurs ($t = 20$). Because this interrupt is of lower priority, it is simply held, and the communications ISR runs to completion.

When the communications ISR is complete ($t = 25$), the previous processor state is restored, which is the execution of the printer ISR. However, before even a single instruction in that routine can be executed, the processor honors the higher-priority disk interrupt and transfers control to the disk ISR. Only when that routine is complete ($t = 35$) is the printer ISR resumed. When that routine completes ($t = 40$), control finally returns to the user program.

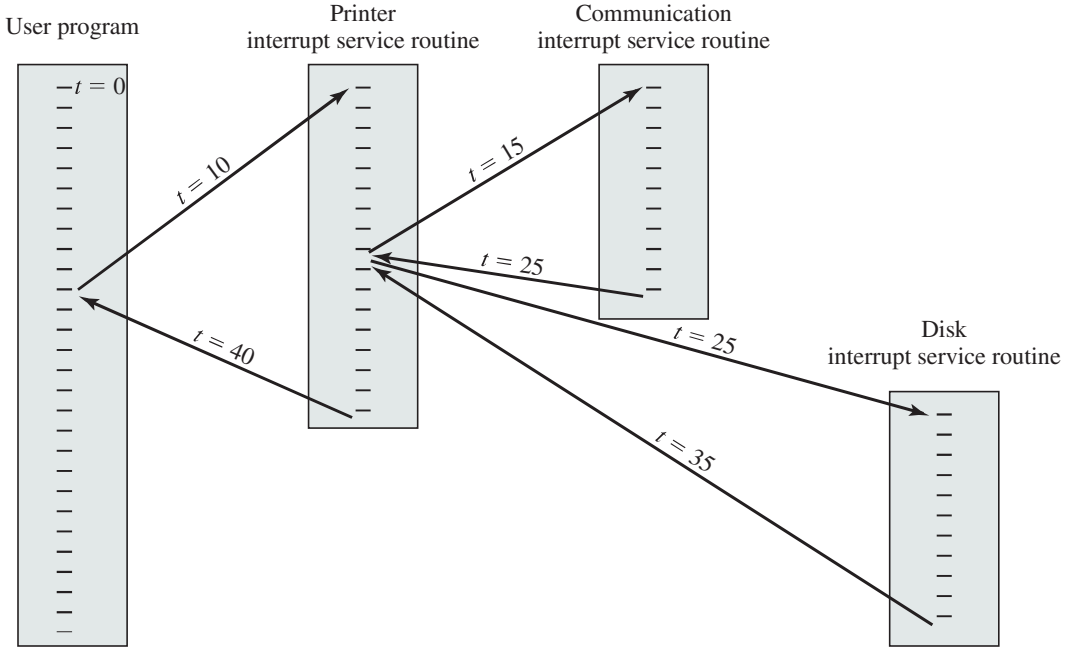


Figure 1.13 **Example Time Sequence of Multiple Interrupts**

ISBN 1-256-52023-3

1.5 THE MEMORY HIERARCHY

The design constraints on a computer's memory can be summed up by three questions: How much? How fast? How expensive?

The question of how much is somewhat open ended. If the capacity is there, applications will likely be developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reasonable in relationship to other components.

As might be expected, there is a trade-off among the three key characteristics of memory: namely, capacity, access time, and cost. A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

- Faster access time, greater cost per bit
- Greater capacity, smaller cost per bit
- Greater capacity, slower access speed

The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with fast access times.

The way out of this dilemma is to not rely on a single memory component or technology, but to employ a **memory hierarchy**. A typical hierarchy is illustrated in Figure 1.14. As one goes down the hierarchy, the following occur:

a. Decreasing cost per bit
b. Increasing capacity
c. Increasing access time
d. Decreasing frequency of access to the memory by the processor

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization is the decreasing frequency of access at lower levels. We will examine this concept in greater detail later in this chapter, when we discuss the cache, and when we discuss virtual memory later in this book. A brief explanation is provided at this point.

Suppose that the processor has access to two levels of memory. Level 1 contains 1,000 bytes and has an access time of 0.1 μs; level 2 contains 100,000 bytes and has an access time of 1 μs. Assume that if a byte to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the byte is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the byte is in level 1 or level 2.

ISBN 1-256-52023-3

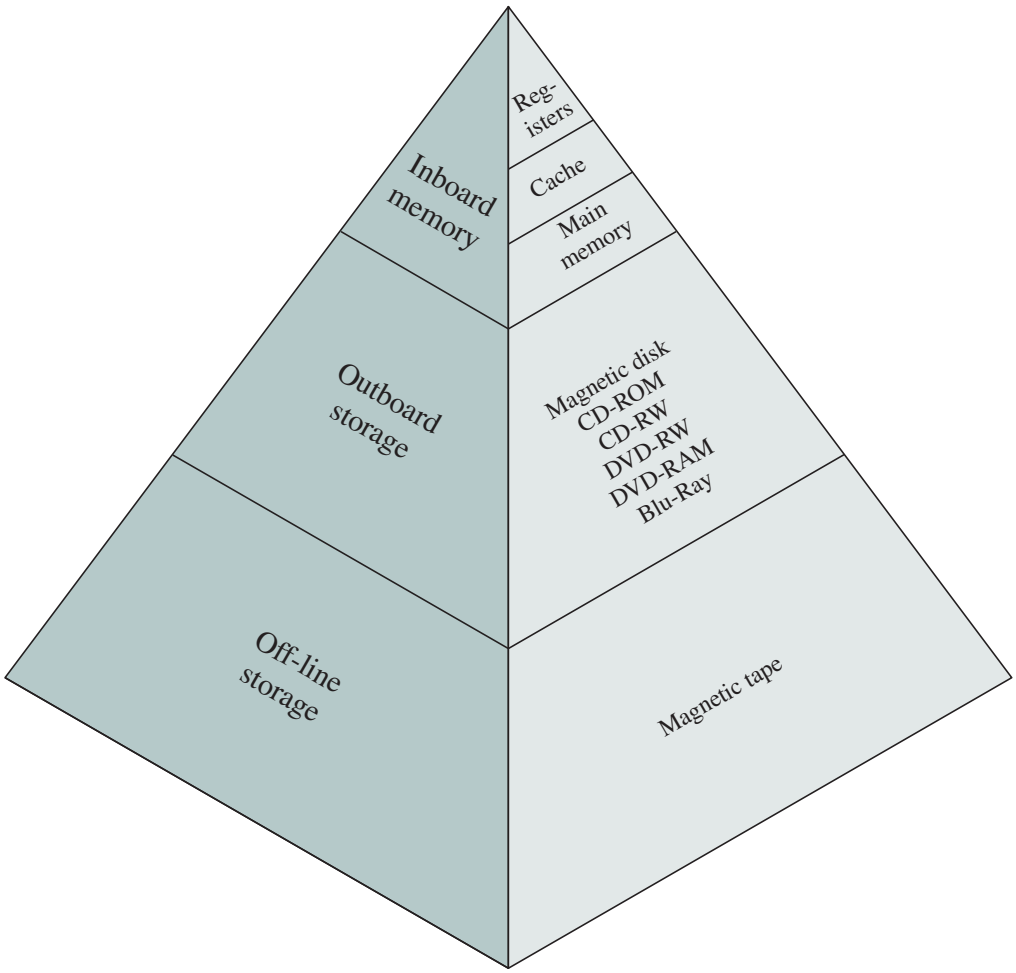


Figure 1.14 **The Memory Hierarchy**

Figure 1.15 shows the general shape of the curve that models this situation. The figure shows the average access time to a two-level memory as a function of the **hit ratio** H, where H is defined as the fraction of all memory accesses that are found in the faster memory (e.g., the cache), T_1 is the access time to level 1, and T_2 is the access time to level 2.[4] As can be seen, for high percentages of level 1 access, the average total access time is much closer to that of level 1 than that of level 2.

In our example, suppose 95% of the memory accesses are found in the cache (H = 0.95). Then the average time to access a byte can be expressed as

$$(0.95)\ (0.1\ \mu s) + (0.05)\ (0.1\ \mu s + 1\ \mu s) = 0.095 + 0.055 = 0.15\ \mu s$$

The result is close to the access time of the faster memory. So the strategy of using two memory levels works in principle, but only if conditions (a) through (d) in the preceding list apply. By employing a variety of technologies, a spectrum of

[4]If the accessed word is found in the faster memory, that is defined as a **hit**. A **miss** occurs if the accessed word is not found in the faster memory.

ISBN 1-256-52023-3

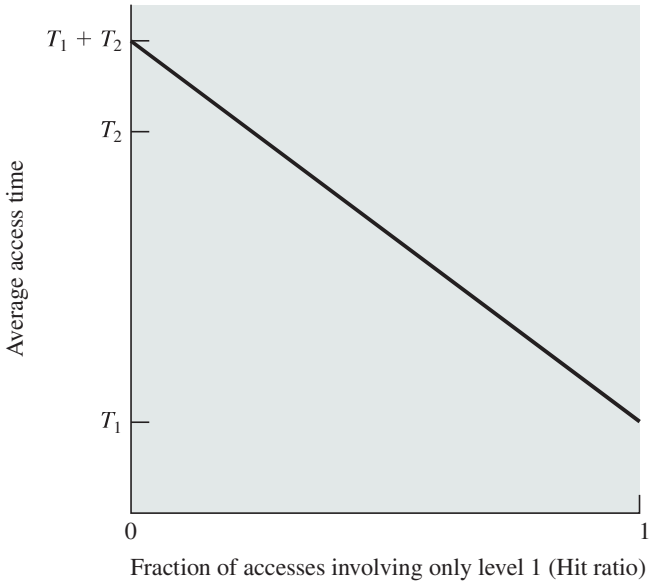


Figure 1.15 **Performance of a Simple Two-Level Memory**

memory systems exists that satisfies conditions (a) through (c). Fortunately, condition (d) is also generally valid.

The basis for the validity of condition (d) is a principle known as **locality of reference** [DENN68]. During the course of execution of a program, memory references by the processor, for both instructions and data, tend to cluster. Programs typically contain a number of iterative loops and subroutines. Once a loop or subroutine is entered, there are repeated references to a small set of instructions. Similarly, operations on tables and arrays involve access to a clustered set of data bytes. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

Accordingly, it is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is substantially less than that of the level above. Consider the two-level example already presented. Let level 2 memory contain all program instructions and data. The current clusters can be temporarily placed in level 1. From time to time, one of the clusters in level 1 will have to be swapped back to level 2 to make room for a new cluster coming in to level 1. On average, however, most references will be to instructions and data contained in level 1.

This principle can be applied across more than two levels of memory. The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor. Typically, a processor will contain a few dozen such registers, although some processors contain hundreds of registers. Skipping down two levels, main memory is the principal internal memory system of the computer. Each location in main memory has a unique address, and most machine instructions refer to one or more main memory addresses. Main memory is usually extended with a higher-speed, smaller cache. The cache is not usually visible to the programmer or, indeed, to the processor. It is a device for staging the movement of data between main memory and processor registers to improve performance.

ISBN 1-256-52023-3

The three forms of memory just described are, typically, volatile and employ semiconductor technology. The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost. Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable disk, tape, and optical storage. External, nonvolatile memory is also referred to as **secondary memory** or **auxiliary memory**. These are used to store program and data files, and are usually visible to the programmer only in terms of files and records, as opposed to individual bytes or words. A hard disk is also used to provide an extension to main memory known as virtual memory, which is discussed in Chapter 8.

Additional levels can be effectively added to the hierarchy in software. For example, a portion of main memory can be used as a buffer to temporarily hold data that are to be read out to disk. Such a technique, sometimes referred to as a disk cache (examined in detail in Chapter 11), improves performance in two ways:

- Disk writes are clustered. Instead of many small transfers of data, we have a few large transfers of data. This improves disk performance and minimizes processor involvement.
- Some data destined for write-out may be referenced by a program before the next dump to disk. In that case, the data are retrieved rapidly from the software cache rather than slowly from the disk.

Appendix 1A examines the performance implications of multilevel memory structures.

1.6 CACHE MEMORY

Although cache memory is invisible to the OS, it interacts with other memory management hardware. Furthermore, many of the principles used in virtual memory schemes (discussed in Chapter 8) are also applied in cache memory.

Motivation

On all instruction cycles, the processor accesses memory at least once, to fetch the instruction, and often one or more additional times, to fetch operands and/or store results. The rate at which the processor can execute instructions is clearly limited by the memory cycle time (the time it takes to read one word from or write one word to memory). This limitation has been a significant problem because of the persistent mismatch between processor and main memory speeds: Over the years, processor speed has consistently increased more rapidly than memory access speed. We are faced with a trade-off among speed, cost, and size. Ideally, main memory should be built with the same technology as that of the processor registers, giving memory cycle times comparable to processor cycle times. This has always been too expensive a strategy. The solution is to exploit the principle of locality by providing a small, fast memory between the processor and main memory, namely the cache.

ISBN 1-256-52023-3

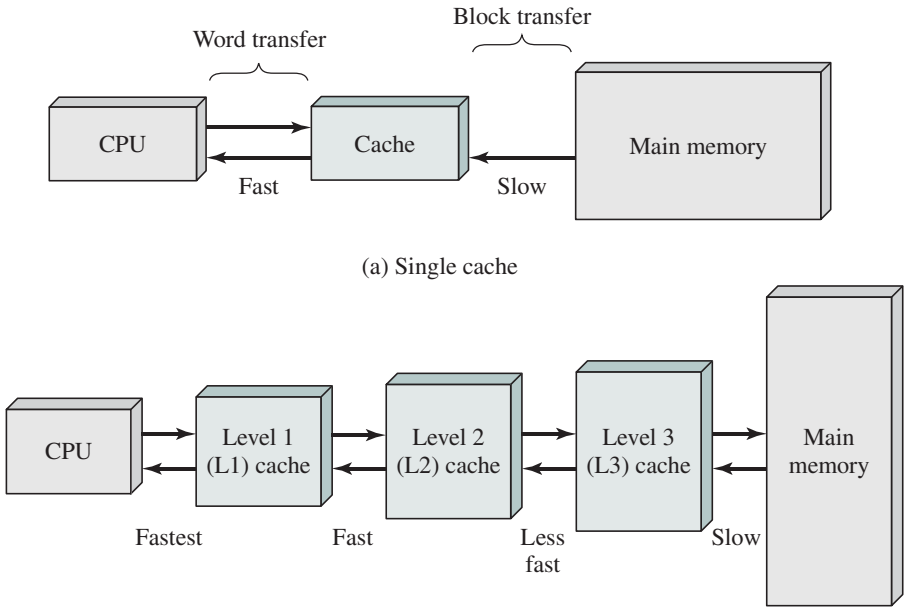


Figure 1.16 Cache and Main Memory

Cache Principles

Cache memory is intended to provide memory access time approaching that of the fastest memories available and at the same time support a large memory size that has the price of less expensive types of semiconductor memories. The concept is illustrated in Figure 1.16a. There is a relatively large and slow main memory together with a smaller, faster cache memory. The cache contains a copy of a portion of main memory. When the processor attempts to read a byte or word of memory, a check is made to determine if the byte or word is in the cache. If so, the byte or word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of bytes, is read into the cache and then the byte or word is delivered to the processor. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that many of the near-future memory references will be to other bytes in the block.

Figure 1.16b depicts the use of multiple levels of cache. The L2 cache is slower and typically larger than the L1 cache, and the L3 cache is slower and typically larger than the L2 cache.

Figure 1.17 depicts the structure of a cache/main memory system. Main memory consists of up to 2^n addressable words, with each word having a unique n-bit address. For mapping purposes, this memory is considered to consist of a number of fixed-length **blocks** of K words each. That is, there are $M = 2^n/K$ blocks. Cache consists of C **slots** (also referred to as *lines*) of K words each, and the number of slots is considerably less than the number of main memory blocks ($C << M$).[5] Some subset of the blocks of main memory resides in the slots of the cache. If a word in a block

[5]The symbol << means *much less than.* Similarly, the symbol >> means *much greater than.*

ISBN 1-256-52023-3

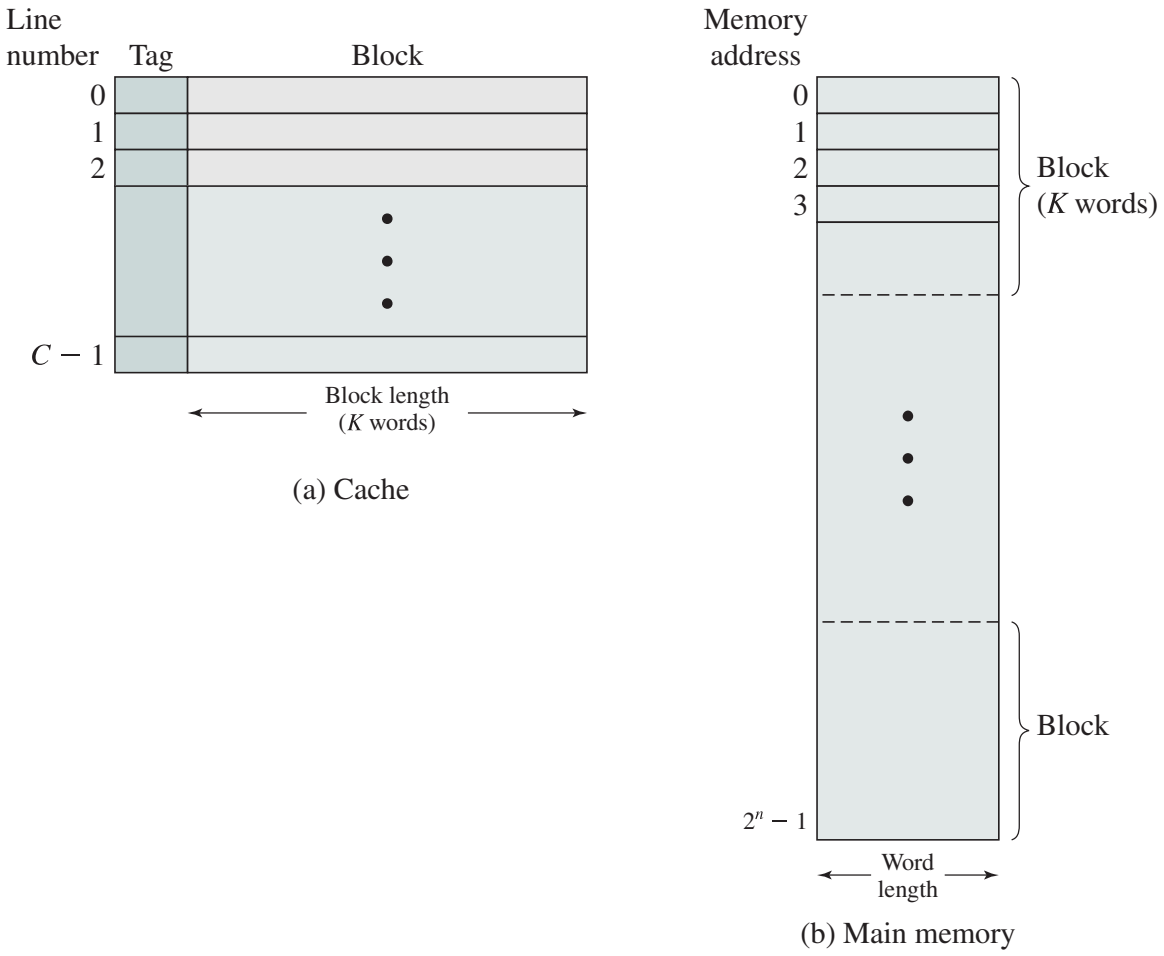


Figure 1.17 Cache/Main-Memory Structure

of memory that is not in the cache is read, that block is transferred to one of the slots of the cache. Because there are more blocks than slots, an individual slot cannot be uniquely and permanently dedicated to a particular block. Therefore, each slot includes a tag that identifies which particular block is currently being stored. The tag is usually some number of higher-order bits of the address and refers to all addresses that begin with that sequence of bits.

As a simple example, suppose that we have a 6-bit address and a 2-bit tag. The tag 01 refers to the block of locations with the following addresses: 010000, 010001, 010010, 010011, 010100, 010101, 010110, 010111, 011000, 011001, 011010, 011011, 011100, 011101, 011110, 011111.

Figure 1.18 illustrates the read operation. The processor generates the address, RA, of a word to be read. If the word is contained in the cache, it is delivered to the processor. Otherwise, the block containing that word is loaded into the cache and the word is delivered to the processor.

Cache Design

A detailed discussion of cache design is beyond the scope of this book. Key elements are briefly summarized here. We will see that similar design issues must be

ISBN 1-256-52023-3

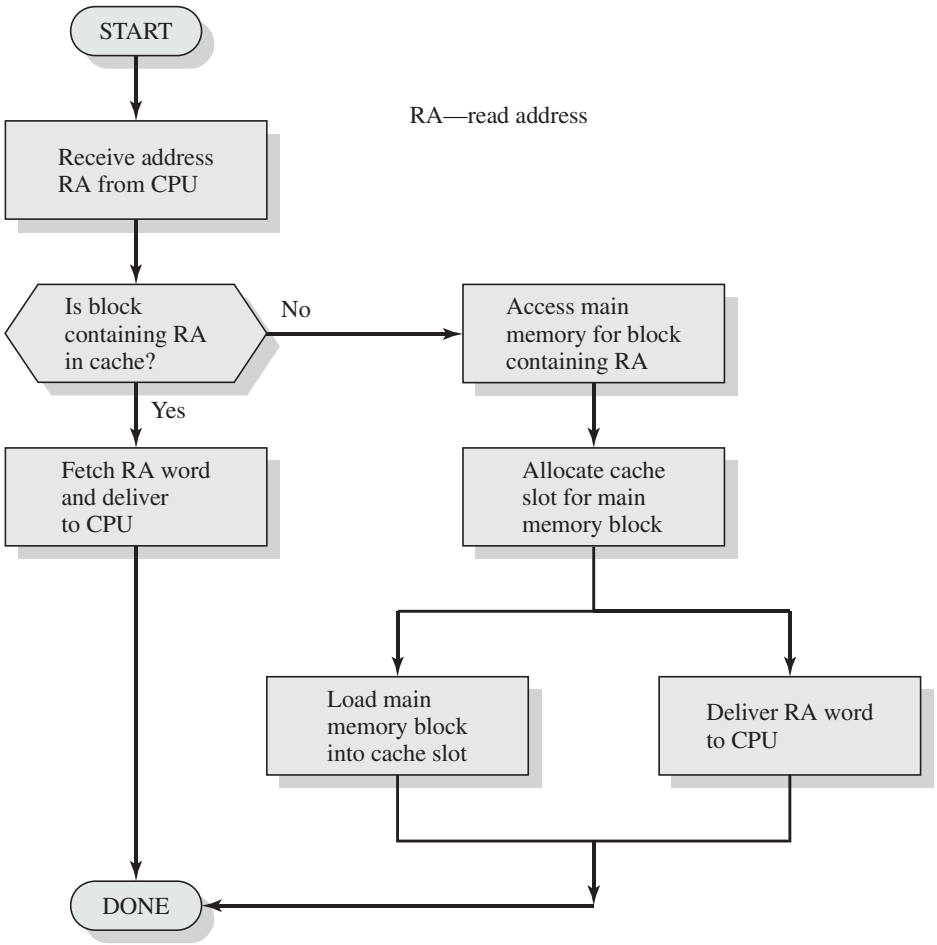


Figure 1.18 Cache Read Operation

addressed in dealing with virtual memory and disk cache design. They fall into the following categories:

- Cache size
- Block size
- Mapping function
- Replacement algorithm
- Write policy
- Number of cache levels

We have already dealt with the issue of **cache size**. It turns out that reasonably small caches can have a significant impact on performance. Another size issue is that of **block size**: the unit of data exchanged between cache and main memory. As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality: the high probability that data in the vicinity of a referenced word are likely to be referenced in the near future. As the

ISBN 1-256-52023-3

block size increases, more useful data are brought into the cache. The hit ratio will begin to decrease, however, as the block becomes even bigger and the probability of using the newly fetched data becomes less than the probability of reusing the data that have to be moved out of the cache to make room for the new block.

When a new block of data is read into the cache, the **mapping function** determines which cache location the block will occupy. Two constraints affect the design of the mapping function. First, when one block is read in, another may have to be replaced. We would like to do this in such a way as to minimize the probability that we will replace a block that will be needed in the near future. The more flexible the mapping function, the more scope we have to design a replacement algorithm to maximize the hit ratio. Second, the more flexible the mapping function, the more complex is the circuitry required to search the cache to determine if a given block is in the cache.

The **replacement algorithm** chooses, within the constraints of the mapping function, which block to replace when a new block is to be loaded into the cache and the cache already has all slots filled with other blocks. We would like to replace the block that is least likely to be needed again in the near future. Although it is impossible to identify such a block, a reasonably effective strategy is to replace the block that has been in the cache longest with no reference to it. This policy is referred to as the least-recently-used (LRU) algorithm. Hardware mechanisms are needed to identify the least-recently-used block.

If the contents of a block in the cache are altered, then it is necessary to write it back to main memory before replacing it. The **write policy** dictates when the memory write operation takes place. At one extreme, the writing can occur every time that the block is updated. At the other extreme, the writing occurs only when the block is replaced. The latter policy minimizes memory write operations but leaves main memory in an obsolete state. This can interfere with multiple-processor operation and with direct memory access by I/O hardware modules.

Finally, it is now commonplace to have multiple levels of cache, labeled L1 (cache closest to the processor), L2, and in many cases a third level L3. A discussion of the performance benefits of multiple cache levels is beyond our scope; see [STAL10] for a discussion.

1.7 DIRECT MEMORY ACCESS

Three techniques are possible for I/O operations: programmed I/O, interrupt-driven I/O, and direct memory access (DMA). Before discussing DMA, we briefly define the other two techniques; see Appendix C for more detail.

When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. In the case of **programmed I/O**, the I/O module performs the requested action and then sets the appropriate bits in the I/O status register but takes no further action to alert the processor. In particular, it does not interrupt the processor. Thus, after the I/O instruction is invoked, the processor must take some active role in determining when the I/O instruction is completed. For this purpose,

ISBN 1-256-52023-3

the processor periodically checks the status of the I/O module until it finds that the operation is complete.

With programmed I/O, the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of more data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the performance level of the entire system is severely degraded.

An alternative, known as **interrupt-driven I/O**, is for the processor to issue an I/O command to a module and then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and then resumes its former processing.

Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor. Thus, both of these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.
2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

When large volumes of data are to be moved, a more efficient technique is required: **direct memory access (DMA).** The DMA function can be performed by a separate module on the system bus or it can be incorporated into an I/O module. In either case, the technique works as follows. When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

- Whether a read or write is requested
- The address of the I/O device involved
- The starting location in memory to read data from or write data to
- The number of words to be read or written

The processor then continues with other work. It has delegated this I/O operation to the DMA module, and that module will take care of it. The DMA module transfers the entire block of data, one word at a time, directly to or from memory without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, the processor is involved only at the beginning and end of the transfer.

The DMA module needs to take control of the bus to transfer data to and from memory. Because of this competition for bus usage, there may be times when the processor needs the bus and must wait for the DMA module. Note that this is not an interrupt; the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle (the time it takes to transfer one word across the bus). The overall effect is to cause the processor to execute more slowly during a DMA transfer when processor access to the bus is required. Nevertheless, for a multiple-word I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

ISBN 1-256-52023-3

1.8 MULTIPROCESSOR AND MULTICORE ORGANIZATION

Traditionally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequences of instructions. A processor executes programs by executing machine instructions in sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results).

This view of the computer has never been entirely true. At the micro-operation level, multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time. Both of these are examples of performing functions in parallel.

As computer technology has evolved and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to improve performance and, in some cases, to improve reliability. In this book, we examine the three most popular approaches to providing parallelism by replicating processors: symmetric multiprocessors (SMPs), multicore computers, and clusters. SMPs and multicore computers are discussed in this section; clusters are examined in Chapter 16.

Symmetric Multiprocessors

DEFINITION An SMP can be defined as a stand-alone computer system with the following characteristics:

1. There are two or more similar processors of comparable capability.
2. These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor.
3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
4. All processors can perform the same functions (hence the term *symmetric*).
5. The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.

Points 1 to 4 should be self-explanatory. Point 5 illustrates one of the contrasts with a loosely coupled multiprocessing system, such as a cluster. In the latter, the physical unit of interaction is usually a message or complete file. In an SMP, individual data elements can constitute the level of interaction, and there can be a high degree of cooperation between processes.

An SMP organization has a number of potential advantages over a uniprocessor organization, including the following:

- **Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type.

ISBN 1-256-52023-3

- **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.
- **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.
- **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

It is important to note that these are potential, rather than guaranteed, benefits. The operating system must provide tools and functions to exploit the parallelism in an SMP system.

An attractive feature of an SMP is that the existence of multiple processors is transparent to the user. The operating system takes care of scheduling of tasks on individual processors and of synchronization among processors.

ORGANIZATION Figure 1.19 illustrates the general organization of an SMP. There are multiple processors, each of which contains its own control unit, arithmetic-logic unit, and registers. Each processor has access to a shared main memory and the I/O devices through some form of interconnection mechanism; a shared bus is a common facility. The processors can communicate with each other through memory (messages and status information left in shared address spaces). It may

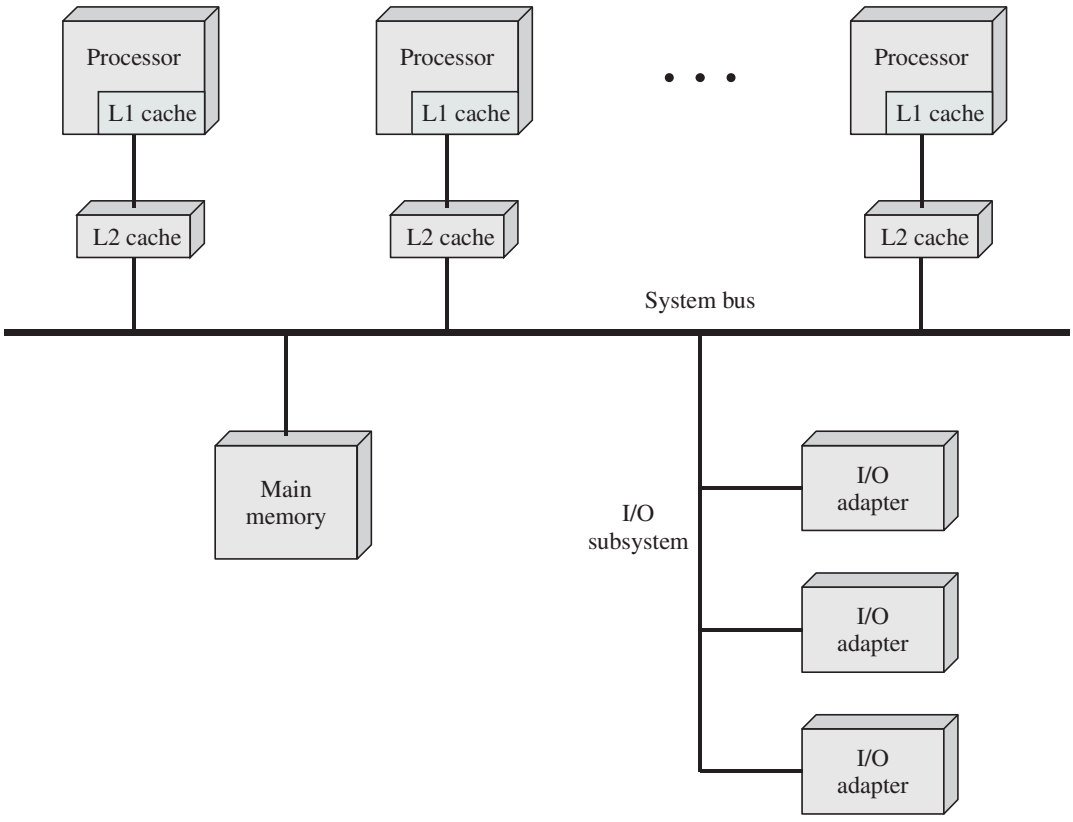


Figure 1.19 Symmetric Multiprocessor Organization

ISBN 1-256-52023-3

also be possible for processors to exchange signals directly. The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible.

In modern computers, processors generally have at least one level of cache memory that is private to the processor. This use of cache introduces some new design considerations. Because each local cache contains an image of a portion of main memory, if a word is altered in one cache, it could conceivably invalidate a word in another cache. To prevent this, the other processors must be alerted that an update has taken place. This problem is known as the cache coherence problem and is typically addressed in hardware rather than by the OS.[6]

Multicore Computers

A **multicore** computer, also known as a **chip multiprocessor**, combines two or more processors (called cores) on a single piece of silicon (called a die). Typically, each core consists of all of the components of an independent processor, such as registers, ALU, pipeline hardware, and control unit, plus L1 instruction and data caches. In addition to the multiple cores, contemporary multicore chips also include L2 cache and, in some cases, L3 cache.

The motivation for the development of multicore computers can be summed up as follows. For decades, microprocessor systems have experienced a steady, usually exponential, increase in performance. This is partly due to hardware trends, such as an increase in clock frequency and the ability to put cache memory closer to the processor because of the increasing miniaturization of microcomputer components. Performance has also been improved by the increased complexity of processor design to exploit parallelism in instruction execution and memory access. In brief, designers have come up against practical limits in the ability to achieve greater performance by means of more complex processors. Designers have found that the best way to improve performance to take advantage of advances in hardware is to put multiple processors and a substantial amount of cache memory on a single chip. A detailed discussion of the rationale for this trend is beyond our scope, but is summarized in Appendix C.

An example of a multicore system is the Intel Core i7, which includes four x86 processors, each with a dedicated L2 cache, and with a shared L3 cache (Figure 1.20). One mechanism Intel uses to make its caches more effective is prefetching, in which the hardware examines memory access patterns and attempts to fill the caches speculatively with data that's likely to be requested soon.

The Core i7 chip supports two forms of external communications to other chips. The **DDR3 memory controller** brings the memory controller for the DDR (double data rate) main memory onto the chip. The interface supports three channels that are 8 bytes wide for a total bus width of 192 bits, for an aggregate data rate of up to 32 GB/s. With the memory controller on the chip, the Front Side Bus is eliminated. The **QuickPath Interconnect (QPI)** is a point-to-point link electrical interconnect specification. It enables high-speed communications among connected processor chips. The QPI link operates at 6.4 GT/s (transfers per second).

[6]A description of hardware-based cache coherency schemes is provided in [STAL10].

ISBN 1-256-52023-3

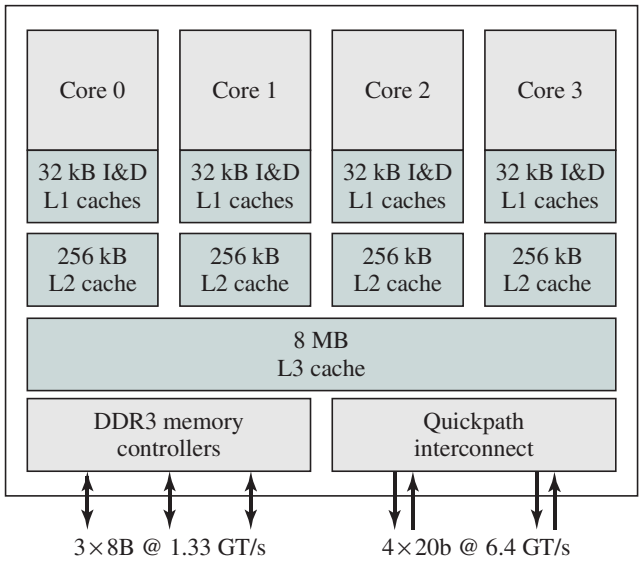


Figure 1.20 **Intel Core i7 Block Diagram**

At 16 bits per transfer, that adds up to 12.8 GB/s; and since QPI links involve dedicated bidirectional pairs, the total bandwidth is 25.6 GB/s.

1.9 RECOMMENDED READING AND WEB SITES

[STAL10] covers the topics of this chapter in detail. In addition, there are many other texts on computer organization and architecture. Among the more worthwhile texts are the following. [PATT09] is a comprehensive survey; [HENN07], by the same authors, is a more advanced text that emphasizes quantitative aspects of design.

[DENN05] looks at the history of the development and application of the locality principle, making for fascinating reading.

DENN05 Denning, P. "The Locality Principle." *Communications of the ACM*, July 2005.

HENN07 Hennessy, J., and Patterson, D. *Computer Architecture: A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann, 2007.

PATT09 Patterson, D., and Hennessy, J. *Computer Organization and Design: The Hardware/Software Interface.* San Mateo, CA: Morgan Kaufmann, 2009.

STAL10 Stallings, W. *Computer Organization and Architecture, 8th ed.* Upper Saddle River, NJ: Prentice Hall, 2010.

Recommended Web sites:

- **WWW Computer Architecture Home Page:** A comprehensive index to information relevant to computer architecture researchers, including architecture groups and projects, technical organizations, literature, employment, and commercial information
- **CPU Info Center:** Information on specific processors, including technical papers, product information, and latest announcements

ISBN 1-256-52023-3

1.10 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

address register	instruction register	program counter
cache memory	interrupt	programmed I/O
cache slot	interrupt-driven I/O	reentrant procedure
central processing unit	I/O module	register
data register	locality	secondary memory
direct memory access	main memory	spatial locality
hit ratio	multicore	stack
input/output	multiprocessor	system bus
instruction	processor	temporal locality
instruction cycle		

Review Questions

1.1. List and briefly define the four main elements of a computer.
1.2. Define the two main categories of processor registers.
1.3. In general terms, what are the four distinct actions that a machine instruction can specify?
1.4. What is an interrupt?
1.5. How are multiple interrupts dealt with?
1.6. What characteristics distinguish the various elements of a memory hierarchy?
1.7. What is cache memory?
1.8. What is the difference between a multiprocessor and a multicore system?
1.9. What is the distinction between spatial locality and temporal locality?
1.10. In general, what are the strategies for exploiting spatial locality and temporal locality?

Problems

1.1. Suppose the hypothetical processor of Figure 1.3 also has two I/O instructions:

0011 = Load AC from I/O
0111 = Store AC to I/O

In these cases, the 12-bit address identifies a particular external device. Show the program execution (using format of Figure 1.4) for the following program:

1. Load AC from device 5.
2. Add contents of memory location 940.
3. Store AC to device 6.

Assume that the next value retrieved from device 5 is 3 and that location 940 contains a value of 2.

1.2. The program execution of Figure 1.4 is described in the text using six steps. Expand this description to show the use of the MAR and MBR.
1.3. Consider a hypothetical 32-bit microprocessor having 32-bit instructions composed of two fields. The first byte contains the opcode and the remainder an immediate operand or an operand address.

ISBN 1-256-52023-3

a. What is the maximum directly addressable memory capacity (in bytes)?
b. Discuss the impact on the system speed if the microprocessor bus has
  1. a 32-bit local address bus and a 16-bit local data bus, or
  2. a 16-bit local address bus and a 16-bit local data bus.
c. How many bits are needed for the program counter and the instruction register?

1.4. Consider a hypothetical microprocessor generating a 16-bit address (e.g., assume that the program counter and the address registers are 16 bits wide) and having a 16-bit data bus.
a. What is the maximum memory address space that the processor can access directly if it is connected to a "16-bit memory"?
b. What is the maximum memory address space that the processor can access directly if it is connected to an "8-bit memory"?
c. What architectural features will allow this microprocessor to access a separate "I/O space"?
d. If an input and an output instruction can specify an 8-bit I/O port number, how many 8-bit I/O ports can the microprocessor support? How many 16-bit I/O ports? Explain.

1.5. Consider a 32-bit microprocessor, with a 16-bit external data bus, driven by an 8-MHz input clock. Assume that this microprocessor has a bus cycle whose minimum duration equals four input clock cycles. What is the maximum data transfer rate across the bus that this microprocessor can sustain in bytes/s? To increase its performance, would it be better to make its external data bus 32 bits or to double the external clock frequency supplied to the microprocessor? State any other assumptions you make and explain. *Hint:* Determine the number of bytes that can be transferred per bus cycle.

1.6. Consider a computer system that contains an I/O module controlling a simple keyboard/printer Teletype. The following registers are contained in the CPU and connected directly to the system bus:

INPR: Input Register, 8 bits
OUTR: Output Register, 8 bits
FGI: Input Flag, 1 bit
FGO: Output Flag, 1 bit
IEN: Interrupt Enable, 1 bit

Keystroke input from the Teletype and output to the printer are controlled by the I/O module. The Teletype is able to encode an alphanumeric symbol to an 8-bit word and decode an 8-bit word into an alphanumeric symbol. The Input flag is set when an 8-bit word enters the input register from the Teletype. The Output flag is set when a word is printed.
a. Describe how the CPU, using the first four registers listed in this problem, can achieve I/O with the Teletype.
b. Describe how the function can be performed more efficiently by also employing IEN.

1.7. In virtually all systems that include DMA modules, DMA access to main memory is given higher priority than processor access to main memory. Why?

1.8. A DMA module is transferring characters to main memory from an external device transmitting at 9600 bits per second (bps). The processor can fetch instructions at the rate of 1 million instructions per second. By how much will the processor be slowed down due to the DMA activity?

1.9. A computer consists of a CPU and an I/O device *D* connected to main memory *M* via a shared bus with a data bus width of one word. The CPU can execute a maximum of 106 instructions per second. An average instruction requires five processor cycles, three of which use the memory bus. A memory read or write operation uses one processor cycle. Suppose that the CPU is continuously executing "background" programs that require 95% of its instruction execution rate but not any I/O instructions.

ISBN 1-256-52023-3

Assume that one processor cycle equals one bus cycle. Now suppose that very large blocks of data are to be transferred between *M* and *D*.

a. If programmed I/O is used and each one-word I/O transfer requires the CPU to execute two instructions, estimate the maximum I/O data transfer rate, in words per second, possible through *D*.

b. Estimate the same rate if DMA transfer is used.

1.10. Consider the following code:

```
for (i = 0; i < 20; i++)
    for (j = 0; j < 10; j++)
        a[i] = a[i] * j
```

a. Give one example of the spatial locality in the code.

b. Give one example of the temporal locality in the code.

1.11. Generalize Equations (1.1) and (1.2) in Appendix 1A to *n*-level memory hierarchies.

1.12. Consider a memory system with the following parameters:

$$Tc = 100 \text{ ns} \quad Cc = 0.01 \text{ cents/bit}$$
$$Tm = 1{,}200 \text{ ns} \quad Cm = 0.001 \text{ cents/bit}$$

a. What is the cost of 1 MByte of main memory?

b. What is the cost of 1 MByte of main memory using cache memory technology?

c. If the effective access time is 10% greater than the cache access time, what is the hit ratio *H*?

1.13. A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20 ns are required to access it. If it is in main memory but not in the cache, 60 ns are needed to load it into the cache (this includes the time to originally check the cache), and then the reference is started again. If the word is not in main memory, 12 ms are required to fetch the word from disk, followed by 60 ns to copy it to the cache, and then the reference is started again. The cache hit ratio is 0.9 and the main-memory hit ratio is 0.6. What is the average time in ns required to access a referenced word on this system?

1.14. Suppose a stack is to be used by the processor to manage procedure calls and returns. Can the program counter be eliminated by using the top of the stack as a program counter?

APPENDIX 1A PERFORMANCE CHARACTERISTICS OF TWO-LEVEL MEMORIES

In this chapter, reference is made to a cache that acts as a buffer between main memory and processor, creating a two-level internal memory. This two-level architecture exploits a property known as locality to provide improved performance over a comparable one-level memory.

The main memory cache mechanism is part of the computer architecture, implemented in hardware and typically invisible to the OS. Accordingly, this mechanism is not pursued in this book. However, there are two other instances of a two-level memory approach that also exploit the property of locality and that are, at least partially, implemented in the OS: virtual memory and the disk cache (Table 1.2). These two topics are explored in Chapters 8 and 11, respectively. In this appendix, we look at some of the performance characteristics of two-level memories that are common to all three approaches.

ISBN 1-256-52023-3

Table 1.2 Characteristics of Two-Level Memories

	Main Memory Cache	Virtual Memory (Paging)	Disk Cache
Typical access time ratios	5 : 1	10^6: 1	10^6: 1
Memory management system	Implemented by special hardware	Combination of hardware and system software	System software
Typical block size	4 to 128 bytes	64 to 4096 bytes	64 to 4096 bytes
Access of processor to second level	Direct access	Indirect access	Indirect access

Locality

The basis for the performance advantage of a two-level memory is the principle of locality, referred to in Section 1.5. This principle states that memory references tend to cluster. Over a long period of time, the clusters in use change; but over a short period of time, the processor is primarily working with fixed clusters of memory references.

Intuitively, the principle of locality makes sense. Consider the following line of reasoning:

1. Except for branch and call instructions, which constitute only a small fraction of all program instructions, program execution is sequential. Hence, in most cases, the next instruction to be fetched immediately follows the last instruction fetched.
2. It is rare to have a long uninterrupted sequence of procedure calls followed by the corresponding sequence of returns. Rather, a program remains confined to a rather narrow window of procedure-invocation depth. Thus, over a short period of time references to instructions tend to be localized to a few procedures.
3. Most iterative constructs consist of a relatively small number of instructions repeated many times. For the duration of the iteration, computation is therefore confined to a small contiguous portion of a program.
4. In many programs, much of the computation involves processing data structures, such as arrays or sequences of records. In many cases, successive references to these data structures will be to closely located data items.

This line of reasoning has been confirmed in many studies. With reference to point (1), a variety of studies have analyzed the behavior of high-level language programs. Table 1.3 includes key results, measuring the appearance of various statement types during execution, from the following studies. The earliest study of programming language behavior, performed by Knuth [KNUT71], examined a collection of FORTRAN programs used as student exercises. Tanenbaum [TANE78] published measurements collected from over 300 procedures used in OS programs and written in a language that supports structured programming (SAL). Patterson and Sequin [PATT82] analyzed a set of measurements taken from compilers and programs for typesetting, computer-aided design (CAD), sorting, and file

ISBN 1-256-52023-3

Table 1.3 Relative Dynamic Frequency of High-Level Language Operations

Study Language Workload	[HUCK83] Pascal Scientific	[KNUT71] FORTRAN Student	[PATT82] Pascal System	[PATT82] C System	[TANE78] SAL System
Assign	74	67	45	38	42
Loop	4	3	5	3	4
Call	1	3	15	12	12
IF	20	11	29	43	36
GOTO	2	9	—	3	—
Other	—	7	6	1	6

comparison. The programming languages C and Pascal were studied. Huck [HUCK83] analyzed four programs intended to represent a mix of general-purpose scientific computing, including fast Fourier transform and the integration of systems of differential equations. There is good agreement in the results of this mixture of languages and applications that branching and call instructions represent only a fraction of statements executed during the lifetime of a program. Thus, these studies confirm assertion (1), from the preceding list.

With respect to assertion (2), studies reported in [PATT85] provide confirmation. This is illustrated in Figure 1.21, which shows call-return behavior. Each call is represented by the line moving down and to the right, and each return by the line moving up and to the right. In the figure, a *window* with depth equal to 5 is defined. Only a sequence of calls and returns with a net movement of 6 in either direction causes the window to move. As can be seen, the executing program can remain within a stationary window for long periods of time. A study by the same analysts of C and Pascal programs showed that a window of depth 8 would only need to shift on less than 1% of the calls or returns [TAMI83].

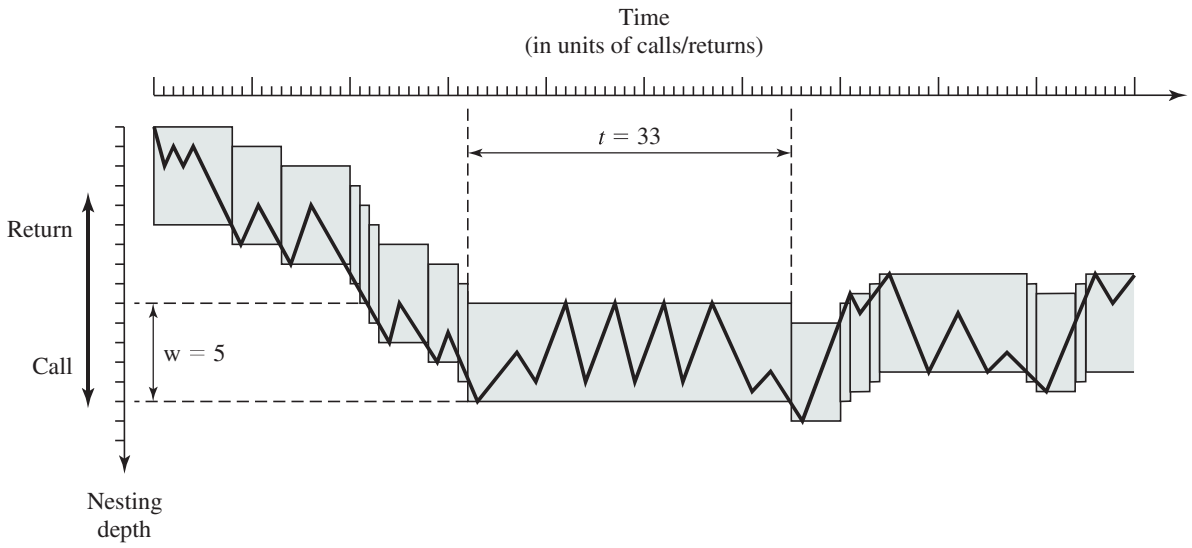


Figure 1.21 Example Call-Return Behavior of a Program

ISBN 1-256-52023-3

A distinction is made in the literature between spatial locality and temporal locality. **Spatial locality** refers to the tendency of execution to involve a number of memory locations that are clustered. This reflects the tendency of a processor to access instructions sequentially. Spatial location also reflects the tendency of a program to access data locations sequentially, such as when processing a table of data. **Temporal locality** refers to the tendency for a processor to access memory locations that have been used recently. For example, when an iteration loop is executed, the processor executes the same set of instructions repeatedly.

Traditionally, temporal locality is exploited by keeping recently used instruction and data values in cache memory and by exploiting a cache hierarchy. Spatial locality is generally exploited by using larger cache blocks and by incorporating prefetching mechanisms (fetching items whose use is expected) into the cache control logic. Recently, there has been considerable research on refining these techniques to achieve greater performance, but the basic strategies remain the same.

Operation of Two-Level Memory

The locality property can be exploited in the formation of a two-level memory. The upper-level memory (M1) is smaller, faster, and more expensive (per bit) than the lower-level memory (M2). M1 is used as a temporary store for part of the contents of the larger M2. When a memory reference is made, an attempt is made to access the item in M1. If this succeeds, then a quick access is made. If not, then a block of memory locations is copied from M2 to M1 and the access then takes place via M1. Because of locality, once a block is brought into M1, there should be a number of accesses to locations in that block, resulting in fast overall service.

To express the average time to access an item, we must consider not only the speeds of the two levels of memory but also the probability that a given reference can be found in M1. We have

$$T_s = H \times T_1 + (1 - H) \times (T_1 + T_2)$$
$$T_1 + (1 - H) \times T_2 \qquad \textbf{(1.1)}$$

where

T_s = average (system) access time
T_1 = access time of M1 (e.g., cache, disk cache)
T_2 = access time of M2 (e.g., main memory, disk)
H = hit ratio (fraction of time reference is found in M1)

Figure 1.15 shows average access time as a function of hit ratio. As can be seen, for a high percentage of hits, the average total access time is much closer to that of M1 than M2.

Performance

Let us look at some of the parameters relevant to an assessment of a two-level memory mechanism. First consider cost. We have

$$C_s = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2} \qquad \textbf{(1.2)}$$

ISBN 1-256-52023-3

where

C_s = average cost per bit for the combined two-level memory
C_1 = average cost per bit of upper-level memory M1
C_2 = average cost per bit of lower-level memory M2
S_1 = size of M1
S_2 = size of M2

We would like $C_s \approx C_2$. Given that $C_1 >> C_2$, this requires $S_1 << S_2$. Figure 1.22 shows the relationship.[7]

Next, consider access time. For a two-level memory to provide a significant performance improvement, we need to have T_s approximately equal to T_1 $T_s \approx T_1$. Given that T_1 is much less than T_2 $T_s >> T_1$, a hit ratio of close to 1 is needed.

So we would like M1 to be small to hold down cost, and large to improve the hit ratio and therefore the performance. Is there a size of M1 that satisfies both requirements to a reasonable extent? We can answer this question with a series of subquestions:

- What value of hit ratio is needed to satisfy the performance requirement?
- What size of M1 will assure the needed hit ratio?
- Does this size satisfy the cost requirement?

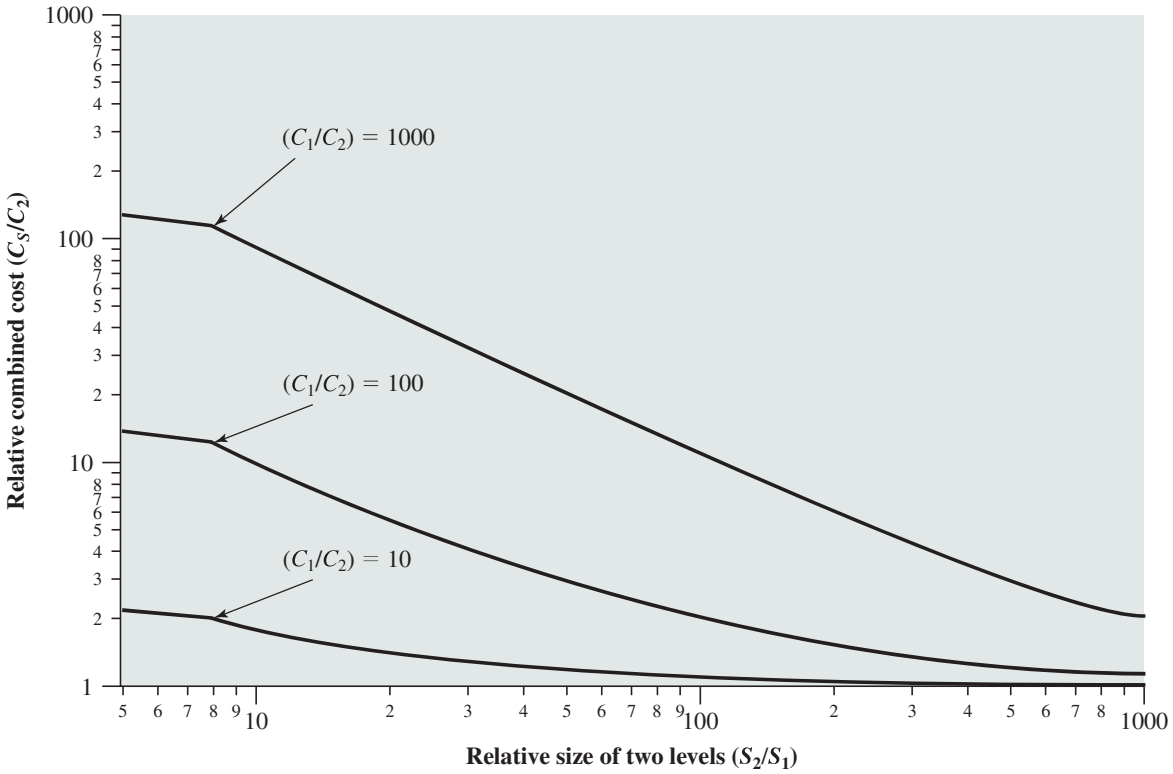


Figure 1.22 Relationship of Average Memory Cost to Relative Memory Size for a Two-Level Memory

[7]Note that both axes use a log scale. A basic review of log scales is in the math refresher document at the Computer Science Student Resource Site at ComputerScienceStudent.com.

ISBN 1-256-52023-3

To get at this, consider the quantity T_1/T_s, which is referred to as the *access efficiency*. It is a measure of how close average access time (T_s) is to M1 access time (T_1). From Equation (1.1),

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H)\frac{T_2}{T_1}} \tag{1.3}$$

In Figure 1.23, we plot T_1/T_s as a function of the hit ratio H, with the quantity T_2/T_1 as a parameter. A hit ratio in the range of 0.8 to 0.9 would seem to be needed to satisfy the performance requirement.

We can now phrase the question about relative memory size more exactly. Is a hit ratio of 0.8 or higher reasonable for $S_1 << S_2$? This will depend on a number of factors, including the nature of the software being executed and the details of the design of the two-level memory. The main determinant is, of course, the degree of locality. Figure 1.24 suggests the effect of locality on the hit ratio. Clearly, if M1 is the same size as M2, then the hit ratio will be 1.0: All of the items in M2 are always stored also in M1. Now suppose that there is no locality; that is, references are completely random. In that case the hit ratio should be a strictly linear function of the relative memory size. For example, if M1 is half the size of M2, then at any time half of the items from M2 are also in M1 and the hit ratio will be 0.5. In practice, however, there is some degree of locality in the references. The effects of moderate and strong locality are indicated in the figure.

So, if there is strong locality, it is possible to achieve high values of hit ratio even with relatively small upper-level memory size. For example, numerous studies

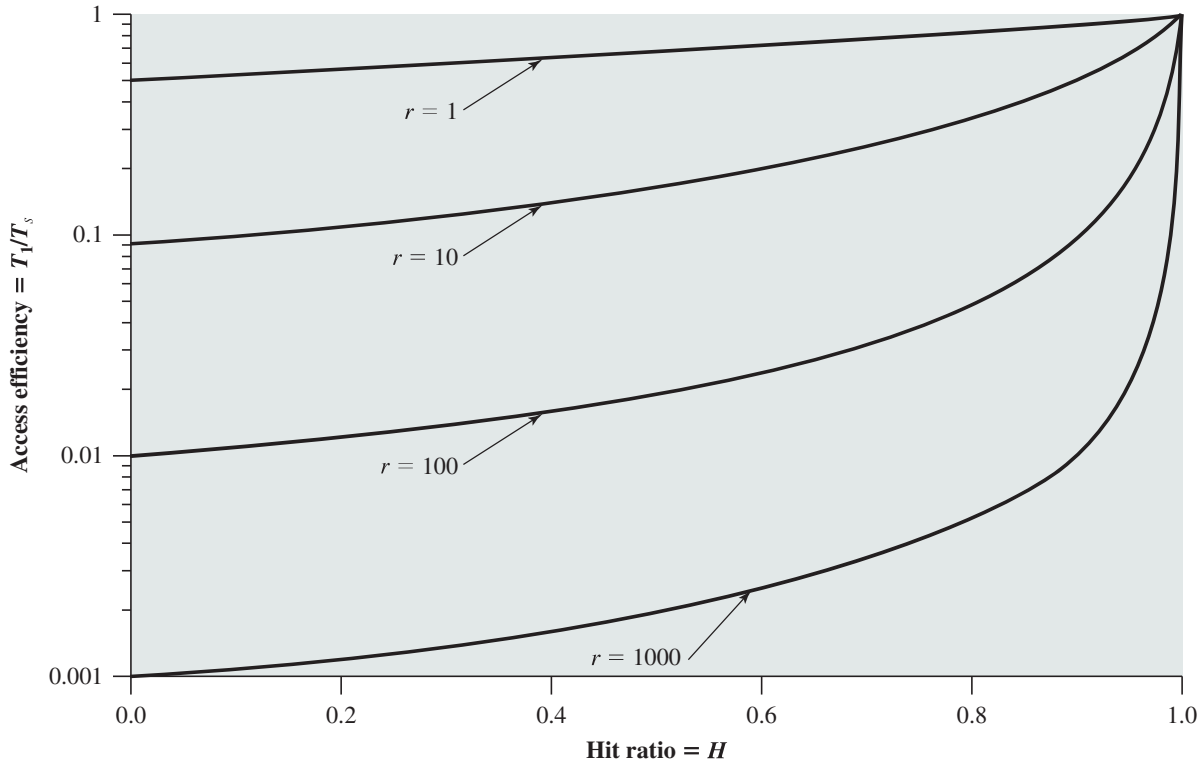


Figure 1.23 **Access Efficiency as a Function of Hit Ratio ($r = T_2/T_1$)**

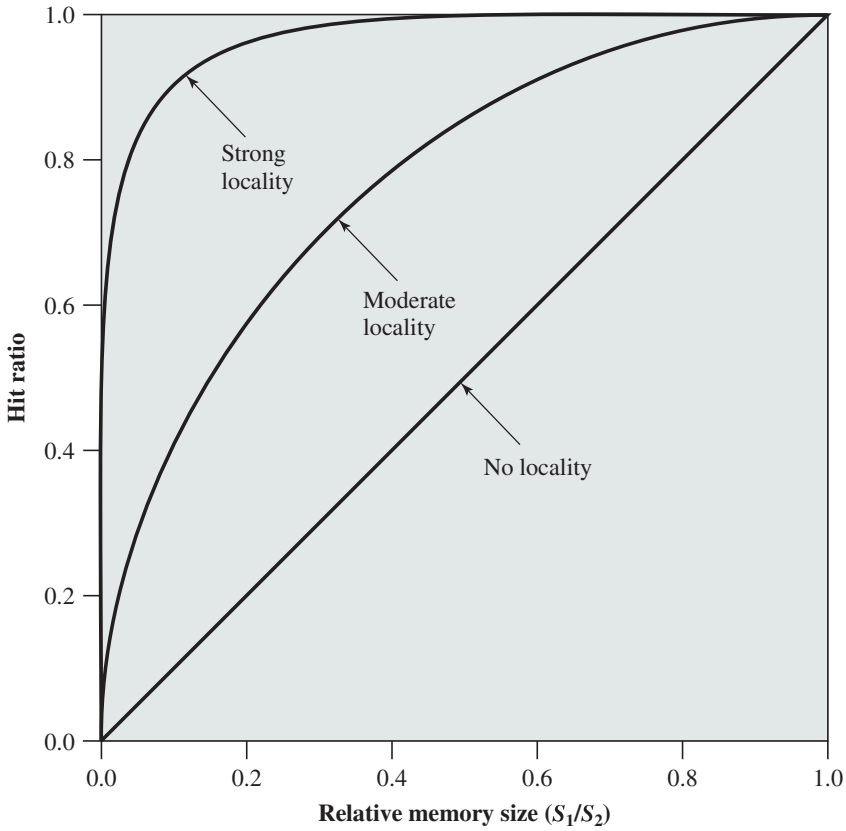


Figure 1.24 **Hit Ratio as a Function of Relative Memory Size**

have shown that rather small cache sizes will yield a hit ratio above 0.75 *regardless of the size of main memory* (e.g., [AGAR89], [PRZY88], [STRE83], and [SMIT82]). A cache in the range of 1K to 128K words is generally adequate, whereas main memory is now typically in the gigabyte range. When we consider virtual memory and disk cache, we will cite other studies that confirm the same phenomenon, namely that a relatively small M1 yields a high value of hit ratio because of locality.

This brings us to the last question listed earlier: Does the relative size of the two memories satisfy the cost requirement? The answer is clearly yes. If we need only a relatively small upper-level memory to achieve good performance, then the average cost per bit of the two levels of memory will approach that of the cheaper lower-level memory.

ISBN 1-256-52023-3

CHAPTER 2

OPERATING SYSTEM OVERVIEW

ISBN 1-256-52023-3

Operating systems are those programs that interface the machine with the applications programs. The main function of these systems is to dynamically allocate the shared system resources to the executing programs. As such, research in this area is clearly concerned with the management and scheduling of memory, processes, and other devices. But the interface with adjacent levels continues to shift with time. Functions that were originally part of the operating system have migrated to the hardware. On the other side, programmed functions extraneous to the problems being solved by the application programs are included in the operating system.

—*What Can Be Automated?: The Computer Science and Engineering Research Study*, MIT Press, 1980

Learning Objectives

After studying this chapter, you should be able to:

- Summarize, at a top level, the key functions of an operating system (OS).
- Discuss the evolution of operating systems for early simple batch systems to modern complex systems.
- Give a brief explanation of each of the major achievements in OS research, as defined in Section 2.3.
- Discuss the key design areas that have been instrumental in the development of modern operating systems.
- Define and discuss virtual machines and virtualization
- Understand the OS design issues raised by the introduction of multiprocessor and multicore organization.
- Understand the basic structure of Windows 7.
- Describe the essential elements of a traditional UNIX system.
- Explain the new features found in modern UNIX systems.
- Discuss Linux and its relationship to UNIX.

We begin our study of operating systems (OSs) with a brief history. This history is itself interesting and also serves the purpose of providing an overview of OS principles. The first section examines the objectives and functions of operating systems. Then we look at how operating systems have evolved from primitive batch systems to sophisticated multitasking, multiuser systems. The remainder of the chapter looks at the history and general characteristics of the two operating systems that serve as examples throughout this book. All of the material in this chapter is covered in greater depth later in the book.

ISBN 1-256-52023-3

2.1 OPERATING SYSTEM OBJECTIVES AND FUNCTIONS

An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware. It can be thought of as having three objectives:

- **Convenience:** An OS makes a computer more convenient to use.
- **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.
- **Ability to evolve:** An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

Let us examine these three aspects of an OS in turn.

The Operating System as a User/Computer Interface

The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion, as depicted in Figure 2.1. The user of those applications, the end user, generally is not concerned with the details of computer hardware. Thus, the end user views a computer system in terms of a set of applications. An application can be expressed in a programming language and is developed by an application programmer. If one were to develop an application program as a set of machine instructions that is completely responsible for controlling the computer hardware, one would be faced with an overwhelmingly complex undertaking. To ease this chore, a set of system programs is provided. Some of these programs are referred to as utilities, or library programs. These implement frequently used functions that assist in program creation, the management of files, and the control of

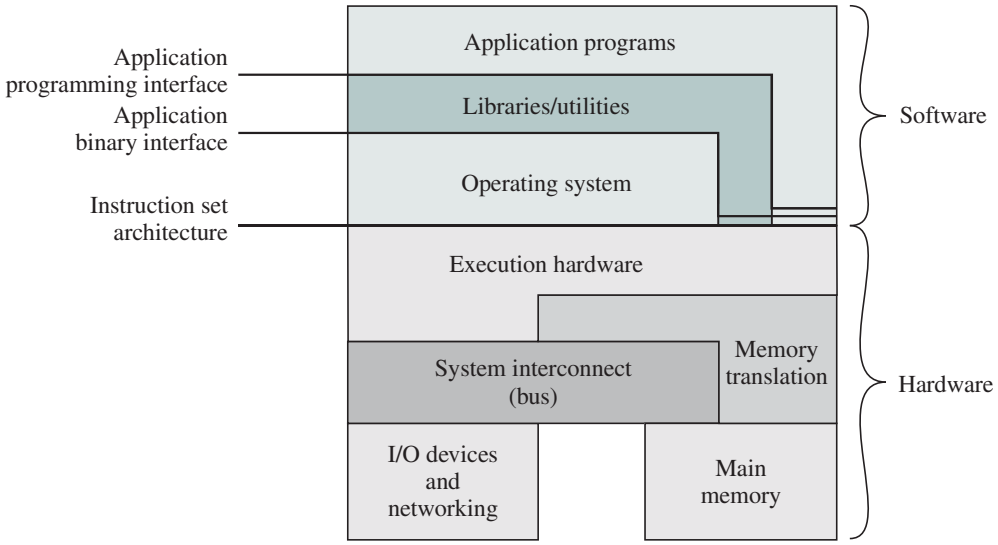


Figure 2.1 Computer Hardware and Software Structure

ISBN 1-256-52023-3

I/O devices. A programmer will make use of these facilities in developing an application, and the application, while it is running, will invoke the utilities to perform certain functions. The most important collection of system programs comprises the OS. The OS masks the details of the hardware from the programmer and provides the programmer with a convenient interface for using the system. It acts as mediator, making it easier for the programmer and for application programs to access and use those facilities and services.

Briefly, the OS typically provides services in the following areas:

- **Program development:** The OS provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs. Typically, these services are in the form of utility programs that, while not strictly part of the core of the OS, are supplied with the OS and are referred to as application program development tools.
- **Program execution:** A number of steps need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared. The OS handles these scheduling duties for the user.
- **Access to I/O devices:** Each I/O device requires its own peculiar set of instructions or control signals for operation. The OS provides a uniform interface that hides these details so that programmers can access such devices using simple reads and writes.
- **Controlled access to files:** For file access, the OS must reflect a detailed understanding of not only the nature of the I/O device (disk drive, tape drive) but also the structure of the data contained in the files on the storage medium. In the case of a system with multiple users, the OS may provide protection mechanisms to control access to the files.
- **System access:** For shared or public systems, the OS controls access to the system as a whole and to specific system resources. The access function must provide protection of resources and data from unauthorized users and must resolve conflicts for resource contention.
- **Error detection and response:** A variety of errors can occur while a computer system is running. These include internal and external hardware errors, such as a memory error, or a device failure or malfunction; and various software errors, such as division by zero, attempt to access forbidden memory location, and inability of the OS to grant the request of an application. In each case, the OS must provide a response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error, to retrying the operation, to simply reporting the error to the application.
- **Accounting:** A good OS will collect usage statistics for various resources and monitor performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance. On a multiuser system, the information can be used for billing purposes.

ISBN 1-256-52023-3

Figure 2.1 also indicates three key interfaces in a typical computer system:

- **Instruction set architecture (ISA)**: The ISA defines the repertoire of machine language instructions that a computer can follow. This interface is the boundary between hardware and software. Note that both application programs and utilities may access the ISA directly. For these programs, a subset of the instruction repertoire is available (user ISA). The OS has access to additional machine language instructions that deal with managing system resources (system ISA).
- **Application binary interface (ABI)**: The ABI defines a standard for binary portability across programs. The ABI defines the system call interface to the operating system and the hardware resources and services available in a system through the user ISA.
- **Application programming interface (API)**: The API gives a program access to the hardware resources and services available in a system through the user ISA supplemented with high-level language (HLL) library calls. Any system calls are usually performed through libraries. Using an API enables application software to be ported easily, through recompilation, to other systems that support the same API.

The Operating System as Resource Manager

A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions. The OS is responsible for managing these resources.

Can we say that it is the OS that controls the movement, storage, and processing of data? From one point of view, the answer is yes: By managing the computer's resources, the OS is in control of the computer's basic functions. But this control is exercised in a curious way. Normally, we think of a control mechanism as something external to that which is controlled, or at least as something that is a distinct and separate part of that which is controlled. (For example, a residential heating system is controlled by a thermostat, which is separate from the heat-generation and heat-distribution apparatus.) This is not the case with the OS, which as a control mechanism is unusual in two respects:

- The OS functions in the same way as ordinary computer software; that is, it is a program or suite of programs executed by the processor.
- The OS frequently relinquishes control and must depend on the processor to allow it to regain control.

Like other computer programs, the OS provides instructions for the processor. The key difference is in the intent of the program. The OS directs the processor in the use of the other system resources and in the timing of its execution of other programs. But in order for the processor to do any of these things, it must cease executing the OS program and execute other programs. Thus, the OS relinquishes control for the processor to do some "useful" work and then resumes control long enough to prepare the processor to do the next piece of work. The mechanisms involved in all this should become clear as the chapter proceeds.

ISBN 1-256-52023-3

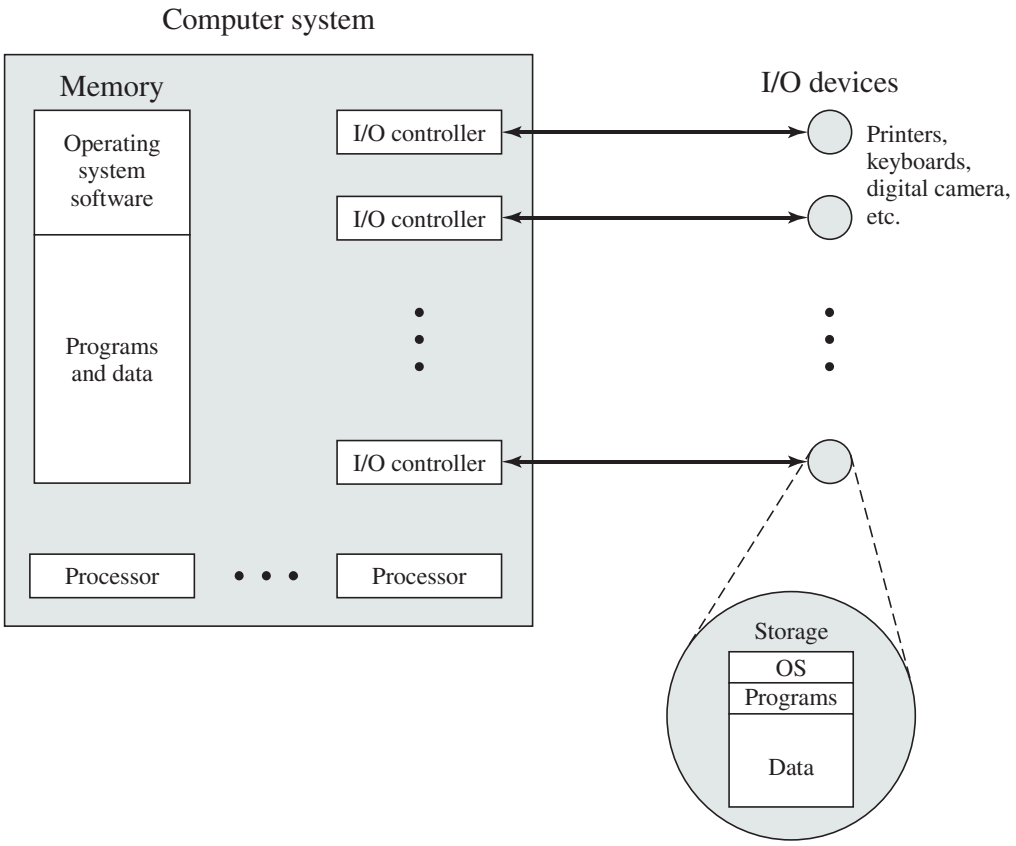


Figure 2.2 The Operating System as Resource Manager

Figure 2.2 suggests the main resources that are managed by the OS. A portion of the OS is in main memory. This includes the **kernel**, or **nucleus**, which contains the most frequently used functions in the OS and, at a given time, other portions of the OS currently in use. The remainder of main memory contains user programs and data. The memory management hardware in the processor and the OS jointly control the allocation of main memory, as we shall see. The OS decides when an I/O device can be used by a program in execution and controls access to and use of files. The processor itself is a resource, and the OS must determine how much processor time is to be devoted to the execution of a particular user program. In the case of a multiple-processor system, this decision must span all of the processors.

Ease of Evolution of an Operating System

A major OS will evolve over time for a number of reasons:

- **Hardware upgrades plus new types of hardware:** For example, early versions of UNIX and the Macintosh OS did not employ a paging mechanism because they were run on processors without paging hardware.[1] Subsequent versions of these operating systems were modified to exploit paging capabilities. Also,

[1]Paging is introduced briefly later in this chapter and is discussed in detail in Chapter 7.

ISBN 1-256-52023-3

the use of graphics terminals and page-mode terminals instead of line-at-a-time scroll mode terminals affects OS design. For example, a graphics terminal typically allows the user to view several applications at the same time through "windows" on the screen. This requires more sophisticated support in the OS.

- **New services:** In response to user demand or in response to the needs of system managers, the OS expands to offer new services. For example, if it is found to be difficult to maintain good performance for users with existing tools, new measurement and control tools may be added to the OS.
- **Fixes:** Any OS has faults. These are discovered over the course of time and fixes are made. Of course, the fix may introduce new faults.

The need to change an OS regularly places certain requirements on its design. An obvious statement is that the system should be modular in construction, with clearly defined interfaces between the modules, and that it should be well documented. For large programs, such as the typical contemporary OS, what might be referred to as straightforward modularization is inadequate [DENN80a]. That is, much more must be done than simply partitioning a program into modules. We return to this topic later in this chapter.

2.2 THE EVOLUTION OF OPERATING SYSTEMS

In attempting to understand the key requirements for an OS and the significance of the major features of a contemporary OS, it is useful to consider how operating systems have evolved over the years.

Serial Processing

With the earliest computers, from the late 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no OS. These computers were run from a console consisting of display lights, toggle switches, some form of input device, and a printer. Programs in machine code were loaded via the input device (e.g., a card reader). If an error halted the program, the error condition was indicated by the lights. If the program proceeded to a normal completion, the output appeared on the printer.

These early systems presented two main problems:

- **Scheduling:** Most installations used a hardcopy sign-up sheet to reserve computer time. Typically, a user could sign up for a block of time in multiples of a half hour or so. A user might sign up for an hour and finish in 45 minutes; this would result in wasted computer processing time. On the other hand, the user might run into problems, not finish in the allotted time, and be forced to stop before resolving the problem.
- **Setup time:** A single program, called a **job**, could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program) and then loading and linking together the object program and common functions. Each of these steps could

ISBN 1-256-52023-3

involve mounting or dismounting tapes or setting up card decks. If an error occurred, the hapless user typically had to go back to the beginning of the setup sequence. Thus, a considerable amount of time was spent just in setting up the program to run.

This mode of operation could be termed *serial processing*, reflecting the fact that users have access to the computer in series. Over time, various system software tools were developed to attempt to make serial processing more efficient. These include libraries of common functions, linkers, loaders, debuggers, and I/O driver routines that were available as common software for all users.

Simple Batch Systems

Early computers were very expensive, and therefore it was important to maximize processor utilization. The wasted time due to scheduling and setup time was unacceptable.

To improve utilization, the concept of a batch OS was developed. It appears that the first batch OS (and the first OS of any kind) was developed in the mid-1950s by General Motors for use on an IBM 701 [WEIZ81]. The concept was subsequently refined and implemented on the IBM 704 by a number of IBM customers. By the early 1960s, a number of vendors had developed batch operating systems for their computer systems. IBSYS, the IBM OS for the 7090/7094 computers, is particularly notable because of its widespread influence on other systems.

The central idea behind the simple batch-processing scheme is the use of a piece of software known as the **monitor**. With this type of OS, the user no longer has direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor. Each program is constructed to branch back to the monitor when it completes processing, at which point the monitor automatically begins loading the next program.

To understand how this scheme works, let us look at it from two points of view: that of the monitor and that of the processor.

- **Monitor point of view:** The monitor controls the sequence of events. For this to be so, much of the monitor must always be in main memory and available for execution (Figure 2.3). That portion is referred to as the **resident monitor**. The rest of the monitor consists of utilities and common functions that are loaded as subroutines to the user program at the beginning of any job that requires them. The monitor reads in jobs one at a time from the input device (typically a card reader or magnetic tape drive). As it is read in, the current job is placed in the user program area, and control is passed to this job. When the job is completed, it returns control to the monitor, which immediately reads in the next job. The results of each job are sent to an output device, such as a printer, for delivery to the user.
- **Processor point of view:** At a certain point, the processor is executing instructions from the portion of main memory containing the monitor. These instructions cause the next job to be read into another portion of main

ISBN 1-256-52023-3

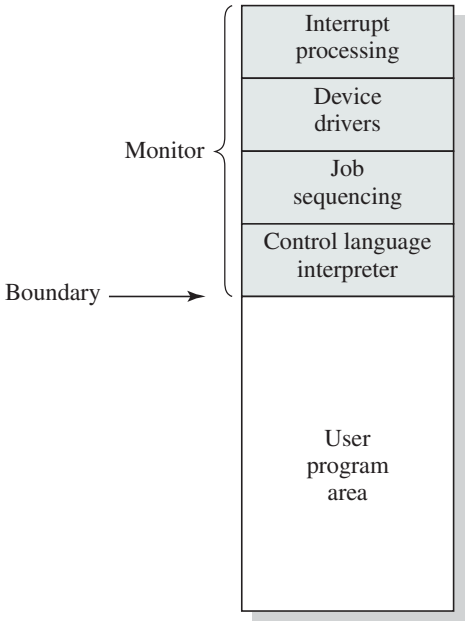


Figure 2.3 Memory Layout for a Resident Monitor

memory. Once a job has been read in, the processor will encounter a branch instruction in the monitor that instructs the processor to continue execution at the start of the user program. The processor will then execute the instructions in the user program until it encounters an ending or error condition. Either event causes the processor to fetch its next instruction from the monitor program. Thus the phrase "control is passed to a job" simply means that the processor is now fetching and executing instructions in a user program, and "control is returned to the monitor" means that the processor is now fetching and executing instructions from the monitor program.

The monitor performs a scheduling function: A batch of jobs is queued up, and jobs are executed as rapidly as possible, with no intervening idle time. The monitor improves job setup time as well. With each job, instructions are included in a primitive form of **job control language (JCL)**. This is a special type of programming language used to provide instructions to the monitor. A simple example is that of a user submitting a program written in the programming language FORTRAN plus some data to be used by the program. All FORTRAN instructions and data are on a separate punched card or a separate record on tape. In addition to FORTRAN and data lines, the job includes job control instructions, which are denoted by the beginning `$`. The overall format of the job looks like this:

```
$JOB
$FTN
.
.
.
```

FORTRAN instructions

ISBN 1-256-52023-3

```
$LOAD
$RUN
•
•          } Data
•
$END
```

To execute this job, the monitor reads the `$FTN` line and loads the appropriate language compiler from its mass storage (usually tape). The compiler translates the user's program into object code, which is stored in memory or mass storage. If it is stored in memory, the operation is referred to as "compile, load, and go." If it is stored on tape, then the `$LOAD` instruction is required. This instruction is read by the monitor, which regains control after the compile operation. The monitor invokes the loader, which loads the object program into memory (in place of the compiler) and transfers control to it. In this manner, a large segment of main memory can be shared among different subsystems, although only one such subsystem could be executing at a time.

During the execution of the user program, any input instruction causes one line of data to be read. The input instruction in the user program causes an input routine that is part of the OS to be invoked. The input routine checks to make sure that the program does not accidentally read in a JCL line. If this happens, an error occurs and control transfers to the monitor. At the completion of the user job, the monitor will scan the input lines until it encounters the next JCL instruction. Thus, the system is protected against a program with too many or too few data lines.

The monitor, or batch OS, is simply a computer program. It relies on the ability of the processor to fetch instructions from various portions of main memory to alternately seize and relinquish control. Certain other hardware features are also desirable:

- **Memory protection:** While the user program is executing, it must not alter the memory area containing the monitor. If such an attempt is made, the processor hardware should detect an error and transfer control to the monitor. The monitor would then abort the job, print out an error message, and load in the next job.
- **Timer:** A timer is used to prevent a single job from monopolizing the system. The timer is set at the beginning of each job. If the timer expires, the user program is stopped, and control returns to the monitor.
- **Privileged instructions:** Certain machine level instructions are designated privileged and can be executed only by the monitor. If the processor encounters such an instruction while executing a user program, an error occurs causing control to be transferred to the monitor. Among the privileged instructions are I/O instructions, so that the monitor retains control of all I/O devices. This prevents, for example, a user program from accidentally reading job control instructions from the next job. If a user program wishes to perform I/O, it must request that the monitor perform the operation for it.

ISBN 1-256-52023-3

- **Interrupts:** Early computer models did not have this capability. This feature gives the OS more flexibility in relinquishing control to and regaining control from user programs.

Considerations of memory protection and privileged instructions lead to the concept of modes of operation. A user program executes in a **user mode**, in which certain areas of memory are protected from the user's use and in which certain instructions may not be executed. The monitor executes in a system mode, or what has come to be called **kernel mode**, in which privileged instructions may be executed and in which protected areas of memory may be accessed.

Of course, an OS can be built without these features. But computer vendors quickly learned that the results were chaos, and so even relatively primitive batch operating systems were provided with these hardware features.

With a batch OS, processor time alternates between execution of user programs and execution of the monitor. There have been two sacrifices: Some main memory is now given over to the monitor and some processor time is consumed by the monitor. Both of these are forms of overhead. Despite this overhead, the simple batch system improves utilization of the computer.

Multiprogrammed Batch Systems

Even with the automatic job sequencing provided by a simple batch OS, the processor is often idle. The problem is that I/O devices are slow compared to the processor. Figure 2.4 details a representative calculation. The calculation concerns a program that processes a file of records and performs, on average, 100 machine instructions per record. In this example, the computer spends over 96% of its time waiting for I/O devices to finish transferring data to and from the file. Figure 2.5a illustrates this situation, where we have a single program, referred to as uniprogramming. The processor spends a certain amount of time executing, until it reaches an I/O instruction. It must then wait until that I/O instruction concludes before proceeding.

This inefficiency is not necessary. We know that there must be enough memory to hold the OS (resident monitor) and one user program. Suppose that there is room for the OS and two user programs. When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O (Figure 2.5b). Furthermore, we might expand memory to hold three, four, or more programs and switch among all of them (Figure 2.5c). The approach is known as **multiprogramming**, or **multitasking**. It is the central theme of modern operating systems.

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
Total	31 μs

$$\text{Percent CPU Utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

Figure 2.4 System Utilization Example

ISBN 1-256-52023-3

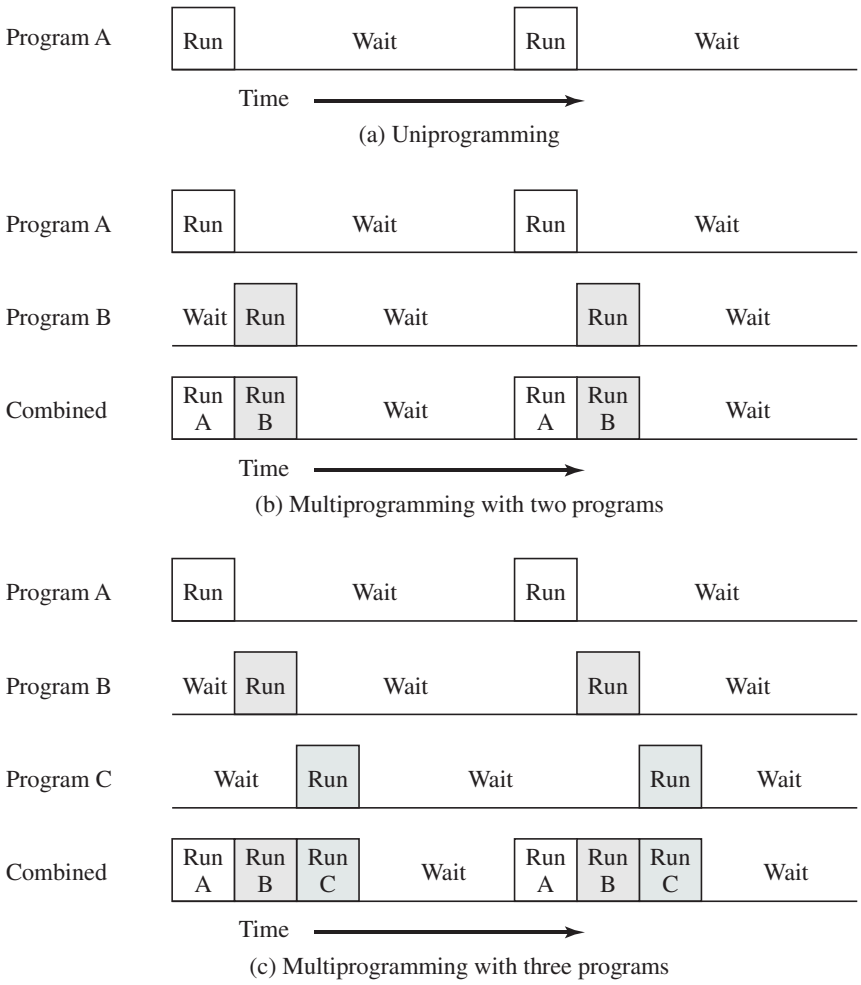


Figure 2.5 **Multiprogramming Example**

To illustrate the benefit of multiprogramming, we give a simple example. Consider a computer with 250 Mbytes of available memory (not used by the OS), a disk, a terminal, and a printer. Three programs, JOB1, JOB2, and JOB3, are submitted for execution at the same time, with the attributes listed in Table 2.1. We assume minimal processor requirements for JOB2 and JOB3 and continuous disk and printer use by JOB3. For a simple batch environment, these jobs will be executed in sequence. Thus, JOB1 completes in 5 minutes. JOB2 must wait until

Table 2.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

ISBN 1-256-52023-3

Table 2.2 Effects of Multiprogramming on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

the 5 minutes are over and then completes 15 minutes after that. JOB3 begins after 20 minutes and completes at 30 minutes from the time it was initially submitted. The average resource utilization, throughput, and response times are shown in the uniprogramming column of Table 2.2. Device-by-device utilization is illustrated in Figure 2.6a. It is evident that there is gross underutilization for all resources when averaged over the required 30-minute time period.

Now suppose that the jobs are run concurrently under a multiprogramming OS. Because there is little resource contention between the jobs, all three can run in nearly minimum time while coexisting with the others in the computer (assuming that JOB2 and JOB3 are allotted enough processor time to keep their input and output operations active). JOB1 will still require 5 minutes to complete, but at the end of that time, JOB2 will be one-third finished and JOB3 half finished. All three jobs will have finished within 15 minutes. The improvement is evident when examining the multiprogramming column of Table 2.2, obtained from the histogram shown in Figure 2.6b.

As with a simple batch system, a multiprogramming batch system must rely on certain computer hardware features. The most notable additional feature that is useful for multiprogramming is the hardware that supports I/O interrupts and DMA (direct memory access). With interrupt-driven I/O or DMA, the processor can issue an I/O command for one job and proceed with the execution of another job while the I/O is carried out by the device controller. When the I/O operation is complete, the processor is interrupted and control is passed to an interrupt-handling program in the OS. The OS will then pass control to another job.

Multiprogramming operating systems are fairly sophisticated compared to single-program, or **uniprogramming**, systems. To have several jobs ready to run, they must be kept in main memory, requiring some form of **memory management**. In addition, if several jobs are ready to run, the processor must decide which one to run, this decision requires an algorithm for scheduling. These concepts are discussed later in this chapter.

Time-Sharing Systems

With the use of multiprogramming, batch processing can be quite efficient. However, for many jobs, it is desirable to provide a mode in which the user interacts directly with the computer. Indeed, for some jobs, such as transaction processing, an interactive mode is essential.

ISBN 1-256-52023-3

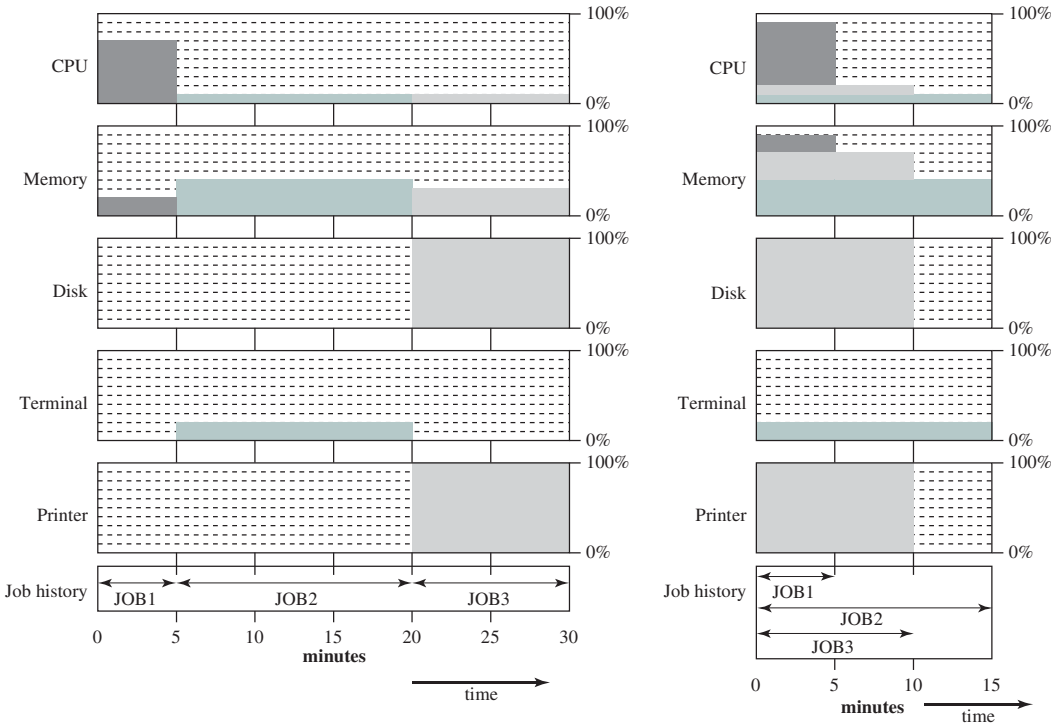


Figure 2.6 **Utilization Histograms**

ISBN 1-256-52023-3

Today, the requirement for an interactive computing facility can be, and often is, met by the use of a dedicated personal computer or workstation. That option was not available in the 1960s, when most computers were big and costly. Instead, time sharing was developed.

Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can also be used to handle multiple interactive jobs. In this latter case, the technique is referred to as **time sharing**, because processor time is shared among multiple users. In a time-sharing system, multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation. Thus, if there are n users actively requesting service at one time, each user will only see on the average $1/n$ of the effective computer capacity, not counting OS overhead. However, given the relatively slow human reaction time, the response time on a properly designed system should be similar to that on a dedicated computer.

Both batch processing and time sharing use multiprogramming. The key differences are listed in Table 2.3.

One of the first time-sharing operating systems to be developed was the Compatible Time-Sharing System (CTSS) [CORB62], developed at MIT by a group known as Project MAC (Machine-Aided Cognition, or Multiple-Access Computers). The system was first developed for the IBM 709 in 1961 and later transferred to an IBM 7094.

Compared to later systems, CTSS is primitive. The system ran on a computer with 32,000 36-bit words of main memory, with the resident monitor consuming 5000 of that. When control was to be assigned to an interactive user, the user's program and data were loaded into the remaining 27,000 words of main memory. A program was always loaded to start at the location of the 5000th word; this simplified both the monitor and memory management. A system clock generated interrupts at a rate of approximately one every 0.2 seconds. At each clock interrupt, the OS regained control and could assign the processor to another user. This technique is known as **time slicing**. Thus, at regular time intervals, the current user would be preempted and another user loaded in. To preserve the old user program status for later resumption, the old user programs and data were written out to disk before the new user programs and data were read in. Subsequently, the old user program code and data were restored in main memory when that program was next given a turn.

To minimize disk traffic, user memory was only written out when the incoming program would overwrite it. This principle is illustrated in Figure 2.7. Assume that there are four interactive users with the following memory requirements, in words:

- JOB1: 15,000
- JOB2: 20,000

Table 2.3 Batch Multiprogramming versus Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

ISBN 1-256-52023-3

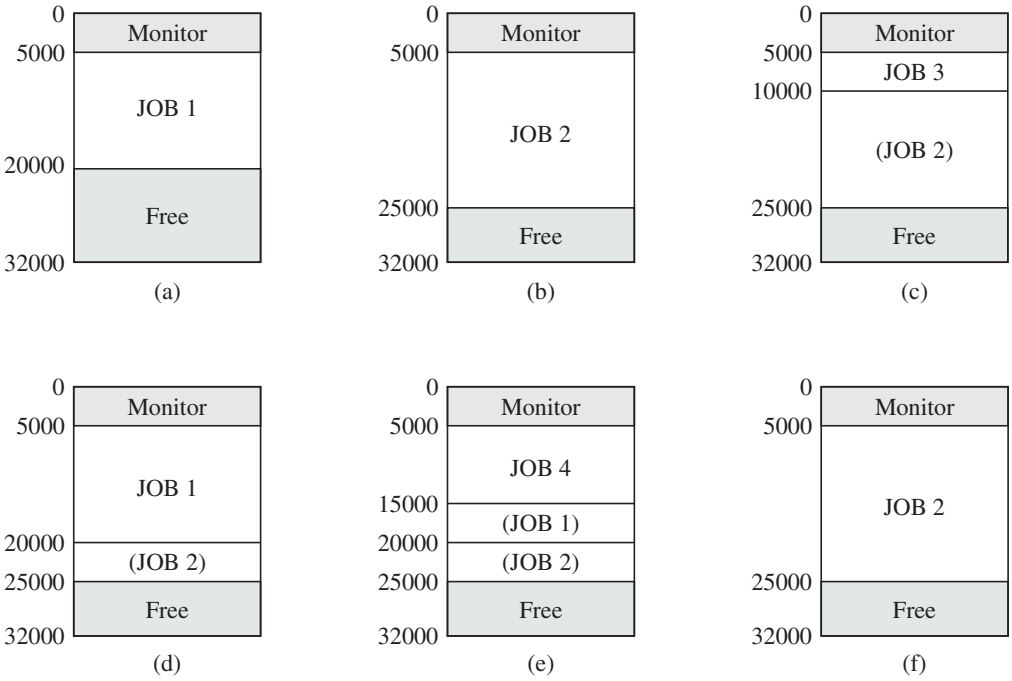


Figure 2.7 CTSS Operation

- JOB3: 5000
- JOB4: 10,000

Initially, the monitor loads JOB1 and transfers control to it (a). Later, the monitor decides to transfer control to JOB2. Because JOB2 requires more memory than JOB1, JOB1 must be written out first, and then JOB2 can be loaded (b). Next, JOB3 is loaded in to be run. However, because JOB3 is smaller than JOB2, a portion of JOB2 can remain in memory, reducing disk write time (c). Later, the monitor decides to transfer control back to JOB1. An additional portion of JOB2 must be written out when JOB1 is loaded back into memory (d). When JOB4 is loaded, part of JOB1 and the portion of JOB2 remaining in memory are retained (e). At this point, if either JOB1 or JOB2 is activated, only a partial load will be required. In this example, it is JOB2 that runs next. This requires that JOB4 and the remaining resident portion of JOB1 be written out and that the missing portion of JOB2 be read in (f).

The CTSS approach is primitive compared to present-day time sharing, but it was effective. It was extremely simple, which minimized the size of the monitor. Because a job was always loaded into the same locations in memory, there was no need for relocation techniques at load time (discussed subsequently). The technique of only writing out what was necessary minimized disk activity. Running on the 7094, CTSS supported a maximum of 32 users.

Time sharing and multiprogramming raise a host of new problems for the OS. If multiple jobs are in memory, then they must be protected from interfering with each other by, for example, modifying each other's data. With multiple interactive users, the file system must be protected so that only authorized users have access

ISBN 1-256-52023-3

to a particular file. The contention for resources, such as printers and mass storage devices, must be handled. These and other problems, with possible solutions, will be encountered throughout this text.

2.3 MAJOR ACHIEVEMENTS

Operating systems are among the most complex pieces of software ever developed. This reflects the challenge of trying to meet the difficult and in some cases competing objectives of convenience, efficiency, and ability to evolve. [DENN80a] proposes that there have been four major theoretical advances in the development of operating systems:

- Processes
- Memory management
- Information protection and security
- Scheduling and resource management

Each advance is characterized by principles, or abstractions, developed to meet difficult practical problems. Taken together, these five areas span many of the key design and implementation issues of modern operating systems. The brief review of these five areas in this section serves as an overview of much of the rest of the text.

The Process

Central to the design of operating systems is the concept of *process*. This term was first used by the designers of Multics in the 1960s [DALE68]. It is a somewhat more general term than *job*. Many definitions have been given for the term *process*, including

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

This concept should become clearer as we proceed.

Three major lines of computer system development created problems in timing and synchronization that contributed to the development of the concept of the process: multiprogramming batch operation, time sharing, and real-time transaction systems. As we have seen, multiprogramming was designed to keep the processor and I/O devices, including storage devices, simultaneously busy to achieve maximum efficiency. The key mechanism is this: In response to signals indicating the completion of I/O transactions, the processor is switched among the various programs residing in main memory.

ISBN 1-256-52023-3

A second line of development was general-purpose time sharing. Here, the key design objective is to be responsive to the needs of the individual user and yet, for cost reasons, be able to support many users simultaneously. These goals are compatible because of the relatively slow reaction time of the user. For example, if a typical user needs an average of 2 seconds of processing time per minute, then close to 30 such users should be able to share the same system without noticeable interference. Of course, OS overhead must be factored into such calculations.

A third important line of development has been real-time transaction processing systems. In this case, a number of users are entering queries or updates against a database. An example is an airline reservation system. The key difference between the transaction processing system and the time-sharing system is that the former is limited to one or a few applications, whereas users of a time-sharing system can engage in program development, job execution, and the use of various applications. In both cases, system response time is paramount.

The principal tool available to system programmers in developing the early multiprogramming and multiuser interactive systems was the interrupt. The activity of any job could be suspended by the occurrence of a defined event, such as an I/O completion. The processor would save some sort of context (e.g., program counter and other registers) and branch to an interrupt-handling routine, which would determine the nature of the interrupt, process the interrupt, and then resume user processing with the interrupted job or some other job.

The design of the system software to coordinate these various activities turned out to be remarkably difficult. With many jobs in progress at any one time, each of which involved numerous steps to be performed in sequence, it became impossible to analyze all of the possible combinations of sequences of events. In the absence of some systematic means of coordination and cooperation among activities, programmers resorted to ad hoc methods based on their understanding of the environment that the OS had to control. These efforts were vulnerable to subtle programming errors whose effects could be observed only when certain relatively rare sequences of actions occurred. These errors were difficult to diagnose because they needed to be distinguished from application software errors and hardware errors. Even when the error was detected, it was difficult to determine the cause, because the precise conditions under which the errors appeared were very hard to reproduce. In general terms, there are four main causes of such errors [DENN80a]:

- **Improper synchronization:** It is often the case that a routine must be suspended awaiting an event elsewhere in the system. For example, a program that initiates an I/O read must wait until the data are available in a buffer before proceeding. In such cases, a signal from some other routine is required. Improper design of the signaling mechanism can result in signals being lost or duplicate signals being received.
- **Failed mutual exclusion:** It is often the case that more than one user or program will attempt to make use of a shared resource at the same time. For example, two users may attempt to edit the same file at the same time. If these accesses are not controlled, an error can occur. There must be some sort of mutual exclusion mechanism that permits only one routine at a time to perform an update against the file. The implementation of such mutual

ISBN 1-256-52023-3

exclusion is difficult to verify as being correct under all possible sequences of events.

- **Nondeterminate program operation:** The results of a particular program normally should depend only on the input to that program and not on the activities of other programs in a shared system. But when programs share memory, and their execution is interleaved by the processor, they may interfere with each other by overwriting common memory areas in unpredictable ways. Thus, the order in which various programs are scheduled may affect the outcome of any particular program.
- **Deadlocks:** It is possible for two or more programs to be hung up waiting for each other. For example, two programs may each require two I/O devices to perform some operation (e.g., disk to tape copy). One of the programs has seized control of one of the devices and the other program has control of the other device. Each is waiting for the other program to release the desired resource. Such a deadlock may depend on the chance timing of resource allocation and release.

What is needed to tackle these problems is a systematic way to monitor and control the various programs executing on the processor. The concept of the process provides the foundation. We can think of a process as consisting of three components:

- An executable program
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context of the program

This last element is essential. The **execution context**, or **process state**, is the internal data by which the OS is able to supervise and control the process. This internal information is separated from the process, because the OS has information not permitted to the process. The context includes all of the information that the OS needs to manage the process and that the processor needs to execute the process properly. The context includes the contents of the various processor registers, such as the program counter and data registers. It also includes information of use to the OS, such as the priority of the process and whether the process is waiting for the completion of a particular I/O event.

Figure 2.8 indicates a way in which processes may be managed. Two processes, A and B, exist in portions of main memory. That is, a block of memory is allocated to each process that contains the program, data, and context information. Each process is recorded in a process list built and maintained by the OS. The process list contains one entry for each process, which includes a pointer to the location of the block of memory that contains the process. The entry may also include part or all of the execution context of the process. The remainder of the execution context is stored elsewhere, perhaps with the process itself (as indicated in Figure 2.8) or frequently in a separate region of memory. The process index register contains the index into the process list of the process currently controlling the processor. The program counter points to the next instruction in that process to be executed. The base and limit registers define the region in memory occupied

ISBN 1-256-52023-3

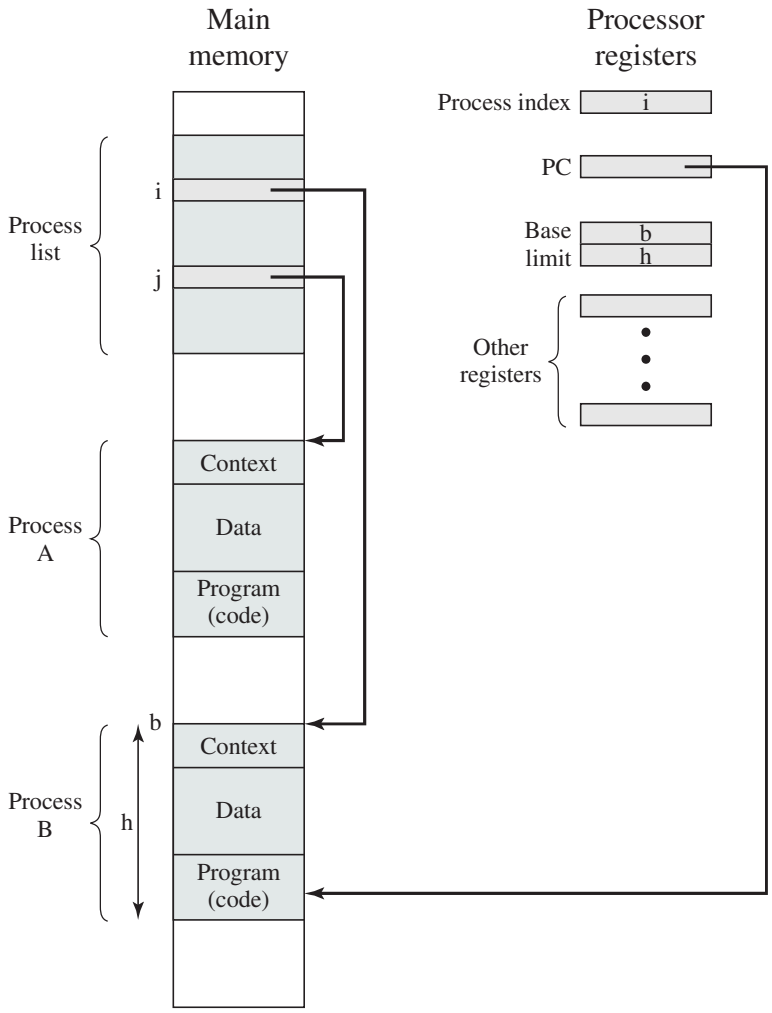


Figure 2.8 Typical Process Implementation

by the process: The base register is the starting address of the region of memory and the limit is the size of the region (in bytes or words). The program counter and all data references are interpreted relative to the base register and must not exceed the value in the limit register. This prevents interprocess interference.

In Figure 2.8, the process index register indicates that process B is executing. Process A was previously executing but has been temporarily interrupted. The contents of all the registers at the moment of A's interruption were recorded in its execution context. Later, the OS can perform a process switch and resume execution of process A. The process switch consists of storing the context of B and restoring the context of A. When the program counter is loaded with a value pointing into A's program area, process A will automatically resume execution.

Thus, the process is realized as a data structure. A process can either be executing or awaiting execution. The entire **state** of the process at any instant is contained in its context. This structure allows the development of powerful techniques for ensuring coordination and cooperation among processes. New features can be designed and incorporated into the OS (e.g., priority) by expanding the context to

ISBN 1-256-52023-3

include any new information needed to support the feature. Throughout this book, we will see a number of examples where this process structure is employed to solve the problems raised by multiprogramming and resource sharing.

A final point, which we introduce briefly here, is the concept of **thread**. In essence, a single process, which is assigned certain resources, can be broken up into multiple, concurrent threads that execute cooperatively to perform the work of the process. This introduces a new level of parallel activity to be managed by the hardware and software.

Memory Management

The needs of users can be met best by a computing environment that supports modular programming and the flexible use of data. System managers need efficient and orderly control of storage allocation. The OS, to satisfy these requirements, has five principal storage management responsibilities:

- **Process isolation:** The OS must prevent independent processes from interfering with each other's memory, both data and instructions.
- **Automatic allocation and management:** Programs should be dynamically allocated across the memory hierarchy as required. Allocation should be transparent to the programmer. Thus, the programmer is relieved of concerns relating to memory limitations, and the OS can achieve efficiency by assigning memory to jobs only as needed.
- **Support of modular programming:** Programmers should be able to define program modules, and to create, destroy, and alter the size of modules dynamically.
- **Protection and access control:** Sharing of memory, at any level of the memory hierarchy, creates the potential for one program to address the memory space of another. This is desirable when sharing is needed by particular applications. At other times, it threatens the integrity of programs and even of the OS itself. The OS must allow portions of memory to be accessible in various ways by various users.
- **Long-term storage:** Many application programs require means for storing information for extended periods of time, after the computer has been powered down.

Typically, operating systems meet these requirements with virtual memory and file system facilities. The file system implements a long-term store, with information stored in named objects, called files. The file is a convenient concept for the programmer and is a useful unit of access control and protection for the OS.

Virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available. Virtual memory was conceived to meet the requirement of having multiple user jobs reside in main memory concurrently, so that there would not be a hiatus between the execution of successive processes while one process was written out to secondary store and the successor process was read in. Because processes vary in size, if the processor switches among a number of processes it is difficult to pack them compactly into main memory. Paging systems were introduced, which allow

ISBN 1-256-52023-3

processes to be comprised of a number of fixed-size blocks, called pages. A program references a word by means of a **virtual address** consisting of a page number and an offset within the page. Each page of a process may be located anywhere in main memory. The paging system provides for a dynamic mapping between the virtual address used in the program and a **real address**, or physical address, in main memory.

With dynamic mapping hardware available, the next logical step was to eliminate the requirement that all pages of a process reside in main memory simultaneously. All the pages of a process are maintained on disk. When a process is executing, some of its pages are in main memory. If reference is made to a page that is not in main memory, the memory management hardware detects this and arranges for the missing page to be loaded. Such a scheme is referred to as **virtual memory** and is depicted in Figure 2.9.

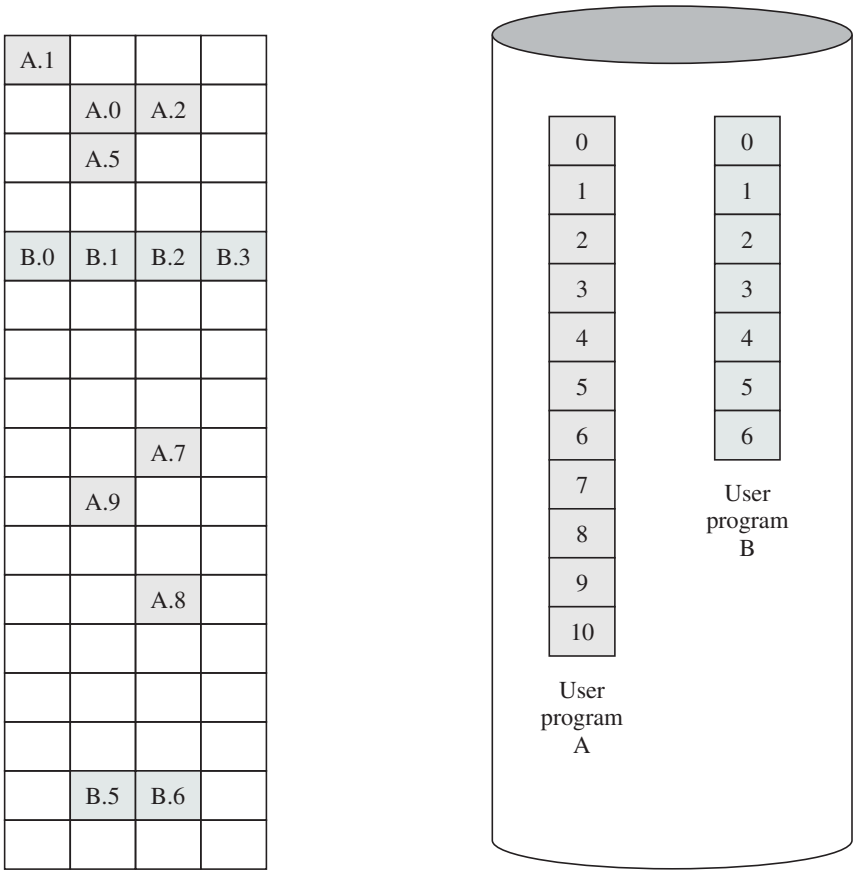


Figure 2.9 Virtual Memory Concepts

ISBN 1-256-52023-3

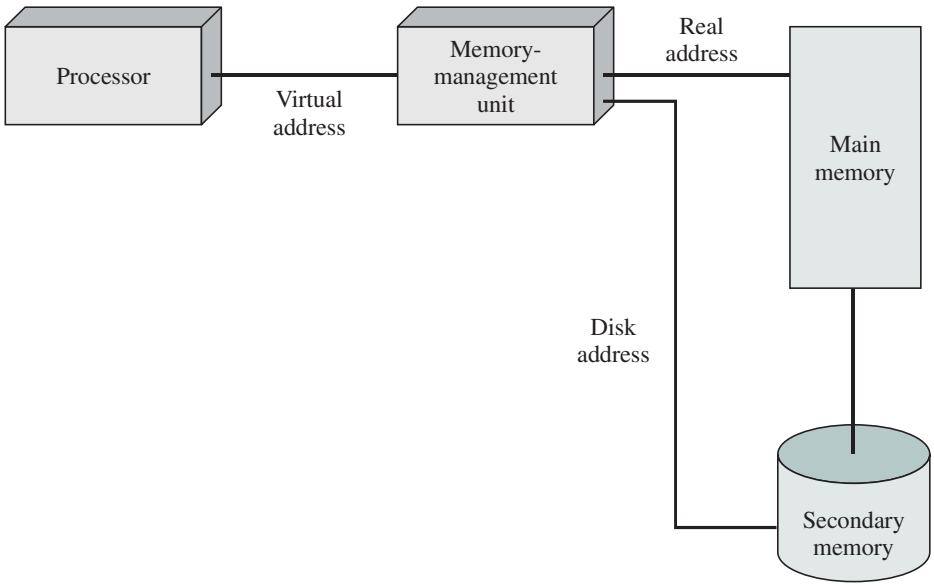


Figure 2.10 Virtual Memory Addressing

The processor hardware, together with the OS, provides the user with a "virtual processor" that has access to a virtual memory. This memory may be a linear address space or a collection of segments, which are variable-length blocks of contiguous addresses. In either case, programming language instructions can reference program and data locations in the virtual memory area. Process isolation can be achieved by giving each process a unique, nonoverlapping virtual memory. Memory sharing can be achieved by overlapping portions of two virtual memory spaces. Files are maintained in a long-term store. Files and portions of files may be copied into the virtual memory for manipulation by programs.

Figure 2.10 highlights the addressing concerns in a virtual memory scheme. Storage consists of directly addressable (by machine instructions) main memory and lower-speed auxiliary memory that is accessed indirectly by loading blocks into main memory. Address translation hardware (memory management unit) is interposed between the processor and memory. Programs reference locations using virtual addresses, which are mapped into real main memory addresses. If a reference is made to a virtual address not in real memory, then a portion of the contents of real memory is swapped out to auxiliary memory and the desired block of data is swapped in. During this activity, the process that generated the address reference must be suspended. The OS designer needs to develop an address translation mechanism that generates little overhead and a storage allocation policy that minimizes the traffic between memory levels.

Information Protection and Security

The growth in the use of time-sharing systems and, more recently, computer networks has brought with it a growth in concern for the protection of information. The nature of the threat that concerns an organization will vary greatly depending on the circumstances. However, there are some general-purpose tools that can be

ISBN 1-256-52023-3

built into computers and operating systems that support a variety of protection and security mechanisms. In general, we are concerned with the problem of controlling access to computer systems and the information stored in them.

Much of the work in security and protection as it relates to operating systems can be roughly grouped into four categories:

- **Availability:** Concerned with protecting the system against interruption.
- **Confidentiality:** Assures that users cannot read data for which access is unauthorized.
- **Data integrity:** Protection of data from unauthorized modification.
- **Authenticity:** Concerned with the proper verification of the identity of users and the validity of messages or data.

Scheduling and Resource Management

A key responsibility of the OS is to manage the various resources available to it (main memory space, I/O devices, processors) and to schedule their use by the various active processes. Any resource allocation and scheduling policy must consider three factors:

- **Fairness:** Typically, we would like all processes that are competing for the use of a particular resource to be given approximately equal and fair access to that resource. This is especially so for jobs of the same class, that is, jobs of similar demands.
- **Differential responsiveness:** On the other hand, the OS may need to discriminate among different classes of jobs with different service requirements. The OS should attempt to make allocation and scheduling decisions to meet the total set of requirements. The OS should also make these decisions dynamically. For example, if a process is waiting for the use of an I/O device, the OS may wish to schedule that process for execution as soon as possible to free up the device for later demands from other processes.
- **Efficiency:** The OS should attempt to maximize throughput, minimize response time, and, in the case of time sharing, accommodate as many users as possible. These criteria conflict; finding the right balance for a particular situation is an ongoing problem for OS research.

Scheduling and resource management are essentially operations-research problems and the mathematical results of that discipline can be applied. In addition, measurement of system activity is important to be able to monitor performance and make adjustments.

Figure 2.11 suggests the major elements of the OS involved in the scheduling of processes and the allocation of resources in a multiprogramming environment. The OS maintains a number of queues, each of which is simply a list of processes waiting for some resource. The short-term queue consists of processes that are in main memory (or at least an essential minimum portion of each is in main memory) and are ready to run as soon as the processor is made available. Any one of these processes could use the processor next. It is up to the short-term scheduler, or

ISBN 1-256-52023-3

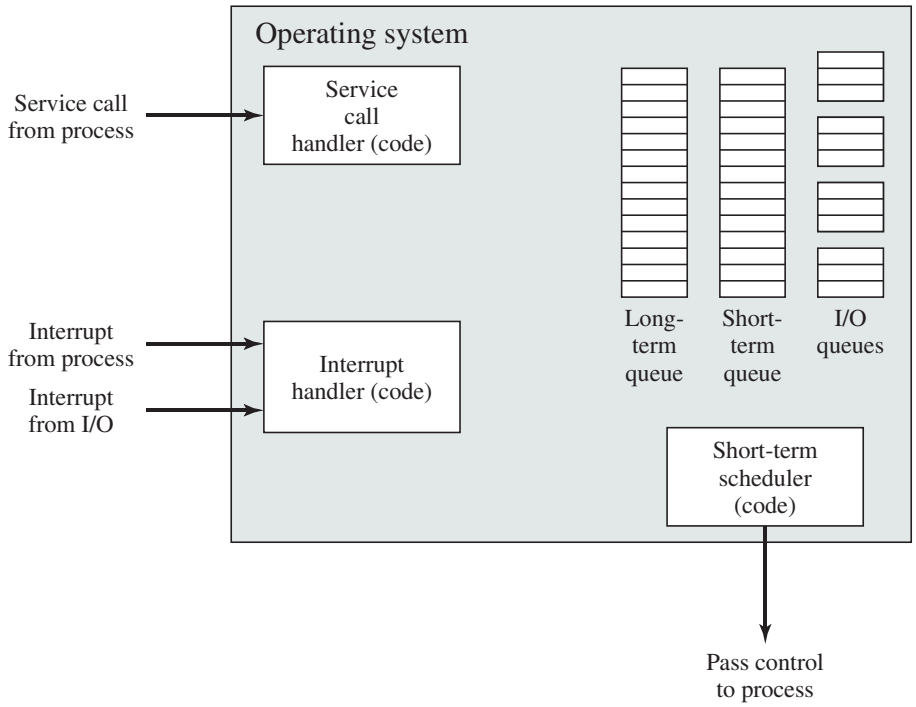


Figure 2.11 **Key Elements of an Operating System for Multiprogramming**

dispatcher, to pick one. A common strategy is to give each process in the queue some time in turn; this is referred to as a **round-robin** technique. In effect, the round-robin technique employs a circular queue. Another strategy is to assign priority levels to the various processes, with the scheduler selecting processes in priority order.

The long-term queue is a list of new jobs waiting to use the processor. The OS adds jobs to the system by transferring a process from the long-term queue to the short-term queue. At that time, a portion of main memory must be allocated to the incoming process. Thus, the OS must be sure that it does not overcommit memory or processing time by admitting too many processes to the system. There is an I/O queue for each I/O device. More than one process may request the use of the same I/O device. All processes waiting to use each device are lined up in that device's queue. Again, the OS must determine which process to assign to an available I/O device.

The OS receives control of the processor at the interrupt handler if an interrupt occurs. A process may specifically invoke some OS service, such as an I/O device handler by means of a service call. In this case, a service call handler is the entry point into the OS. In any case, once the interrupt or service call is handled, the short-term scheduler is invoked to pick a process for execution.

The foregoing is a functional description; details and modular design of this portion of the OS will differ in various systems. Much of the research and development effort in operating systems has been directed at picking algorithms and data structures for this function that provide fairness, differential responsiveness, and efficiency.

ISBN 1-256-52023-3

2.4 DEVELOPMENTS LEADING TO MODERN OPERATING SYSTEMS

Over the years, there has been a gradual evolution of OS structure and capabilities. However, in recent years a number of new design elements have been introduced into both new operating systems and new releases of existing operating systems that create a major change in the nature of operating systems. These modern operating systems respond to new developments in hardware, new applications, and new security threats. Among the key hardware drivers are multiprocessor systems, greatly increased processor speed, high-speed network attachments, and increasing size and variety of memory storage devices. In the application arena, multimedia applications, Internet and Web access, and client/server computing have influenced OS design. With respect to security, Internet access to computers has greatly increased the potential threat and increasingly sophisticated attacks, such as viruses, worms, and hacking techniques, have had a profound impact on OS design.

The rate of change in the demands on operating systems requires not just modifications and enhancements to existing architectures but new ways of organizing the OS. A wide range of different approaches and design elements has been tried in both experimental and commercial operating systems, but much of the work fits into the following categories:

- Microkernel architecture
- Multithreading
- Symmetric multiprocessing
- Distributed operating systems
- Object-oriented design

Most operating systems, until recently, featured a large **monolithic kernel**. Most of what is thought of as OS functionality is provided in these large kernels, including scheduling, file system, networking, device drivers, memory management, and more. Typically, a monolithic kernel is implemented as a single process, with all elements sharing the same address space. A **microkernel architecture** assigns only a few essential functions to the kernel, including address spaces, interprocess communication (IPC), and basic scheduling. Other OS services are provided by processes, sometimes called servers, that run in user mode and are treated like any other application by the microkernel. This approach decouples kernel and server development. Servers may be customized to specific application or environment requirements. The microkernel approach simplifies implementation, provides flexibility, and is well suited to a distributed environment. In essence, a microkernel interacts with local and remote server processes in the same way, facilitating construction of distributed systems.

Multithreading is a technique in which a process, executing an application, is divided into threads that can run concurrently. We can make the following distinction:

- **Thread:** A dispatchable unit of work. It includes a processor context (which includes the program counter and stack pointer) and its own data area for a

ISBN 1-256-52023-3

stack (to enable subroutine branching). A thread executes sequentially and is interruptable so that the processor can turn to another thread.

- **Process:** A collection of one or more threads and associated system resources (such as memory containing both code and data, open files, and devices). This corresponds closely to the concept of a program in execution. By breaking a single application into multiple threads, the programmer has great control over the modularity of the application and the timing of application-related events.

Multithreading is useful for applications that perform a number of essentially independent tasks that do not need to be serialized. An example is a database server that listens for and processes numerous client requests. With multiple threads running within the same process, switching back and forth among threads involves less processor overhead than a major process switch between different processes. Threads are also useful for structuring processes that are part of the OS kernel as described in subsequent chapters.

Symmetric multiprocessing (SMP) is a term that refers to a computer hardware architecture (described in Chapter 1) and also to the OS behavior that exploits that architecture. The OS of an SMP schedules processes or threads across all of the processors. SMP has a number of potential advantages over uniprocessor architecture, including the following:

- **Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type. This is illustrated in Figure 2.12. With multiprogramming, only one process can execute at a time; meanwhile all other processes are waiting for the processor. With multiprocessing, more than one process can be running simultaneously, each on a different processor.
- **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the system. Instead, the system can continue to function at reduced performance.
- **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.
- **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

It is important to note that these are potential, rather than guaranteed, benefits. The OS must provide tools and functions to exploit the parallelism in an SMP system.

Multithreading and SMP are often discussed together, but the two are independent facilities. Even on a uniprocessor system, multithreading is useful for structuring applications and kernel processes. An SMP system is useful even for nonthreaded processes, because several processes can run in parallel. However, the two facilities complement each other and can be used effectively together.

An attractive feature of an SMP is that the existence of multiple processors is transparent to the user. The OS takes care of scheduling of threads or processes on

ISBN 1-256-52023-3

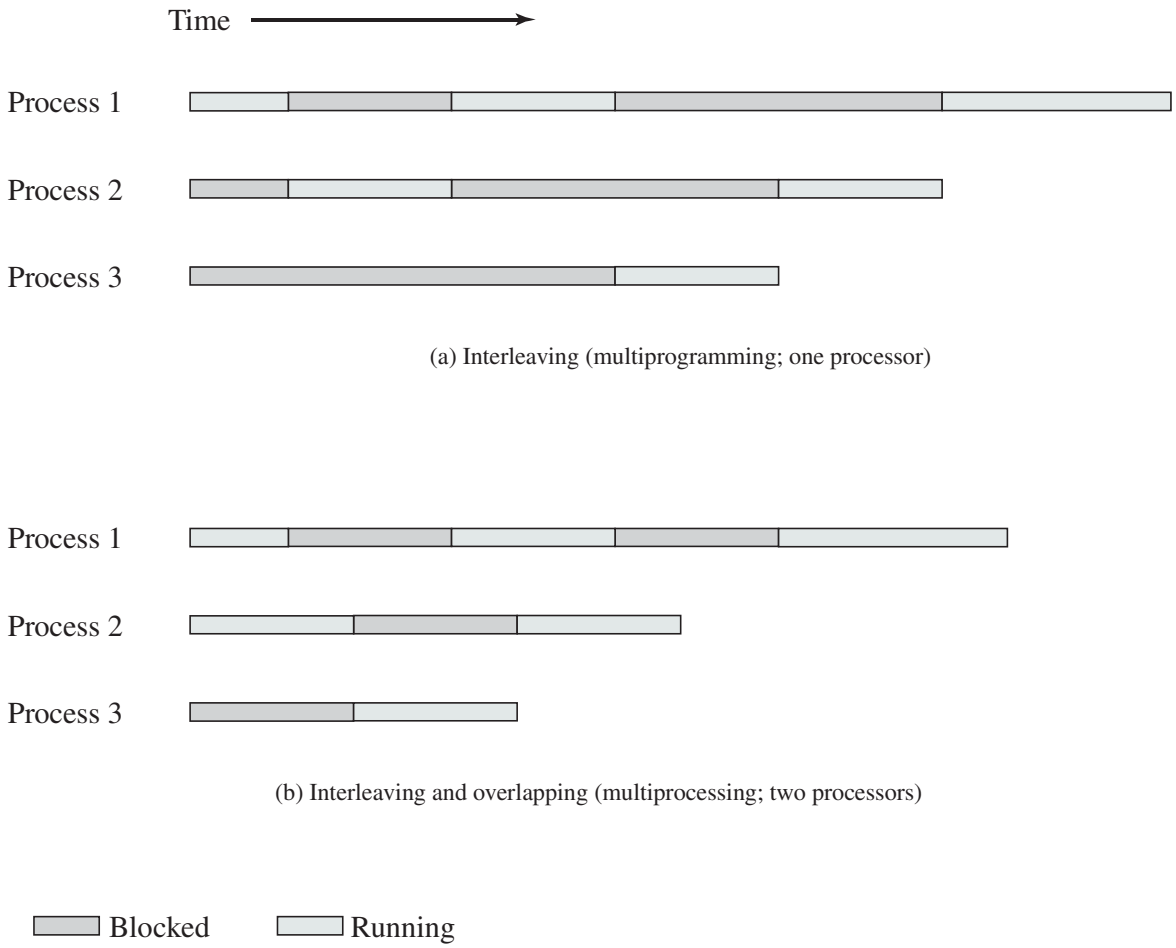


Figure 2.12 Multiprogramming and Multiprocessing

individual processors and of synchronization among processors. This book discusses the scheduling and synchronization mechanisms used to provide the single-system appearance to the user. A different problem is to provide the appearance of a single system for a cluster of separate computers—a multicomputer system. In this case, we are dealing with a collection of entities (computers), each with its own main memory, secondary memory, and other I/O modules. A **distributed operating system** provides the illusion of a single main memory space and a single secondary memory space, plus other unified access facilities, such as a distributed file system. Although clusters are becoming increasingly popular, and there are many cluster products on the market, the state of the art for distributed operating systems lags that of uniprocessor and SMP operating systems. We examine such systems in Part Eight.

Another innovation in OS design is the use of object-oriented technologies. **Object-oriented design** lends discipline to the process of adding modular extensions to a small kernel. At the OS level, an object-based structure enables programmers to customize an OS without disrupting system integrity. Object orientation also eases the development of distributed tools and full-blown distributed operating systems.

ISBN 1-256-52023-3

2.5 VIRTUAL MACHINES

Virtual Machines and Virtualizing

Traditionally, applications have run directly on an OS on a PC or a server. Each PC or server would run only one OS at a time. Thus, the vendor had to rewrite parts of its applications for each OS/platform they would run on. An effective strategy for dealing with this problem is known as **virtualization**. Virtualization technology enables a single PC or server to simultaneously run multiple operating systems or multiple sessions of a single OS. A machine with virtualization can host numerous applications, including those that run on different operating systems, on a single platform. In essence, the host operating system can support a number of **virtual machines (VM)**, each of which has the characteristics of a particular OS and, in some versions of virtualization, the characteristics of a particular hardware platform.

The VM approach is becoming a common way for businesses and individuals to deal with legacy applications and to optimize their hardware usage by maximizing the number of kinds of applications that a single computer can handle [GEER09]. Commercial VM offerings by companies such as VMware and Microsoft are widely used, with millions of copies having been sold. In addition to their use in server environments, these VM technologies also are used in desktop environments to run multiple operating systems, typically Windows and Linux.

The specific architecture of the VM approach varies among vendors. Figure 2.13 shows a typical arrangement. The **virtual machine monitor (VMM)**, or **hypervisor**, runs on top of (or is incorporated into) the host OS. The VMM supports VMs, which are emulated hardware devices. Each VM runs a separate OS. The VMM handles each operating system's communications with the processor, the storage medium, and the network. To execute programs, the VMM hands off the processor control to a virtual OS on a VM. Most VMs use virtualized network

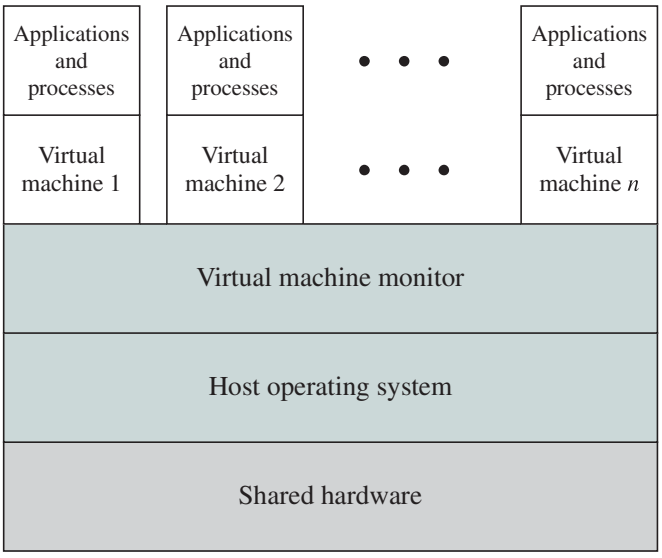


Figure 2.13 Virtual Memory Concept

ISBN 1-256-52023-3

connections to communicate with one another, when such communication is needed. Key to the success of this approach is that the VMM provides a layer between software environments and the underlying hardware and host OS that is programmable, transparent to the software above it, and makes efficient use of the hardware below it.

Virtual Machine Architecture[2]

Recall from Section 2.1 (see Figure 2.1) the discussion of the application programming interface, the application binary interface, and the instruction set architecture. Let us use these interface concepts to clarify the meaning of *machine* in the term *virtual machine.* Consider a process executing a compiled application program. From the perspective of the **process**, the machine on which it executes consists of the virtual memory space assigned to the process, the processor registers it may use, the user-level machine instructions it may execute, and the OS system calls it may invoke for I/O. Thus the **ABI** defines the machine as seen by a process.

From the perspective of an **application**, the machine characteristics are specified by high-level language capabilities, and OS and system library calls. Thus, the **API** defines the machine for an application.

For the **operating system**, the machine hardware defines the system that supports the operation of the OS and the numerous processes that execute concurrently. These processes share a file system and other I/O resources. The system allocates real memory and I/O resources to the processes and allows the processes to interact with their resources. From the OS perspective, therefore, it is the **ISA** that provides the interface between the system and machine.

With these considerations in mind, we can consider two architectural approaches to implementing virtual machines: process VMs and system VMs.

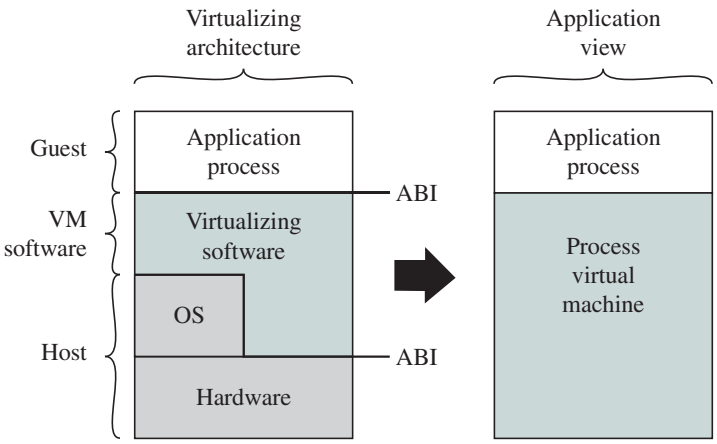
Process Virtual Machine In essence, a process VM presents an ABI to an application process, translates a set of OS and user-level instructions composing one platform to those of another (Figure 2.14a). A process VM is a virtual platform for executing a single process. As such, the process VM is created when the process is created and terminated when the process is terminated.

In order to provide cross-platform portability, a common implementation of the process VM architecture is as part of an overall HLL application environment. The resulting ABI does not correspond to any specific machine. Instead, the ABI specification is designed to easily support a given HLL or set of HLLs and to be easily portable to a variety of ISAs. The HLL VM includes a front-end compiler that generates a virtual binary code for execution or interpretation. This code can then be executed on any machine that has the process VM implemented.

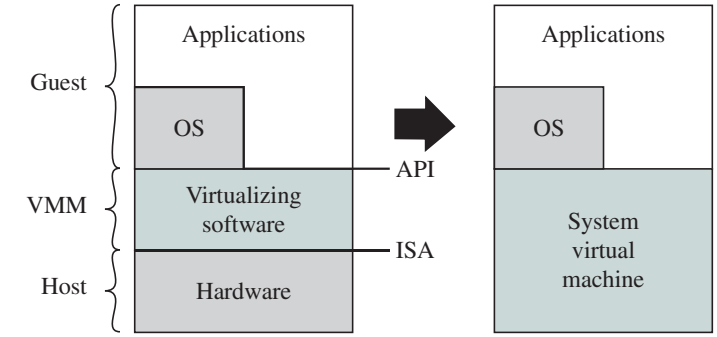
Two widely used examples of this approach are the Java VM architecture and the Microsoft Common Language Infrastructure, which is the foundation of the .NET framework.

ISBN 1-256-52023-3

[2]Much of the discussion that follows is based on [SMIT05].



(a) Process VM



(b) System VM

Figure 2.14 Process and System Virtual Machines

System Virtual Machine In a system VM, virtualizing software translates the ISA used by one hardware platform to that of another. Note in Figure 2.14a that the virtualizing software in the process VM approach makes use of the services of the host OS, while in the system VM approach there is logically no separate host OS, rather the host system OS incorporates the VM capability. In the system VM case, the virtualizing software is host to a number of guest operating systems, with each VM including its own OS. The VMM emulates the hardware ISA so that the guest software can potentially execute a different ISA from the one implemented on the host.

With the system VM approach, a single hardware platform can support multiple, isolated guest OS environments simultaneously. This approach provides a number of benefits, including application portability, support of legacy systems without the need to maintain legacy hardware, and security by means of isolation of each guest OS environment from the other guest environments.

A variant on the architecture shown in Figure 2.14b is referred to as a *hosted VM*. In this case, the VMM is built on top of an existing host OS. The VMM relies on the host OS to provide device drivers and other lower-level services. An example of a hosted VM is the VMware GSX server.

ISBN 1-256-52023-3

2.6 OS DESIGN CONSIDERATIONS FOR MULTIPROCESSOR AND MULTICORE

Symmetric Multiprocessor OS Considerations

In an SMP system, the kernel can execute on any processor, and typically each processor does self-scheduling from the pool of available processes or threads. The kernel can be constructed as multiple processes or multiple threads, allowing portions of the kernel to execute in parallel. The SMP approach complicates the OS. The OS designer must deal with the complexity due to sharing resources (like data structures) and coordinating actions (like accessing devices) from multiple parts of the OS executing at the same time. Techniques must be employed to resolve and synchronize claims to resources.

An SMP operating system manages processor and other computer resources so that the user may view the system in the same fashion as a multiprogramming uniprocessor system. A user may construct applications that use multiple processes or multiple threads within processes without regard to whether a single processor or multiple processors will be available. Thus, a multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors. The key design issues include the following:

- **Simultaneous concurrent processes or threads:** Kernel routines need to be reentrant to allow several processors to execute the same kernel code simultaneously. With multiple processors executing the same or different parts of the kernel, kernel tables and management structures must be managed properly to avoid data corruption or invalid operations.
- **Scheduling:** Any processor may perform scheduling, which complicates the task of enforcing a scheduling policy and assuring that corruption of the scheduler data structures is avoided. If kernel-level multithreading is used, then the opportunity exists to schedule multiple threads from the same process simultaneously on multiple processors. Multiprocessor scheduling is examined in Chapter 10.
- **Synchronization:** With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization. Synchronization is a facility that enforces mutual exclusion and event ordering. A common synchronization mechanism used in multiprocessor operating systems is locks, described in Chapter 5.
- **Memory management:** Memory management on a multiprocessor must deal with all of the issues found on uniprocessor computers and is discussed in Part Three. In addition, the OS needs to exploit the available hardware parallelism to achieve the best performance. The paging mechanisms on different processors must be coordinated to enforce consistency when several processors share a page or segment and to decide on page replacement. The reuse of physical pages is the biggest problem of concern; that is, it must be guaranteed that a physical page can no longer be accessed with its old contents before the page is put to a new use.

ISBN 1-256-52023-3

- **Reliability and fault tolerance:** The OS should provide graceful degradation in the face of processor failure. The scheduler and other portions of the OS must recognize the loss of a processor and restructure management tables accordingly.

Because multiprocessor OS design issues generally involve extensions to solutions to multiprogramming uniprocessor design problems, we do not treat multiprocessor operating systems separately. Rather, specific multiprocessor issues are addressed in the proper context throughout this book.

Multicore OS Considerations

The considerations for multicore systems include all the design issues discussed so far in this section for SMP systems. But additional concerns arise. The issue is one of the scale of the potential parallelism. Current multicore vendors offer systems with up to eight cores on a single chip. With each succeeding processor technology generation, the number of cores and the amount of shared and dedicated cache memory increases, so that we are now entering the era of "many-core" systems.

The design challenge for a many-core multicore system is to efficiently harness the multicore processing power and intelligently manage the substantial on-chip resources efficiently. A central concern is how to match the inherent parallelism of a many-core system with the performance requirements of applications. The potential for parallelism in fact exists at three levels in contemporary multicore system. First, there is hardware parallelism within each core processor, known as instruction level parallelism, which may or may not be exploited by application programmers and compilers. Second, there is the potential for multiprogramming and multithreaded execution within each processor. Finally, there is the potential for a single application to execute in concurrent processes or threads across multiple cores. Without strong and effective OS support for the last two types of parallelism just mentioned, hardware resources will not be efficiently used.

In essence, then, since the advent of multicore technology, OS designers have been struggling with the problem of how best to extract parallelism from computing workloads. A variety of approaches are being explored for next-generation operating systems. We introduce two general strategies in this section and consider some details in later chapters.

PARALLELISM WITHIN APPLICATIONS Most applications can, in principle, be subdivided into multiple tasks that can execute in parallel, with these tasks then being implemented as multiple processes, perhaps each with multiple threads. The difficulty is that the developer must decide how to split up the application work into independently executable tasks. That is, the developer must decide what pieces can or should be executed asynchronously or in parallel. It is primarily the compiler and the programming language features that support the parallel programming design process. But, the OS can support this design process, at minimum, by efficiently allocating resources among parallel tasks as defined by the developer.

Perhaps the most effective initiative to support developers is implemented in the latest release of the UNIX-based Mac OS X operating system. Mac OS X 10.6 includes a multicore support capability known as Grand Central Dispatch (GCD).

ISBN 1-256-52023-3

GCD does not help the developer decide how to break up a task or application into separate concurrent parts. But once a developer has identified something that can be split off into a separate task, GCD makes it as easy and noninvasive as possible to actually do so.

In essence, GCD is a thread pool mechanism, in which the OS maps tasks onto threads representing an available degree of concurrency (plus threads for blocking on I/O). Windows also has a thread pool mechanism (since 2000), and thread pools have been heavily used in server applications for years. What is new in GCD is the extension to programming languages to allow anonymous functions (called blocks) as a way of specifying tasks. GCD is hence not a major evolutionary step. Nevertheless, it is a new and valuable tool for exploiting the available parallelism of a multicore system.

One of Apple's slogans for GCD is "islands of serialization in a sea of concurrency." That captures the practical reality of adding more concurrency to run-of-the-mill desktop applications. Those islands are what isolate developers from the thorny problems of simultaneous data access, deadlock, and other pitfalls of multithreading. Developers are encouraged to identify functions of their applications that would be better executed off the main thread, even if they are made up of several sequential or otherwise partially interdependent tasks. GCD makes it easy to break off the entire unit of work while maintaining the existing order and dependencies between subtasks. In later chapters, we look at some of the details of GCD.

Virtual Machine Approach An alternative approach is to recognize that with the ever-increasing number of cores on a chip, the attempt to multiprogram individual cores to support multiple applications may be a misplaced use of resources [JACK10]. If instead, we allow one or more cores to be dedicated to a particular process and then leave the processor alone to devote its efforts to that process, we avoid much of the overhead of task switching and scheduling decisions. The multicore OS could then act as a hypervisor that makes a high-level decision to allocate cores to applications but does little in the way of resource allocation beyond that.

The reasoning behind this approach is as follows. In the early days of computing, one program was run on a single processor. With multiprogramming, each application is given the illusion that it is running on a dedicated processor. Multiprogramming is based on the concept of a process, which is an abstraction of an execution environment. To manage processes, the OS requires protected space, free from user and program interference. For this purpose, the distinction between kernel mode and user mode was developed. In effect, kernel mode and user mode abstracted the processor into two processors. With all these virtual processors, however, come struggles over who gets the attention of the real processor. The overhead of switching between all these processors starts to grow to the point where responsiveness suffers, especially when multiple cores are introduced. But with many-core systems, we can consider dropping the distinction between kernel and user mode. In this approach, the OS acts more like a hypervisor. The programs themselves take on many of the duties of resource management. The OS assigns an application a processor and some memory, and the program itself, using metadata generated by the compiler, would best know how to use these resources.

ISBN 1-256-52023-3

2.7 MICROSOFT WINDOWS OVERVIEW

History

The story of Windows begins with a very different OS, developed by Microsoft for the first IBM personal computer and referred to as MS-DOS. The initial version, MS-DOS 1.0, was released in August 1981. It consisted of 4000 lines of assembly language source code and ran in 8 Kbytes of memory using the Intel 8086 microprocessor.

The IBM PC was an important stage in a continuing revolution in computing that has expanded computing from the data center of the 1960s, to the departmental minicomputer of the 1970s, and to the desktop in the 1980s. The revolution has continued with computing moving into the briefcase in the 1990s, and into our pockets during the most recent decade.

Microsoft's initial OS ran a single application at a time, using a command line interface to control the system. It took a long time for Microsoft to develop a true GUI interface for the PC; on their third try they succeeded. The 16-bit Windows 3.0 shipped in 1990 and instantly became successful, selling a million copies in six months. Windows 3.0 was implemented as a layer on top of MS-DOS and suffered from the limitations of that primitive system. Five years later, Microsoft shipped a 32-bit version, Windows 95, which was also very successful and led to the development of additional versions: Windows 98 and Windows Me.

Meanwhile, it had become clear to Microsoft that the MS-DOS platform could not sustain a truly modern OS. In 1989 Microsoft hired Dave Cutler, who had developed the very successful RSX-11M and VAX/VMS operating systems at Digital Equipment Corporation. Cutler's charter was to develop a modern OS, which was portable to architectures other than the Intel x86 family, and yet compatible with the OS/2 system that Microsoft was jointly developing with IBM, as well as the portable UNIX standard, POSIX. This system was christened NT (New Technology).

The first version of Windows NT (3.1) was released in 1993, with the same GUI as Windows 3.1, the follow-on to Windows 3.0. However, NT 3.1 was a new 32-bit OS with the ability to support older DOS and Windows applications as well as provide OS/2 support. Several versions of NT 3.x followed with support for additional hardware platforms. In 1996, Microsoft released NT 4.0 with the same user interface as Windows 95. In 2000, Microsoft introduced the next major upgrade of the NT OS: Windows 2000. The underlying Executive and Kernel architecture is fundamentally the same as in NT 3.1, but new features have been added. The emphasis in Windows 2000 was the addition of services and functions to support distributed processing. The central element of Windows 2000's new features was Active Directory, which is a distributed directory service able to map names of arbitrary objects to any kind of information about those objects. Windows 2000 also added the plug-and-play and power-management facilities that were already in Windows 98, the successor to Windows 95. These features are particularly important for laptop computers.

In 2001, a new desktop version of NT was released, known as Windows XP. The goal of Windows XP was to finally replace the versions of Windows based on MS-DOS with an OS based on NT. In 2007, Microsoft shipped Windows Vista for the desktop and a short time later, Windows Server 2008. In 2009, they shipped

ISBN 1-256-52023-3

Windows 7 and Windows Server 2008 R2. Despite the difference in naming, the client and server versions of these systems use many of the same files, but with additional features and capabilities enabled for servers.

Over the years, NT has attempted to support multiple processor architectures; the Intel i860 was the original target for NT as well as the x86. Subsequently, NT added support for the Digital Alpha architecture, the PowerPC, and the MIPS. Later came the Intel IA64 (Itanium) and the 64-bit version of the x86, based on the AMD64 processor architecture. Windows 7 supports only x86 and AMD64. Windows Server 2008 R2 supports only AMD64 and IA64—but Microsoft has announced that it will end support for IA64 in future releases. All the other processor architectures have failed in the market, and today only the x86, AMD64, and ARM architectures are viable. Microsoft's support for ARM is limited to their Windows CE OS, which runs on phones and handheld devices. Windows CE has little relationship to the NT-based Windows that runs on slates, netbooks/laptops, desktops, and servers.

Microsoft has announced that it is developing a version of NT that targets cloud computing: Windows Azure. Azure includes a number of features that are specific to the requirements of public and private clouds. Though it is closely related to Windows Server, it does not share files in the same way that the Windows client and server versions do.

The Modern OS

Modern operating systems, such as today's Windows and UNIX (with all its flavors like Solaris, Linux, and MacOS X), must exploit the capabilities of all the billions of transistors on each silicon chip. They must work with multiple 32-bit and 64-bit CPUs, with adjunct GPUs, DSPs, and fixed function units. They must provide support for sophisticated input/output (multiple touch-sensitive displays, cameras, microphones, biometric and other sensors) and handle a variety of data challenges (streaming media, photos, scientific number crunching, search queries)—all while giving a human being a responsive, real-time experience with the computing system.

To handle these requirements, the computer cannot be doing only one thing at a time. Unlike the early days of the PC, when the OS ran a single application at a time, hundreds of activities are taking place to provide the modern computing experience. The OS can no longer just switch to the application and step away until it is needed; it must aggressively manage the system and coordinate between all the competing computations that are taking place often simultaneously on the multiple CPUs, GPUs, and DSPs that may be present in a modern computing environment. Thus all modern operating systems have multitasking capability, even though they may be acting on behalf of only a single human being (called the user).

Windows is a sophisticated multitasking OS, designed to manage the complexity of the modern computing environment, provide a rich platform for application developers, and support a rich set of experiences for users. Like Solaris, Windows is designed to have the features that enterprises need, while at the same time Windows, like MacOS, provides the simplicity and ease-of-use that consumers require. In the following sections we will present an overview of the fundamental structure and capabilities of Windows.

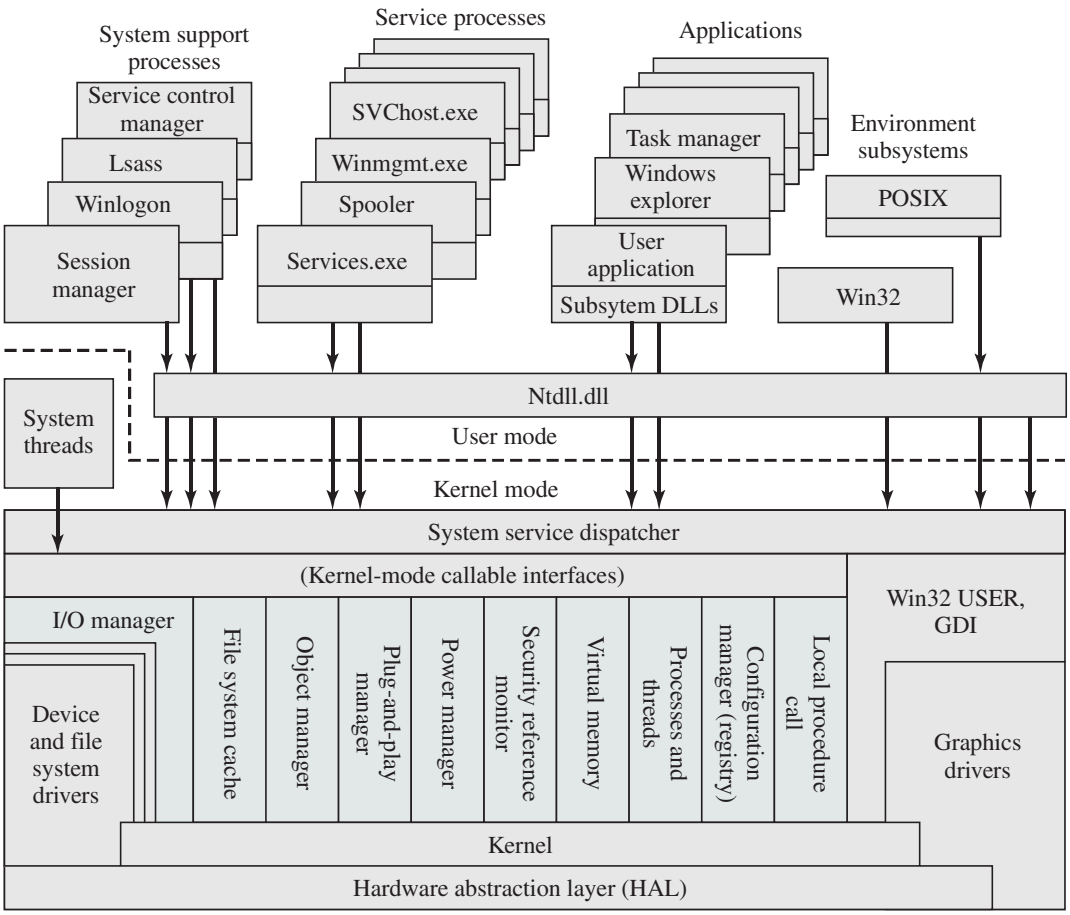
ISBN 1-256-52023-3

Architecture

Figure 2.15 illustrates the overall structure of Windows 7; all releases of Windows based on NT have essentially the same structure at this level of detail.

As with virtually all operating systems, Windows separates application-oriented software from the core OS software. The latter, which includes the Executive, the Kernel, device drivers, and the hardware abstraction layer, runs in kernel mode. Kernel mode software has access to system data and to the hardware. The remaining software, running in user mode, has limited access to system data.

Operating System Organization Windows has a highly modular architecture. Each system function is managed by just one component of the OS. The rest of the OS and all applications access that function through the responsible component using standard interfaces. Key system data can only be accessed through the appropriate



Lsass = local security authentication server
POSIX = portable operating system interface
GDI = graphics device interface
DLL = dynamic link libraries

Colored area indicates Executive

Figure 2.15 Windows and Windows Vista Architecture [RUSS11]

ISBN 1-256-52023-3

function. In principle, any module can be removed, upgraded, or replaced without rewriting the entire system or its standard application program interfaces (APIs).

The kernel-mode components of Windows are the following:

- **Executive:** Contains the core OS services, such as memory management, process and thread management, security, I/O, and interprocess communication.
- **Kernel:** Controls execution of the processors. The Kernel manages thread scheduling, process switching, exception and interrupt handling, and multiprocessor synchronization. Unlike the rest of the Executive and the user level, the Kernel's own code does not run in threads.
- **Hardware abstraction layer (HAL):** Maps between generic hardware commands and responses and those unique to a specific platform. It isolates the OS from platform-specific hardware differences. The HAL makes each computer's system bus, direct memory access (DMA) controller, interrupt controller, system timers, and memory controller look the same to the Executive and Kernel components. It also delivers the support needed for SMP, explained subsequently.
- **Device drivers:** Dynamic libraries that extend the functionality of the Executive. These include hardware device drivers that translate user I/O function calls into specific hardware device I/O requests and software components for implementing file systems, network protocols, and any other system extensions that need to run in kernel mode.
- **Windowing and graphics system:** Implements the GUI functions, such as dealing with windows, user interface controls, and drawing.

The Windows Executive includes components for specific system functions and provides an API for user-mode software. Following is a brief description of each of the Executive modules:

- **I/O manager:** Provides a framework through which I/O devices are accessible to applications, and is responsible for dispatching to the appropriate device drivers for further processing. The I/O manager implements all the Windows I/O APIs and enforces security and naming for devices, network protocols, and file systems (using the object manager). Windows I/O is discussed in Chapter 11.
- **Cache manager:** Improves the performance of file-based I/O by causing recently referenced file data to reside in main memory for quick access, and deferring disk writes by holding the updates in memory for a short time before sending them to the disk in more efficient batches.
- **Object manager:** Creates, manages, and deletes Windows Executive objects that are used to represent resources such as processes, threads, and synchronization objects. It enforces uniform rules for retaining, naming, and setting the security of objects. The object manager also creates the entries in each processes' handle table, which consist of access control information and a pointer to the object. Windows objects are discussed later in this section.
- **Plug-and-play manager:** Determines which drivers are required to support a particular device and loads those drivers.

ISBN 1-256-52023-3

- **Power manager:** Coordinates power management among various devices and can be configured to reduce power consumption by shutting down idle devices, putting the processor to sleep, and even writing all of memory to disk and shutting off power to the entire system.
- **Security reference monitor:** Enforces access-validation and audit-generation rules. The Windows object-oriented model allows for a consistent and uniform view of security, right down to the fundamental entities that make up the Executive. Thus, Windows uses the same routines for access validation and for audit checks for all protected objects, including files, processes, address spaces, and I/O devices. Windows security is discussed in Chapter 15.
- **Virtual memory manager:** Manages virtual addresses, physical memory, and the paging files on disk. Controls the memory management hardware and data structures which map virtual addresses in the process's address space to physical pages in the computer's memory. Windows virtual memory management is described in Chapter 8.
- **Process/thread manager:** Creates, manages, and deletes process and thread objects. Windows process and thread management are described in Chapter 4.
- **Configuration manager:** Responsible for implementing and managing the system registry, which is the repository for both system-wide and per-user settings of various parameters.
- **Advanced local procedure call (ALPC) facility:** Implements an efficient cross-process procedure call mechanism for communication between local processes implementing services and subsystems. Similar to the remote procedure call (RPC) facility used for distributed processing.

User-Mode Processes Four basic types of user-mode processes are supported by Windows:

- **Special system processes:** User-mode services needed to manage the system, such as the session manager, the authentication subsystem, the service manager, and the logon process.
- **Service processes:** The printer spooler, the event logger, user-mode components that cooperate with device drivers, various network services, and many, many others. Services are used by both Microsoft and external software developers to extend system functionality as they are the only way to run background user-mode activity on a Windows system.
- **Environment subsystems:** Provide different OS personalities (environments). The supported subsystems are Win32 and POSIX. Each environment subsystem includes a subsystem process shared among all applications using the subsystem and dynamic link libraries (DLLs) that convert the user application calls to ALPC calls on the subsystem process, and/or native Windows calls.
- **User applications:** Executables (EXEs) and DLLs that provide the functionality users run to make use of the system. EXEs and DLLs are generally targeted at a specific environment subsystem; although some of the programs that are provided as part of the OS use the native system interfaces (NT API). There is also support for running 32-bit programs on 64-bit systems.

ISBN 1-256-52023-3

Windows is structured to support applications written for multiple OS personalities. Windows provides this support using a common set of kernel mode components that underlie the OS environment subsystems. The implementation of each environment subsystem includes a separate process, which contains the shared data structures, privileges, and Executive object handles needed to implement a particular personality. The process is started by the Windows Session Manager when the first application of that type is started. The subsystem process runs as a system user, so the Executive will protect its address space from processes run by ordinary users.

An environment subsystem provides a graphical or command-line user interface that defines the look and feel of the OS for a user. In addition, each subsystem provides the API for that particular environment. This means that applications created for a particular operating environment need only be recompiled to run on Windows. Because the OS interface that they see is the same as that for which they were written, the source code does not need to be modified.

Client/Server Model

The Windows OS services, the environment subsystems, and the applications are structured using the client/server computing model, which is a common model for distributed computing and which is discussed in Part Six. This same architecture can be adopted for use internally to a single system, as is the case with Windows.

The native NT API is a set of kernel-based services which provide the core abstractions used by the system, such as processes, threads, virtual memory, I/O, and communication. Windows provides a far richer set of services by using the client/server model to implement functionality in user-mode processes. Both the environment subsystems and the Windows user-mode services are implemented as processes that communicate with clients via RPC. Each server process waits for a request from a client for one of its services (e.g., memory services, process creation services, or networking services). A client, which can be an application program or another server program, requests a service by sending a message. The message is routed through the Executive to the appropriate server. The server performs the requested operation and returns the results or status information by means of another message, which is routed through the Executive back to the client.

Advantages of a client/server architecture include the following:

- It simplifies the Executive. It is possible to construct a variety of APIs implemented in user-mode servers without any conflicts or duplications in the Executive. New APIs can be added easily.
- It improves reliability. Each new server runs outside of the kernel, with its own partition of memory, protected from other servers. A single server can fail without crashing or corrupting the rest of the OS.
- It provides a uniform means for applications to communicate with services via RPCs without restricting flexibility. The message-passing process is hidden from the client applications by function stubs, which are small pieces of code which wrap the RPC call. When an application makes an API call to an environment subsystem or a service, the stub in the client application packages the parameters for the call and sends them as a message to the server process that implements the call.

ISBN 1-256-52023-3

- It provides a suitable base for distributed computing. Typically, distributed computing makes use of a client/server model, with remote procedure calls implemented using distributed client and server modules and the exchange of messages between clients and servers. With Windows, a local server can pass a message on to a remote server for processing on behalf of local client applications. Clients need not know whether a request is being serviced locally or remotely. Indeed, whether a request is serviced locally or remotely can change dynamically based on current load conditions and on dynamic configuration changes.

Threads and SMP

Two important characteristics of Windows are its support for threads and for symmetric multiprocessing (SMP), both of which were introduced in Section 2.4. [RUSS11] lists the following features of Windows that support threads and SMP:

- OS routines can run on any available processor, and different routines can execute simultaneously on different processors.
- Windows supports the use of multiple threads of execution within a single process. Multiple threads within the same process may execute on different processors simultaneously.
- Server processes may use multiple threads to process requests from more than one client simultaneously.
- Windows provides mechanisms for sharing data and resources between processes and flexible interprocess communication capabilities.

Windows Objects

Though the core of Windows is written in C, the design principles followed draw heavily on the concepts of object-oriented design. This approach facilitates the sharing of resources and data among processes and the protection of resources from unauthorized access. Among the key object-oriented concepts used by Windows are the following:

- **Encapsulation:** An object consists of one or more items of data, called attributes, and one or more procedures that may be performed on those data, called services. The only way to access the data in an object is by invoking one of the object's services. Thus, the data in the object can easily be protected from unauthorized use and from incorrect use (e.g., trying to execute a nonexecutable piece of data).
- **Object class and instance:** An object class is a template that lists the attributes and services of an object and defines certain object characteristics. The OS can create specific instances of an object class as needed. For example, there is a single process object class and one process object for every currently active process. This approach simplifies object creation and management.
- **Inheritance:** Although the implementation is hand coded, the Executive uses inheritance to extend object classes by adding new features. Every Executive

ISBN 1-256-52023-3

class is based on a base class which specifies virtual methods that support creating, naming, securing, and deleting objects. Dispatcher objects are Executive objects that inherit the properties of an event object, so they can use common synchronization methods. Other specific object types, such as the device class, allow classes for specific devices to inherit from the base class, and add additional data and methods.

- **Polymorphism:** Internally, Windows uses a common set of API functions to manipulate objects of any type; this is a feature of polymorphism, as defined in Appendix D. However, Windows is not completely polymorphic because there are many APIs that are specific to a single object type.

The reader unfamiliar with object-oriented concepts should review Appendix D.

Not all entities in Windows are objects. Objects are used in cases where data are intended for user mode access or when data access is shared or restricted. Among the entities represented by objects are files, processes, threads, semaphores, timers, and graphical windows. Windows creates and manages all types of objects in a uniform way, via the object manager. The object manager is responsible for creating and destroying objects on behalf of applications and for granting access to an object's services and data.

Each object within the Executive, sometimes referred to as a kernel object (to distinguish from user-level objects not of concern to the Executive), exists as a memory block allocated by the kernel and is directly accessible only by kernel mode components. Some elements of the data structure (e.g., object name, security parameters, usage count) are common to all object types, while other elements are specific to a particular object type (e.g., a thread object's priority). Because these object data structures are in the part of each process's address space accessible only by the kernel, it is impossible for an application to reference these data structures and read or write them directly. Instead, applications manipulate objects indirectly through the set of object manipulation functions supported by the Executive. When an object is created, the application that requested the creation receives back a handle for the object. In essence, a handle is an index into a per-process Executive table containing a pointer to the referenced object. This handle can then be used by any thread within the same process to invoke Win32 functions that work with objects, or can be duplicated into other processes.

Objects may have security information associated with them, in the form of a Security Descriptor (SD). This security information can be used to restrict access to the object based on contents of a token object which describes a particular user. For example, a process may create a named semaphore object with the intent that only certain users should be able to open and use that semaphore. The SD for the semaphore object can list those users that are allowed (or denied) access to the semaphore object along with the sort of access permitted (read, write, change, etc.).

In Windows, objects may be either named or unnamed. When a process creates an unnamed object, the object manager returns a handle to that object, and the handle is the only way to refer to it. Handles can be inherited by child processes, or duplicated between processes. Named objects are also given a name that other unrelated processes can use to obtain a handle to the object. For example, if proc-

ISBN 1-256-52023-3

Table 2.4 Windows Kernel Control Objects

Asynchronous Procedure Call	Used to break into the execution of a specified thread and to cause a procedure to be called in a specified processor mode.
Deferred Procedure Call	Used to postpone interrupt processing to avoid delaying hardware interrupts. Also used to implement timers and interprocessor communication.
Interrupt	Used to connect an interrupt source to an interrupt service routine by means of an entry in an Interrupt Dispatch Table (IDT). Each processor has an IDT that is used to dispatch interrupts that occur on that processor.
Process	Represents the virtual address space and control information necessary for the execution of a set of thread objects. A process contains a pointer to an address map, a list of ready threads containing thread objects, a list of threads belonging to the process, the total accumulated time for all threads executing within the process, and a base priority.
Thread	Represents thread objects, including scheduling priority and quantum, and which processors the thread may run on.
Profile	Used to measure the distribution of run time within a block of code. Both user and system code can be profiled.

ess A wishes to synchronize with process B, it could create a named event object and pass the name of the event to B. Process B could then open and use that event object. However, if A simply wished to use the event to synchronize two threads within itself, it would create an unnamed event object, because there is no need for other processes to be able to use that event.

There are two categories of objects used by Windows for synchronizing the use of the processor:

- **Dispatcher objects:** The subset of Executive objects which threads can wait on to control the dispatching and synchronization of thread-based system operations. These are described in Chapter 6.
- **Control objects:** Used by the Kernel component to manage the operation of the processor in areas not managed by normal thread scheduling. Table 2.4 lists the Kernel control objects.

Windows is not a full-blown object-oriented OS. It is not implemented in an object-oriented language. Data structures that reside completely within one Executive component are not represented as objects. Nevertheless, Windows illustrates the power of object-oriented technology and represents the increasing trend toward the use of this technology in OS design.

What Is New in Windows 7

The core architecture of Windows has been very stable; however, at each release there are new features and improvements made even at the lower levels of the system. Many of the changes in Windows are not visible in the features themselves, but in the performance and stability of the system. These are due to changes in the engineering behind Windows. Other improvements are due to new features, or improvements to existing features:

ISBN 1-256-52023-3

- **Engineering improvements:** The performance of hundreds of key scenarios, such as opening a file from the GUI, are tracked and continuously characterized to identify and fix problems. The system is now built in layers which can be separately tested, improving modularity and reducing complexity.
- **Performance improvements:** The amount of memory required has been reduced, both for clients and servers. The VMM is more aggressive about limiting the memory use of runaway processes (see Section 8.5). Background processes can arrange to start upon an event trigger, such as a plugging in a camera, rather than running continuously.
- **Reliability improvements:** The user-mode heap is more tolerant of memory allocation errors by C/C++ programmers, such as continuing to use memory after it is freed. Programs that make such errors are detected and the heap allocation policies are modified for that program to defer freeing memory and avoid corruption of the program's data.
- **Energy efficiency:** Many improvements have been made to the energy efficiency of Windows. On servers, unused processors can be "parked," reducing their energy use. All Windows systems are more efficient in how the timers work; avoiding timer interrupts and the associated background activity allows the processors to remain idle longer, which allows modern processors to consume less energy. Windows accomplishes this by coalescing timer interrupts into batches.
- **Security:** Windows 7 builds on the security features in Windows Vista, which added integrity levels to the security model, provided BitLocker volume encryption (see Section 15.6), and limited privileged actions by ordinary users. BitLocker is now easier to set up and use, and privileged actions result in many fewer annoying GUI pop-ups.
- **Thread improvements:** The most interesting Windows 7 changes were in the Kernel. The number of logical CPUs available on each system is growing dramatically. Previous versions of Windows limited the number of CPUs to 64, because of the bitmasks used to represent values like processor affinity (see Section 4.4). Windows 7 can support hundreds of CPUs. To ensure that the performance of the system scaled with the number of CPUs, major improvements were made to the Kernel-scheduling code to break apart locks and reduce contention. As the number of available CPUs increase, new programming environments are being developed to support the finer-grain parallelism than is available with threads. Windows 7 supports a form of User-Mode Scheduling which separates the user-mode and kernel-mode portions of threads, allowing the user-mode portions to yield the CPU without entering the Kernel scheduler. Finally, Windows Server 2008 R2 introduced Dynamic Fair Share Scheduling (DFSS) to allow multiuser servers to limit how much one user can interfere with another. DFSS keeps a user with 20 running threads from getting twice as much processor time as a user with only 10 running threads.

ISBN 1-256-52023-3

2.8 TRADITIONAL UNIX SYSTEMS

History

The history of UNIX is an oft-told tale and will not be repeated in great detail here. Instead, we provide a brief summary.

UNIX was initially developed at Bell Labs and became operational on a PDP-7 in 1970. Some of the people involved at Bell Labs had also participated in the time-sharing work being done at MIT's Project MAC. That project led to the development of first CTSS and then Multics. Although it is common to say that the original UNIX was a scaled-down version of Multics, the developers of UNIX actually claimed to be more influenced by CTSS [RITC78]. Nevertheless, UNIX incorporated many ideas from Multics.

Work on UNIX at Bell Labs, and later elsewhere, produced a series of versions of UNIX. The first notable milestone was porting the UNIX system from the PDP-7 to the PDP-11. This was the first hint that UNIX would be an OS for all computers. The next important milestone was the rewriting of UNIX in the programming language C. This was an unheard-of strategy at the time. It was generally felt that something as complex as an OS, which must deal with time-critical events, had to be written exclusively in assembly language. Reasons for this attitude include the following:

- Memory (both RAM and secondary store) was small and expensive by today's standards, so effective use was important. This included various techniques for overlaying memory with different code and data segments, and self-modifying code.
- Even though compilers had been available since the 1950s, the computer industry was generally skeptical of the quality of automatically generated code. With resource capacity small, efficient code, both in terms of time and space, was essential.
- Processor and bus speeds were relatively slow, so saving clock cycles could make a substantial difference in execution time.

The C implementation demonstrated the advantages of using a high-level language for most if not all of the system code. Today, virtually all UNIX implementations are written in C.

These early versions of UNIX were popular within Bell Labs. In 1974, the UNIX system was described in a technical journal for the first time [RITC74]. This spurred great interest in the system. Licenses for UNIX were provided to commercial institutions as well as universities. The first widely available version outside Bell Labs was Version 6, in 1976. The follow-on Version 7, released in 1978, is the ancestor of most modern UNIX systems. The most important of the non-AT&T systems to be developed was done at the University of California at Berkeley, called UNIX BSD (Berkeley Software Distribution), running first on PDP and then VAX computers. AT&T continued to develop and refine the system. By 1982, Bell Labs had combined several AT&T variants of UNIX into a single system, marketed commercially as UNIX System III. A number of features was later added to the OS to produce UNIX System V.

ISBN 1-256-52023-3

Description

Figure 2.16 provides a general description of the classic UNIX architecture. The underlying hardware is surrounded by the OS software. The OS is often called the system kernel, or simply the kernel, to emphasize its isolation from the user and applications. It is the UNIX kernel that we will be concerned with in our use of UNIX as an example in this book. UNIX also comes equipped with a number of user services and interfaces that are considered part of the system. These can be grouped into the shell, other interface software, and the components of the C compiler (compiler, assembler, loader). The layer outside of this consists of user applications and the user interface to the C compiler.

A closer look at the kernel is provided in Figure 2.17. User programs can invoke OS services either directly or through library programs. The system call interface is the boundary with the user and allows higher-level software to gain access to specific kernel functions. At the other end, the OS contains primitive routines that interact directly with the hardware. Between these two interfaces, the system is divided into two main parts, one concerned with process control and the other concerned with file management and I/O. The process control subsystem is responsible for memory management, the scheduling and dispatching of processes, and the synchronization and interprocess communication of processes. The file system exchanges data between memory and external devices either as a stream of characters or in blocks. To achieve this, a variety of device drivers are used. For block-oriented transfers, a disk cache approach is used: A system buffer in main memory is interposed between the user address space and the external device.

The description in this subsection has dealt with what might be termed *traditional UNIX systems*; [VAHA96] uses this term to refer to System V Release 3 (SVR3), 4.3BSD, and earlier versions. The following general statements may be

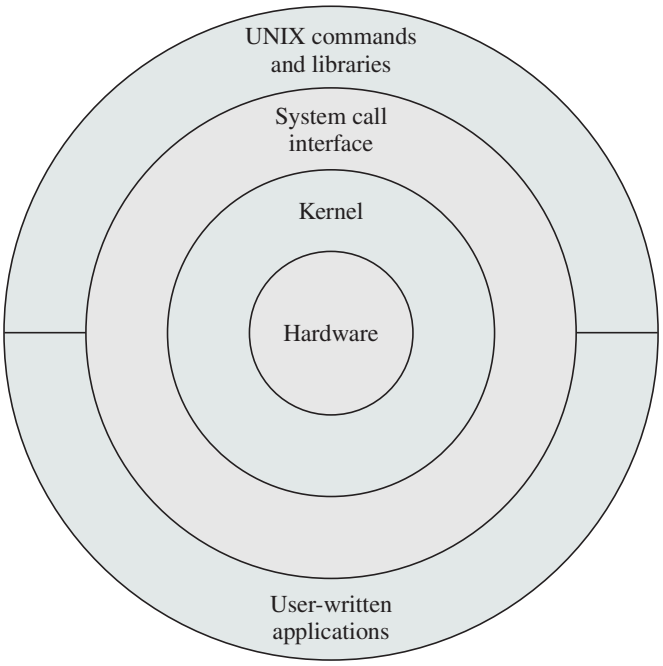


Figure 2.16 General UNIX Architecture

ISBN 1-256-52023-3

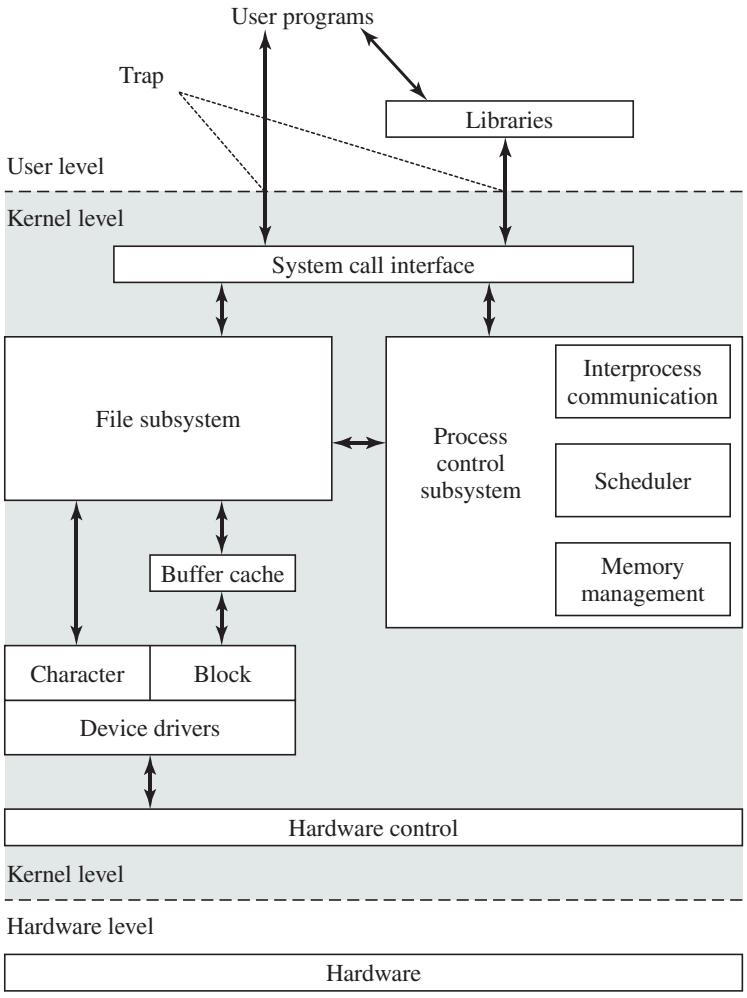


Figure 2.17 Traditional UNIX Kernel

made about a traditional UNIX system. It is designed to run on a single processor and lacks the ability to protect its data structures from concurrent access by multiple processors. Its kernel is not very versatile, supporting a single type of file system, process scheduling policy, and executable file format. The traditional UNIX kernel is not designed to be extensible and has few facilities for code reuse. The result is that, as new features were added to the various UNIX versions, much new code had to be added, yielding a bloated and unmodular kernel.

2.9 MODERN UNIX SYSTEMS

As UNIX evolved, the number of different implementations proliferated, each providing some useful features. There was a need to produce a new implementation that unified many of the important innovations, added other modern OS design features, and produced a more modular architecture. Typical of the modern UNIX kernel is the architecture depicted in Figure 2.18. There is a small core of facilities, written in

ISBN 1-256-52023-3

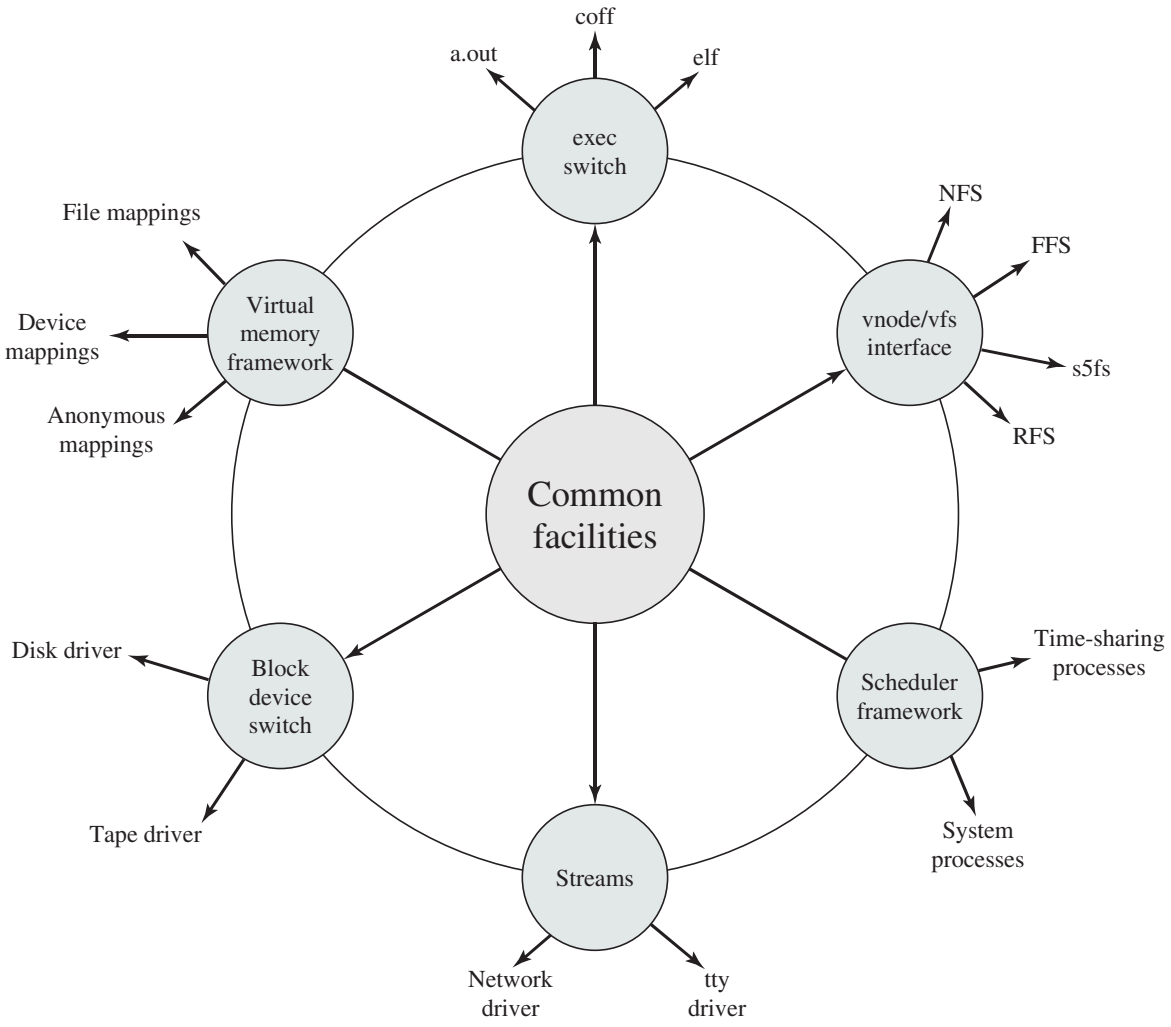


Figure 2.18 Modern UNIX Kernel

a modular fashion, that provide functions and services needed by a number of OS processes. Each of the outer circles represents functions and an interface that may be implemented in a variety of ways.

We now turn to some examples of modern UNIX systems.

System V Release 4 (SVR4)

SVR4, developed jointly by AT&T and Sun Microsystems, combines features from SVR3, 4.3BSD, Microsoft Xenix System V, and SunOS. It was almost a total rewrite of the System V kernel and produced a clean, if complex, implementation. New features in the release include real-time processing support, process scheduling classes, dynamically allocated data structures, virtual memory management, virtual file system, and a preemptive kernel.

SVR4 draws on the efforts of both commercial and academic designers and was developed to provide a uniform platform for commercial UNIX deployment. It has succeeded in this objective and is perhaps the most important UNIX variant. It incorporates most of the important features ever developed on any UNIX system

ISBN 1-256-52023-3

and does so in an integrated, commercially viable fashion. SVR4 runs on processors ranging from 32-bit microprocessors up to supercomputers.

BSD

The Berkeley Software Distribution (BSD) series of UNIX releases have played a key role in the development of OS design theory. 4.xBSD is widely used in academic installations and has served as the basis of a number of commercial UNIX products. It is probably safe to say that BSD is responsible for much of the popularity of UNIX and that most enhancements to UNIX first appeared in BSD versions.

4.4BSD was the final version of BSD to be released by Berkeley, with the design and implementation organization subsequently dissolved. It is a major upgrade to 4.3BSD and includes a new virtual memory system, changes in the kernel structure, and a long list of other feature enhancements.

One of the most widely used and best documented versions of BSD is FreeBSD. FreeBSD is popular for Internet-based servers and firewalls and is used in a number of embedded systems.

The latest version of the Macintosh OS, Mac OS X, is based on FreeBSD 5.0 and the Mach 3.0 microkernel.

Solaris 10

Solaris is Sun's SVR4-based UNIX release, with the latest version being 10. Solaris provides all of the features of SVR4 plus a number of more advanced features, such as a fully preemptable, multithreaded kernel, full support for SMP, and an object-oriented interface to file systems. Solaris is the most widely used and most successful commercial UNIX implementation.

2.10 LINUX

History

Linux started out as a UNIX variant for the IBM PC (Intel 80386) architecture. Linus Torvalds, a Finnish student of computer science, wrote the initial version. Torvalds posted an early version of Linux on the Internet in 1991. Since then, a number of people, collaborating over the Internet, have contributed to the development of Linux, all under the control of Torvalds. Because Linux is free and the source code is available, it became an early alternative to other UNIX workstations, such as those offered by Sun Microsystems and IBM. Today, Linux is a full-featured UNIX system that runs on all of these platforms and more, including Intel Pentium and Itanium, and the Motorola/IBM PowerPC.

Key to the success of Linux has been the availability of free software packages under the auspices of the Free Software Foundation (FSF). FSF's goal is stable, platform-independent software that is free, high quality, and embraced by the user community. FSF's GNU project[3] provides tools for software developers, and the

[3]GNU is a recursive acronym for *GNU's Not Unix*. The GNU project is a free software set of packages and tools for developing a UNIX-like operating system; it is often used with the Linux kernel.

ISBN 1-256-52023-3

GNU Public License (GPL) is the FSF seal of approval. Torvalds used GNU tools in developing his kernel, which he then released under the GPL. Thus, the Linux distributions that you see today are the product of FSF's GNU project, Torvald's individual effort, and the efforts of many collaborators all over the world.

In addition to its use by many individual programmers, Linux has now made significant penetration into the corporate world. This is not only because of the free software, but also because of the quality of the Linux kernel. Many talented programmers have contributed to the current version, resulting in a technically impressive product. Moreover, Linux is highly modular and easily configured. This makes it easy to squeeze optimal performance from a variety of hardware platforms. Plus, with the source code available, vendors can tweak applications and utilities to meet specific requirements. Throughout this book, we will provide details of Linux kernel internals based on the most recent version, Linux 2.6.

Modular Structure

Most UNIX kernels are monolithic. Recall from earlier in this chapter that a monolithic kernel is one that includes virtually all of the OS functionality in one large block of code that runs as a single process with a single address space. All the functional components of the kernel have access to all of its internal data structures and routines. If changes are made to any portion of a typical monolithic OS, all the modules and routines must be relinked and reinstalled and the system rebooted before the changes can take effect. As a result, any modification, such as adding a new device driver or file system function, is difficult. This problem is especially acute for Linux, for which development is global and done by a loosely associated group of independent programmers.

Although Linux does not use a microkernel approach, it achieves many of the potential advantages of this approach by means of its particular modular architecture. Linux is structured as a collection of modules, a number of which can be automatically loaded and unloaded on demand. These relatively independent blocks are referred to as **loadable modules** [GOYE99]. In essence, a module is an object file whose code can be linked to and unlinked from the kernel at runtime. Typically, a module implements some specific function, such as a file system, a device driver, or some other feature of the kernel's upper layer. A module does not execute as its own process or thread, although it can create kernel threads for various purposes as necessary. Rather, a module is executed in kernel mode on behalf of the current process.

Thus, although Linux may be considered monolithic, its modular structure overcomes some of the difficulties in developing and evolving the kernel.

The Linux loadable modules have two important characteristics:

- **Dynamic linking:** A kernel module can be loaded and linked into the kernel while the kernel is already in memory and executing. A module can also be unlinked and removed from memory at any time.
- **Stackable modules:** The modules are arranged in a hierarchy. Individual modules serve as libraries when they are referenced by client modules higher up in the hierarchy, and as clients when they reference modules further down.

ISBN 1-256-52023-3

Dynamic linking [FRAN97] facilitates configuration and saves kernel memory. In Linux, a user program or user can explicitly load and unload kernel modules using the insmod and rmmod commands. The kernel itself monitors the need for particular functions and can load and unload modules as needed. With stackable modules, dependencies between modules can be defined. This has two benefits:

1. Code common to a set of similar modules (e.g., drivers for similar hardware) can be moved into a single module, reducing replication.
2. The kernel can make sure that needed modules are present, refraining from unloading a module on which other running modules depend, and loading any additional required modules when a new module is loaded.

Figure 2.19 is an example that illustrates the structures used by Linux to manage modules. The figure shows the list of kernel modules after only two modules have been loaded: FAT and VFAT. Each module is defined by two tables, the module table and the symbol table. The module table includes the following elements:

- ***next:** Pointer to the following module. All modules are organized into a linked list. The list begins with a pseudomodule (not shown in Figure 2.19).
- ***name:** Pointer to module name
- **size:** Module size in memory pages
- **usecount:** Module usage counter. The counter is incremented when an operation involving the module's functions is started and decremented when the operation terminates.

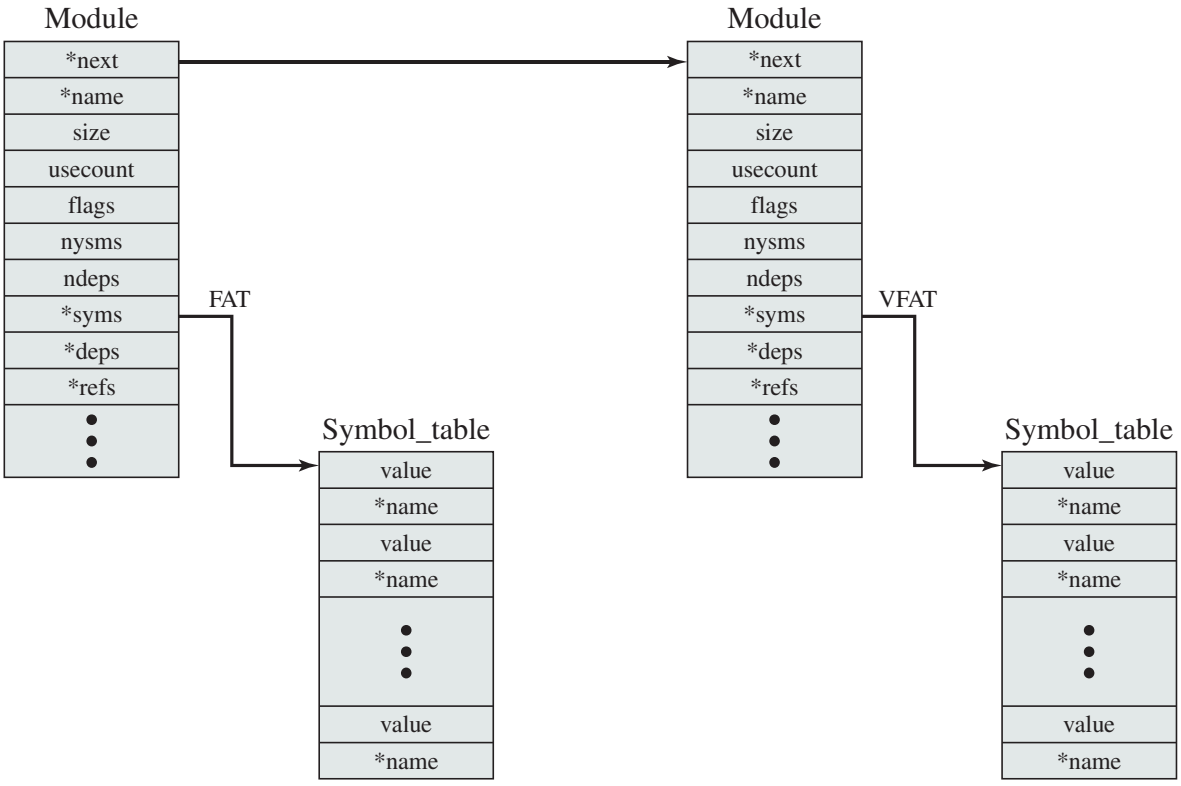


Figure 2.19 Example List of Linux Kernel Modules

ISBN 1-256-52023-3

- **flags:** Module flags
- **nsyms:** Number of exported symbols
- **ndeps:** Number of referenced modules
- ***syms:** Pointer to this module's symbol table.
- ***deps:** Pointer to list of modules that are referenced by this module.
- ***refs:** Pointer to list of modules that use this module.

The symbol table defines those symbols controlled by this module that are used elsewhere.

Figure 2.19 shows that the VFAT module was loaded after the FAT module and that the VFAT module is dependent on the FAT module.

Kernel Components

Figure 2.20, taken from [MOSB02], shows the main components of the Linux kernel as implemented on an IA-64 architecture (e.g., Intel Itanium). The figure shows several processes running on top of the kernel. Each box indicates a separate process, while each squiggly line with an arrowhead represents a thread of execution.[4] The kernel itself consists of an interacting collection of components, with arrows

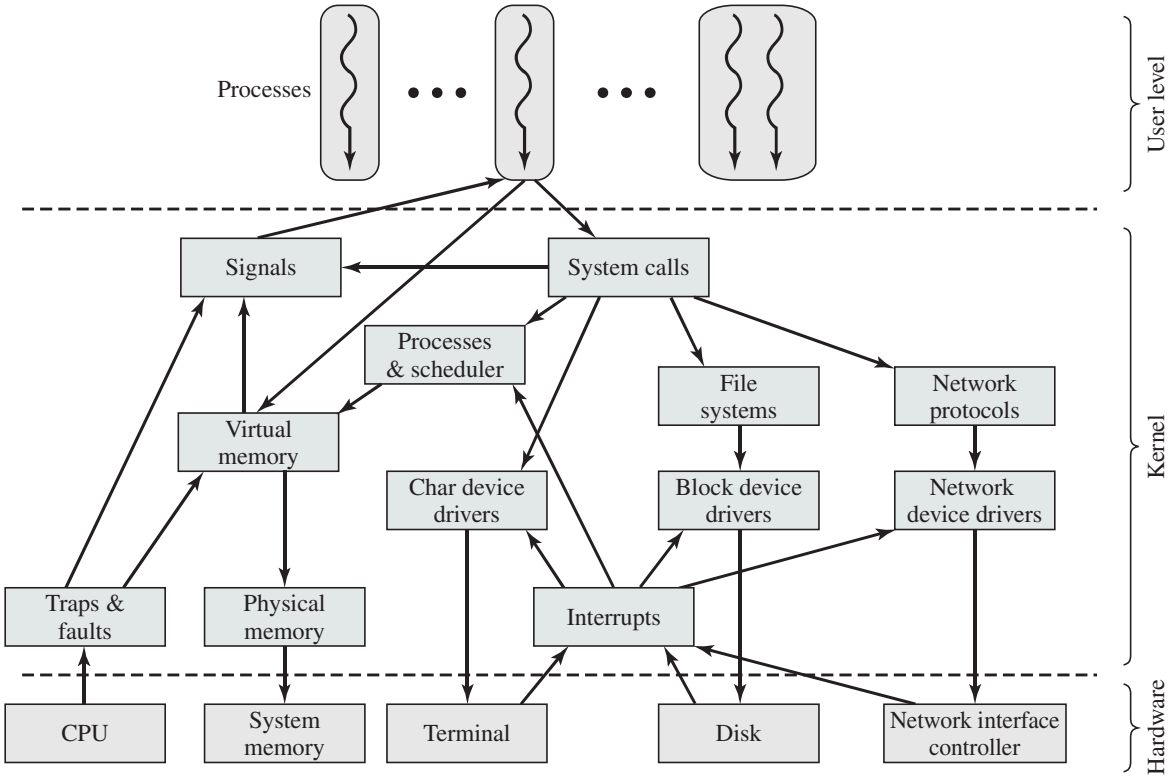


Figure 2.20 Linux Kernel Components

[4]In Linux, there is no distinction between the concepts of processes and threads. However, multiple threads in Linux can be grouped together in such a way that, effectively, you can have a single process comprising multiple threads. These matters are discussed in Chapter 4.

ISBN 1-256-52023-3

Table 2.5 Some Linux Signals

SIGHUP	Terminal hangup	SIGCONT	Continue
SIGQUIT	Keyboard quit	SIGTSTP	Keyboard stop
SIGTRAP	Trace trap	SIGTTOU	Terminal write
SIGBUS	Bus error	SIGXCPU	CPU limit exceeded
SIGKILL	Kill signal	SIGVTALRM	Virtual alarm clock
SIGSEGV	Segmentation violation	SIGWINCH	Window size unchanged
SIGPIPT	Broken pipe	SIGPWR	Power failure
SIGTERM	Termination	SIGRTMIN	First real-time signal
SIGCHLD	Child status unchanged	SIGRTMAX	Last real-time signal

indicating the main interactions. The underlying hardware is also depicted as a set of components with arrows indicating which kernel components use or control which hardware components. All of the kernel components, of course, execute on the processor but, for simplicity, these relationships are not shown.

Briefly, the principal kernel components are the following:

- **Signals:** The kernel uses signals to call into a process. For example, signals are used to notify a process of certain faults, such as division by zero. Table 2.5 gives a few examples of signals.
- **System calls:** The system call is the means by which a process requests a specific kernel service. There are several hundred system calls, which can be roughly grouped into six categories: file system, process, scheduling, interprocess communication, socket (networking), and miscellaneous. Table 2.6 defines a few examples in each category.
- **Processes and scheduler:** Creates, manages, and schedules processes.
- **Virtual memory:** Allocates and manages virtual memory for processes.

Table 2.6 Some Linux System Calls

File system Related	
close	Close a file descriptor.
link	Make a new name for a file.
open	Open and possibly create a file or device.
read	Read from file descriptor.
write	Write to file descriptor.
Process Related	
execve	Execute program.
exit	Terminate the calling process.
getpid	Get process identification.
setuid	Set user identity of the current process.
prtrace	Provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers.

ISBN 1-256-52023-3

Table 2.6 (*continued*)

Scheduling Related	
sched_getparam	Set the scheduling parameters associated with the scheduling policy for the process identified by `pid`.
sched_get_priority_max	Return the maximum priority value that can be used with the scheduling algorithm identified by `policy`.
sched_setscheduler	Set both the scheduling policy (e.g., FIFO) and the associated parameters for the process `pid`.
sched_rr_get_interval	Write into the timespec structure pointed to by the parameter `tp` the round-robin time quantum for the process `pid`.
sched_yield	A process can relinquish the processor voluntarily without blocking via this system call. The process will then be moved to the end of the queue for its static priority and a new process gets to run.
Interprocess Communication (IPC) Related	
msgrcv	A message buffer structure is allocated to receive a message. The system call then reads a message from the message queue specified by msqid into the newly created message buffer.
semctl	Perform the control operation specified by `cmd` on the semaphore set `semid`.
semop	Perform operations on selected members of the semaphore set `semid`.
shmat	Attach the shared memory segment identified by `semid` to the data segment of the calling process.
shmctl	Allow the user to receive information on a shared memory segment; set the owner, group, and permissions of a shared memory segment; or destroy a segment.
Socket (networking) Related	
bind	Assigns the local IP address and port for a socket. Returns 0 for success and −1 for error.
connect	Establish a connection between the given socket and the remote socket associated with sockaddr.
gethostname	Return local host name.
send	Send the bytes contained in buffer pointed to by *msg over the given socket.
setsockopt	Set the options on a socket
Miscellaneous	
create_module	Attempt to create a loadable module entry and reserve the kernel memory that will be needed to hold the module.
fsync	Copy all in-core parts of a file to disk, and waits until the device reports that all parts are on stable storage.
query_module	Request information related to loadable modules from the kernel.
time	Return the time in seconds since January 1, 1970.
vhangup	Simulate a hangup on the current terminal. This call arranges for other users to have a “clean” tty at login time.

ISBN 1-256-52023-3

- **File systems:** Provides a global, hierarchical namespace for files, directories, and other file related objects and provides file system functions.
- **Network protocols:** Supports the Sockets interface to users for the TCP/IP protocol suite.
- **Character device drivers:** Manages devices that require the kernel to send or receive data one byte at a time, such as terminals, modems, and printers.
- **Block device drivers:** Manages devices that read and write data in blocks, such as various forms of secondary memory (magnetic disks, CD-ROMs, etc.).
- **Network device drivers:** Manages network interface cards and communications ports that connect to network devices, such as bridges and routers.
- **Traps and faults:** Handles traps and faults generated by the processor, such as a memory fault.
- **Physical memory:** Manages the pool of page frames in real memory and allocates pages for virtual memory.
- **Interrupts:** Handles interrupts from peripheral devices.

2.11 LINUX VSERVER VIRTUAL MACHINE ARCHITECTURE

Linux VServer is an open-source, fast, lightweight approach to implementing virtual machines on a Linux server [SOLT07, LIGN05]. Only a single copy of the Linux kernel is involved. VServer consists of a relatively modest modification to the kernel plus a small set of OS userland[5] tools. The VServer Linux kernel supports a number of separate *virtual servers.* The kernel manages all system resources and tasks, including process scheduling, memory, disk space, and processor time. This is closer in concept to the process VM rather than the system VM of Figure 2.14.

Each virtual server is isolated from the others using Linux kernel capabilities. This provides security and makes it easy to set up multiple virtual machines on a single platform. The isolation involves four elements: chroot, chcontext, chbind, and capabilities.

The **chroot** command is a UNIX or Linux command to make the root directory (/) become something other than its default for the lifetime of the current process. It can only be run by privileged users and is used to give a process (commonly a network server such as FTP or HTTP) access to a restricted portion of the file system. This command provides **file system isolation**. All commands executed by the virtual server can only affect files that start with the defined root for that server.

The **chcontext** Linux utility allocates a new security context and executes commands in that context. The usual or *hosted* security context is the context 0. This context has the same privileges as the root user (UID 0): This context can see and kill other tasks in the other contexts. Context number 1 is used to view

[5]The term *userland* refers to all application software that runs in user space rather than kernel space. *OS userland* usually refers to the various programs and libraries that the operating system uses to interact with the kernel: software that performs input/output, manipulates file system objects, etc.

ISBN 1-256-52023-3

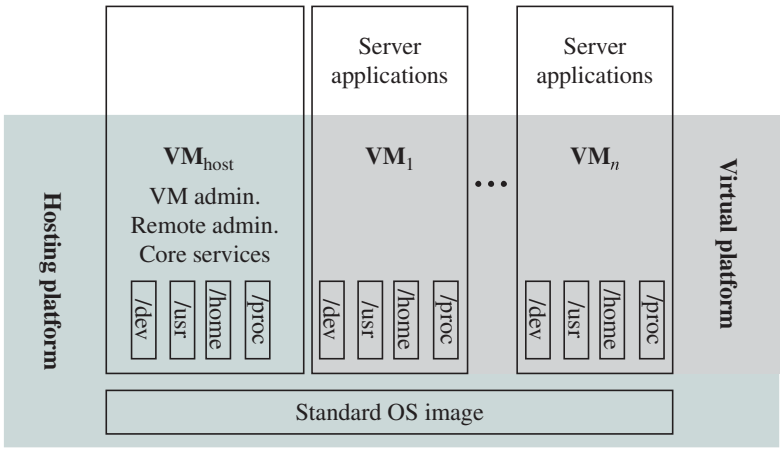


Figure 2.21 Linux VServer Architecture

other contexts but cannot affect them. All other contexts provide complete isolation: Processes from one context can neither see nor interact with processes from another context. This provides the ability to run similar contexts on the same computer without any interaction possible at the application level. Thus, each virtual server has its own execution context that provides **process isolation**.

The **chbind** utility executes a command, and locks the resulting process and its children into using a specific IP address. Once called, all packets sent out by this virtual server through the system's network interface are assigned the sending IP address derived from the argument given to chbind. This system call provides **network isolation**: Each virtual server uses a separate and distinct IP address. Incoming traffic intended for one virtual server cannot be accessed by other virtual servers.

Finally, each virtual server is assigned a set of **capabilities**. The concept of capabilities, as used in Linux, refers to a partitioning of the privileges available to a root user, such as the ability to read files or to trace processes owned by another user. Thus, each virtual server can be assigned a limited subset of the root user's privileges. This provides **root isolation**. VServer can also set resource limits, such as limits to the amount of virtual memory a process may use.

Figure 2.21, based on [SOLT07], shows the general architecture of Linux VServer. VServer provides a shared, virtualized OS image, consisting of a root file system, and a shared set of system libraries and kernel services. Each VM can be booted, shut down, and rebooted independently. Figure 2.21 shows three groupings of software running on the computer system. The **hosting platform** includes the shared OS image and a privileged host VM, whose function is to monitor and manage the other VMs. The **virtual platform** creates virtual machines and is the view of the system seen by the **applications** running on the individual VMs.

2.12 RECOMMENDED READING AND WEB SITES

[BRIN01] is an excellent collection of papers covering major advances in OS design over the years. [SWAI07] is a provocative and interesting short article on the future of operating systems.

ISBN 1-256-52023-3

[MUKH96] provides a good discussion of OS design issues for SMPs. [CHAP97] contains five articles on recent design directions for multiprocessor operating systems. Worthwhile discussions of the principles of microkernel design are contained in [LIED95] and [LIED96]; the latter focuses on performance issues.

[LI10] and [SMIT05] provide good treatments of virtual machines.

An excellent treatment of UNIX internals, which provides a comparative analysis of a number of variants, is [VAHA96]. For UNIX SVR4, [GOOD94] provides a definitive treatment, with ample technical detail. For the popular open-source FreeBSD, [MCKU05] is highly recommended. [MCDO07] provides a good treatment of Solaris internals. Good treatments of Linux internals are [LOVE10] and [MAUE08].

Although there are countless books on various versions of Windows, there is remarkably little material available on Windows internals. The book to read is [RUSS11].

BRIN01 Brinch Hansen, P. *Classic Operating Systems: From Batch Processing to Distributed Systems.* New York: Springer-Verlag, 2001.

CHAP97 Chapin, S., and Maccabe, A., eds. "Multiprocessor Operating Systems: Harnessing the Power." special issue of *IEEE Concurrency*, April–June 1997.

GOOD94 Goodheart, B., and Cox, J. *The Magic Garden Explained: The Internals of UNIX System V Release 4.* Englewood Cliffs, NJ: Prentice Hall, 1994.

LOVE10 Love, R. *Linux Kernel Development.* Upper Saddle River, NJ: Addison-Wesley, 2010.

LI10 Li, Y.; Li, W.; and Jiang, C. "A Survey of Virtual Machine Systems: Current Technology and Future Trends." *Proceedings, Third International Symposium on Electronic Commerce and Security*, 2010.

LIED95 Liedtke, J. "On μ-Kernel Construction." *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

LIED96 Liedtke, J. "Toward Real Microkernels." *Communications of the ACM*, September 1996.

MAUE08 Mauerer, W. *Professional Linux Kernel Architecture.* New York: Wiley, 2008.

MCDO07 McDougall, R., and Mauro, J. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture.* Palo Alto, CA: Sun Microsystems Press, 2007.

MCKU05 McKusick, M., and Neville-Neil, J. *The Design and Implementation of the FreeBSD Operating System.* Reading, MA: Addison-Wesley, 2005.

MUKH96 Mukherjee, B., and Karsten, S. "Operating Systems for Parallel Machines." In *Parallel Computers: Theory and Practice.* Edited by T. Casavant, P. Tvrkik, and F. Plasil. Los Alamitos, CA: IEEE Computer Society Press, 1996.

RUSS11 Russinovich, M.; Solomon, D.; and Ionescu, A. *Windows Internals: Covering Windows 7 and Windows Server 2008 R2.* Redmond, WA: Microsoft Press, 2011.

SMIT05 Smith, J., and Nair, R. "The Architecture of Virtual Machines." *Computer*, May 2005.

SWAI07 Swaine, M. "Wither Operating Systems?" *Dr. Dobb's Journal*, March 2007.

VAHA96 Vahalia, U. *UNIX Internals: The New Frontiers.* Upper Saddle River, NJ: Prentice Hall, 1996.

ISBN 1-256-52023-3

Recommended Web sites:

- **The Operating System Resource Center:** A useful collection of documents and papers on a wide range of OS topics.
- **Operating System Technical Comparison:** Includes a substantial amount of information on a variety of operating systems.
- **ACM Special Interest Group on Operating Systems:** Information on SIGOPS publications and conferences.
- **IEEE Technical Committee on Operating Systems and Application Environments:** Includes an online newsletter and links to other sites.
- **The comp.os.research FAQ:** Lengthy and worthwhile FAQ covering OS design issues.
- **UNIX Guru Universe:** Excellent source of UNIX information.
- **Linux Documentation Project:** The name describes the site.
- **IBM's Linux Website:** Provides a wide range of technical and user information on Linux. Much of it is devoted to IBM products but there is a lot of useful general technical information.
- **Windows Development:** Good source of information on Windows internals.

2.13 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

batch processing	multiprogramming	round robin
batch system	multitasking	scheduling
execution context	multithreading	serial processing
interrupt	nucleus	symmetric multiprocessing
job	operating system	task
job control language	physical address	thread
kernel	privileged instruction	time sharing
memory management	process	time-sharing system
microkernel	process state	uniprogramming
monitor	real address	virtual address
monolithic kernel	resident monitor	virtual machine
multiprogrammed batch system		

Review Questions

2.1 What are three objectives of an OS design?
2.2 What is the kernel of an OS?
2.3 What is multiprogramming?
2.4 What is a process?
2.5 How is the execution context of a process used by the OS?

ISBN 1-256-52023-3

2.6 List and briefly explain five storage management responsibilities of a typical OS.
2.7 Explain the distinction between a real address and a virtual address.
2.8 Describe the round-robin scheduling technique.
2.9 Explain the difference between a monolithic kernel and a microkernel.
2.10 What is multithreading?
2.11 List the key design issues for an SMP operating system.

Problems

2.1 Suppose that we have a multiprogrammed computer in which each job has identical characteristics. In one computation period, T, for a job, half the time is spent in I/O and the other half in processor activity. Each job runs for a total of N periods. Assume that a simple round-robin scheduling is used, and that I/O operations can overlap with processor operation. Define the following quantities:

- Turnaround time = actual time to complete a job
- Throughput = average number of jobs completed per time period T
- Processor utilization = percentage of time that the processor is active (not waiting)

Compute these quantities for one, two, and four simultaneous jobs, assuming that the period T is distributed in each of the following ways:

a. I/O first half, processor second half
b. I/O first and fourth quarters, processor second and third quarter

2.2 An I/O-bound program is one that, if run alone, would spend more time waiting for I/O than using the processor. A processor-bound program is the opposite. Suppose a short-term scheduling algorithm favors those programs that have used little processor time in the recent past. Explain why this algorithm favors I/O-bound programs and yet does not permanently deny processor time to processor-bound programs.

2.3 Contrast the scheduling policies you might use when trying to optimize a time-sharing system with those you would use to optimize a multiprogrammed batch system.

2.4 What is the purpose of system calls, and how do system calls relate to the OS and to the concept of dual-mode (kernel-mode and user-mode) operation?

2.5 In IBM's mainframe OS, OS/390, one of the major modules in the kernel is the System Resource Manager. This module is responsible for the allocation of resources among address spaces (processes). The SRM gives OS/390 a degree of sophistication unique among operating systems. No other mainframe OS, and certainly no other type of OS, can match the functions performed by SRM. The concept of resource includes processor, real memory, and I/O channels. SRM accumulates statistics pertaining to utilization of processor, channel, and various key data structures. Its purpose is to provide optimum performance based on performance monitoring and analysis. The installation sets forth various performance objectives, and these serve as guidance to the SRM, which dynamically modifies installation and job performance characteristics based on system utilization. In turn, the SRM provides reports that enable the trained operator to refine the configuration and parameter settings to improve user service.

This problem concerns one example of SRM activity. Real memory is divided into equal-sized blocks called frames, of which there may be many thousands. Each frame can hold a block of virtual memory referred to as a page. SRM receives control approximately 20 times per second and inspects each and every page frame. If the page has not been referenced or changed, a counter is incremented by 1. Over time, SRM averages these numbers to determine the average number of seconds that a page frame in the system goes untouched. What might be the purpose of this and what action might SRM take?

2.6 A multiprocessor with eight processors has 20 attached tape drives. There is a large number of jobs submitted to the system that each require a maximum of four tape

ISBN 1-256-52023-3

drives to complete execution. Assume that each job starts running with only three tape drives for a long period before requiring the fourth tape drive for a short period toward the end of its operation. Also assume an endless supply of such jobs.

a. Assume the scheduler in the OS will not start a job unless there are four tape drives available. When a job is started, four drives are assigned immediately and are not released until the job finishes. What is the maximum number of jobs that can be in progress at once? What are the maximum and minimum number of tape drives that may be left idle as a result of this policy?

b. Suggest an alternative policy to improve tape drive utilization and at the same time avoid system deadlock. What is the maximum number of jobs that can be in progress at once? What are the bounds on the number of idling tape drives?

ISBN 1-256-52023-3

PART 2 Processes

CHAPTER 3

PROCESS DESCRIPTION AND CONTROL

ISBN 1-256-52023-3

The concept of process is fundamental to the structure of modern computer operating systems. Its evolution in analyzing problems of synchronization, deadlock, and scheduling in operating systems has been a major intellectual contribution of computer science.

WHAT CAN BE AUTOMATED?: THE COMPUTER SCIENCE AND ENGINEERING RESEARCH STUDY, MIT PRESS, 1980

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Define the term *process* and explain the relationship between processes and process control blocks.
- Explain the concept of a process state and discuss the state transitions the processes undergo.
- List and describe the purpose of the data structures and data structure elements used by an OS to manage processes.
- Assess the requirements for process control by the OS.
- Understand the issues involved in the execution of OS code.
- Assess the key security issues that relate to operating systems.
- Describe the process management scheme for UNIX SVR4.

All multiprogramming operating systems, from single-user systems such as Windows for end users to mainframe systems such as IBM's mainframe operating system, z/OS, which can support thousands of users, are built around the concept of the process. Most requirements that the OS must meet can be expressed with reference to processes:

- The OS must interleave the execution of multiple processes, to maximize processor utilization while providing reasonable response time.
- The OS must allocate resources to processes in conformance with a specific policy (e.g., certain functions or applications are of higher priority) while at the same time avoiding deadlock.[1]
- The OS may be required to support interprocess communication and user creation of processes, both of which may aid in the structuring of applications.

We begin with an examination of the way in which the OS represents and controls processes. Then, the chapter discusses process states, which characterize the behavior of processes. Then we look at the data structures that the OS uses to manage processes. These include data structures to represent the state of each

[1]Deadlock is examined in Chapter 6. As a simple example, deadlock occurs if two processes need the same two resources to continue and each has ownership of one. Unless some action is taken, each process will wait indefinitely for the missing resource.

ISBN 1-256-52023-3

process and data structures that record other characteristics of processes that the OS needs to achieve its objectives. Next, we look at the ways in which the OS uses these data structures to control process execution. Finally, we discuss process management in UNIX SVR4. Chapter 4 provides more modern examples of process management.

This chapter occasionally refers to virtual memory. Much of the time, we can ignore this concept in dealing with processes, but at certain points in the discussion, virtual memory considerations are pertinent. Virtual memory is previewed in Chapter 2 and discussed in detail in Chapter 8. A set of animations that illustrate concepts in this chapter is available online. Click on the rotating globe at this book's Web site at WilliamStallings.com/OS/OS7e.html for access.

3.1 WHAT IS A PROCESS?

Background

Before defining the term *process*, it is useful to summarize some of the concepts introduced in Chapters 1 and 2:

1. A computer platform consists of a collection of hardware resources, such as the processor, main memory, I/O modules, timers, disk drives, and so on.
2. Computer applications are developed to perform some task. Typically, they accept input from the outside world, perform some processing, and generate output.
3. It is inefficient for applications to be written directly for a given hardware platform. The principal reasons for this are as follows:
   a. Numerous applications can be developed for the same platform. Thus, it makes sense to develop common routines for accessing the computer's resources.
   b. The processor itself provides only limited support for multiprogramming. Software is needed to manage the sharing of the processor and other resources by multiple applications at the same time.
   c. When multiple applications are active at the same time, it is necessary to protect the data, I/O use, and other resource use of each application from the others.
4. The OS was developed to provide a convenient, feature-rich, secure, and consistent interface for applications to use. The OS is a layer of software between the applications and the computer hardware (Figure 2.1) that supports applications and utilities.
5. We can think of the OS as providing a uniform, abstract representation of resources that can be requested and accessed by applications. Resources include main memory, network interfaces, file systems, and so on. Once the OS has created these resource abstractions for applications to use, it must also manage their use. For example, an OS may permit resource sharing and resource protection.

ISBN 1-256-52023-3

Now that we have the concepts of applications, system software, and resources, we are in a position to discuss how the OS can, in an orderly fashion, manage the execution of applications so that

- Resources are made available to multiple applications.
- The physical processor is switched among multiple applications so all will appear to be progressing.
- The processor and I/O devices can be used efficiently.

The approach taken by all modern operating systems is to rely on a model in which the execution of an application corresponds to the existence of one or more processes.

Processes and Process Control Blocks

Recall from Chapter 2 that we suggested several definitions of the term *process*, including

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources

We can also think of a process as an entity that consists of a number of elements. Two essential elements of a process are **program code** (which may be shared with other processes that are executing the same program) and a **set of data** associated with that code. Let us suppose that the processor begins to execute this program code, and we refer to this executing entity as a process. At any given point in time, *while the program is executing*, this process can be uniquely characterized by a number of elements, including the following:

- **Identifier:** A unique identifier associated with this process, to distinguish it from all other processes.
- **State:** If the process is currently executing, it is in the running state.
- **Priority:** Priority level relative to other processes.
- **Program counter:** The address of the next instruction in the program to be executed.
- **Memory pointers:** Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data:** These are data that are present in registers in the processor while the process is executing.
- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., disk drives) assigned to this process, a list of files in use by the process, and so on.
- **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

ISBN 1-256-52023-3

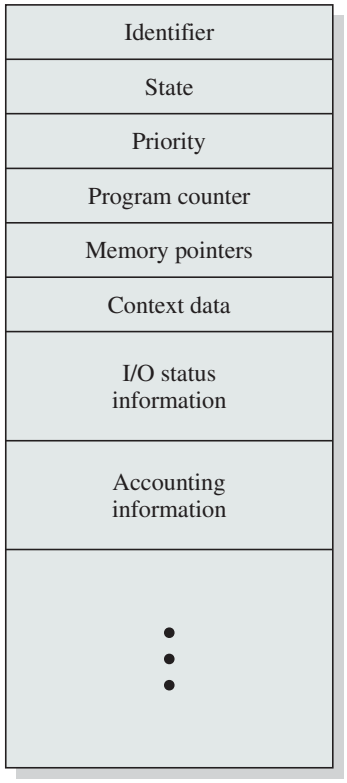


Figure 3.1 Simplified Process Control Block

The information in the preceding list is stored in a data structure, typically called a **process control block** (Figure 3.1), that is created and managed by the OS. The significant point about the process control block is that it contains sufficient information so that it is possible to interrupt a running process and later resume execution as if the interruption had not occurred. The process control block is the key tool that enables the OS to support multiple processes and to provide for multiprocessing. When a process is interrupted, the current values of the program counter and the processor registers (context data) are saved in the appropriate fields of the corresponding process control block, and the state of the process is changed to some other value, such as *blocked* or *ready* (described subsequently). The OS is now free to put some other process in the running state. The program counter and context data for this process are loaded into the processor registers and this process now begins to execute.

Thus, we can say that a process consists of program code and associated data plus a process control block. For a single-processor computer, at any given time, at most one process is executing and that process is in the *running* state.

3.2 PROCESS STATES

As just discussed, for a program to be executed, a process, or task, is created for that program. From the processor's point of view, it executes instructions from its repertoire in some sequence dictated by the changing values in the program counter

ISBN 1-256-52023-3

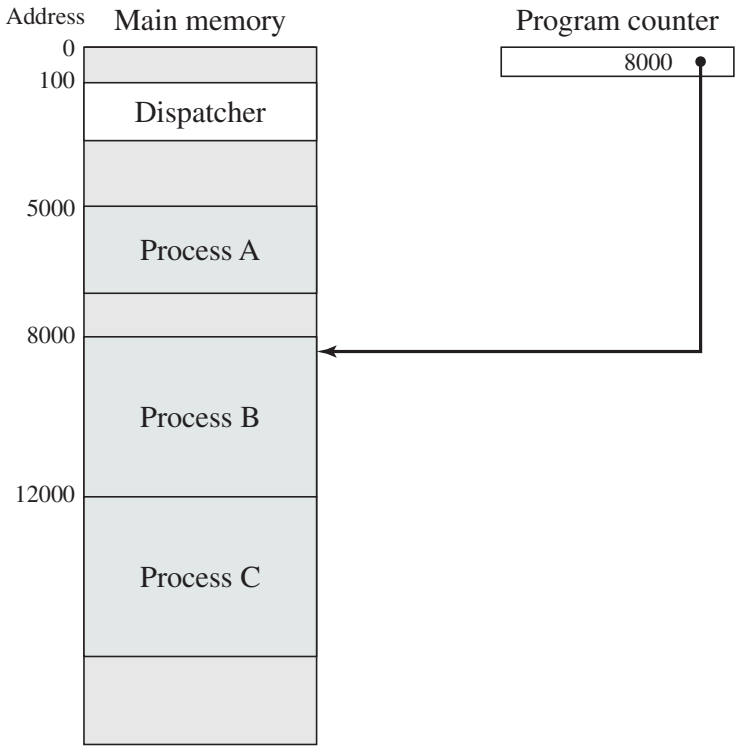


Figure 3.2 **Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13**

register. Over time, the program counter may refer to code in different programs that are part of different processes. From the point of view of an individual program, its execution involves a sequence of instructions within that program.

We can characterize the behavior of an individual process by listing the sequence of instructions that execute for that process. Such a listing is referred to as a **trace** of the process. We can characterize behavior of the processor by showing how the traces of the various processes are interleaved.

Let us consider a very simple example. Figure 3.2 shows a memory layout of three processes. To simplify the discussion, we assume no use of virtual memory; thus all three processes are represented by programs that are fully loaded in main memory. In addition, there is a small **dispatcher** program that switches the processor from one process to another. Figure 3.3 shows the traces of each of the processes during the early part of their execution. The first 12 instructions executed in processes A and C are shown. Process B executes four instructions, and we assume that the fourth instruction invokes an I/O operation for which the process must wait.

Now let us view these traces from the processor's point of view. Figure 3.4 shows the interleaved traces resulting from the first 52 instruction cycles (for convenience, the instruction cycles are numbered). In this figure, the shaded areas represent code executed by the dispatcher. The same sequence of instructions is executed by the dispatcher in each instance because the same functionality of the dispatcher is being executed. We assume that the OS only allows a process to continue execution for a maximum of six instruction cycles, after which it is interrupted;

ISBN 1-256-52023-3

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of process A (b) Trace of process B (c) Trace of process C

5000 = Starting address of program of process A
8000 = Starting address of program of process B
12000 = Starting address of program of process C

Figure 3.3 Traces of Processes of Figure 3.2

this prevents any single process from monopolizing processor time. As Figure 3.4 shows, the first six instructions of process A are executed, followed by a time-out and the execution of some code in the dispatcher, which executes six instructions before turning control to process B.[2] After four instructions are executed, process B requests an I/O action for which it must wait. Therefore, the processor stops executing process B and moves on, via the dispatcher, to process C. After a time-out, the processor moves back to process A. When this process times out, process B is still waiting for the I/O operation to complete, so the dispatcher moves on to process C again.

A Two-State Process Model

The operating system's principal responsibility is controlling the execution of processes; this includes determining the interleaving pattern for execution and allocating resources to processes. The first step in designing an OS to control processes is to describe the behavior that we would like the processes to exhibit.

We can construct the simplest possible model by observing that, at any time, a process is either being executed by a processor or not. In this model, a process may be in one of two states: Running or Not Running, as shown in Figure 3.5a. When the OS creates a new process, it creates a process control block for the process and enters that process into the system in the Not Running state. The process exists, is known to the OS, and is waiting for an opportunity to execute. From time to time, the currently running process will be interrupted and the dispatcher portion of the OS will select some other process to run. The former process moves from the

[2]The small number of instructions executed for the processes and the dispatcher are unrealistically low; they are used in this simplified example to clarify the discussion.

ISBN 1-256-52023-3

1	5000	27	12004
2	5001	28	12005
3	5002	----------Time-out	
4	5003	29	100
5	5004	30	101
6	5005	31	102
----------Time-out		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	----------Time-out	
16	8003	41	100
----------I/O request		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
		----------Time-out	

100 = Starting address of dispatcher program

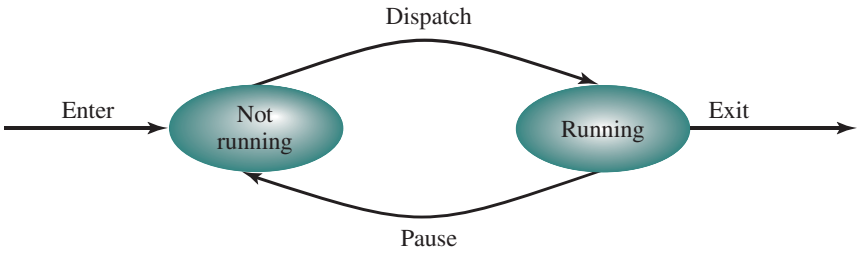
Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

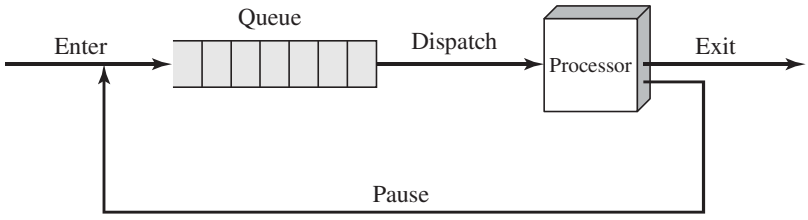
Running state to the Not Running state, and one of the other processes moves to the Running state.

From this simple model, we can already begin to appreciate some of the design elements of the OS. Each process must be represented in some way so that the OS can keep track of it. That is, there must be some information relating to each process, including current state and location in memory; this is the process control block. Processes that are not running must be kept in some sort of queue, waiting their turn to execute. Figure 3.5b suggests a structure. There is a single queue in which each entry is a pointer to the process control block of a particular process. Alternatively,

ISBN 1-256-52023-3



(a) State transition diagram



(b) Queueing diagram

Figure 3.5 Two-State Process Model

the queue may consist of a linked list of data blocks, in which each block represents one process; we will explore this latter implementation subsequently.

We can describe the behavior of the dispatcher in terms of this queueing diagram. A process that is interrupted is transferred to the queue of waiting processes. Alternatively, if the process has completed or aborted, it is discarded (exits the system). In either case, the dispatcher takes another process from the queue to execute.

The Creation and Termination of Processes

Before refining our simple two-state model, it will be useful to discuss the creation and termination of processes; ultimately, and regardless of the model of process behavior that is used, the life of a process is bounded by its creation and termination.

Process Creation When a new process is to be added to those currently being managed, the OS builds the data structures that are used to manage the process and allocates address space in main memory to the process. We describe these data structures in Section 3.3. These actions constitute the creation of a new process.

Four common events lead to the creation of a process, as indicated in Table 3.1. In a batch environment, a process is created in response to the submission of a job. In an interactive environment, a process is created when a new user attempts to log on. In both cases, the OS is responsible for the creation of the new process. An OS may also create a process on behalf of an application. For example, if a user requests that a file be printed, the OS can create a process that will manage the printing. The requesting process can thus proceed independently of the time required to complete the printing task.

ISBN 1-256-52023-3

Table 3.1 Reasons for Process Creation

New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive log-on	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Traditionally, the OS created all processes in a way that was transparent to the user or application program, and this is still commonly found with many contemporary operating systems. However, it can be useful to allow one process to cause the creation of another. For example, an application process may generate another process to receive data that the application is generating and to organize those data into a form suitable for later analysis. The new process runs in parallel to the original process and is activated from time to time when new data are available. This arrangement can be very useful in structuring the application. As another example, a server process (e.g., print server, file server) may generate a new process for each request that it handles. When the OS creates a process at the explicit request of another process, the action is referred to as **process spawning**.

When one process spawns another, the former is referred to as the **parent process,** and the spawned process is referred to as the **child process.** Typically, the "related" processes need to communicate and cooperate with each other. Achieving this cooperation is a difficult task for the programmer; this topic is discussed in Chapter 5.

PROCESS TERMINATION Table 3.2 summarizes typical reasons for process termination. Any computer system must provide a means for a process to indicate its completion. A batch job should include a Halt instruction or an explicit OS service call for termination. In the former case, the Halt instruction will generate an interrupt to alert the OS that a process has completed. For an interactive application, the action of the user will indicate when the process is completed. For example, in a time-sharing system, the process for a particular user is to be terminated when the user logs off or turns off his or her terminal. On a personal computer or workstation, a user may quit an application (e.g., word processing or spreadsheet). All of these actions ultimately result in a service request to the OS to terminate the requesting process.

Additionally, a number of error and fault conditions can lead to the termination of a process. Table 3.2 lists some of the more commonly recognized conditions.[3]

Finally, in some operating systems, a process may be terminated by the process that created it or when the parent process is itself terminated.

[3]A forgiving operating system might, in some cases, allow the user to recover from a fault without terminating the process. For example, if a user requests access to a file and that access is denied, the operating system might simply inform the user that access is denied and allow the process to proceed.

ISBN 1-256-52023-3

Table 3.2 Reasons for Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

A Five-State Model

If all processes were always ready to execute, then the queueing discipline suggested by Figure 3.5b would be effective. The queue is a first-in-first-out list and the processor operates in **round-robin** fashion on the available processes (each process in the queue is given a certain amount of time, in turn, to execute and then returned to the queue, unless blocked). However, even with the simple example that we have described, this implementation is inadequate: Some processes in the Not Running state are ready to execute, while others are blocked, waiting for an I/O operation to complete. Thus, using a single queue, the dispatcher could not just select the process at the oldest end of the queue. Rather, the dispatcher would have to scan the list looking for the process that is not blocked and that has been in the queue the longest.

A more natural way to handle this situation is to split the Not Running state into two states: Ready and Blocked. This is shown in Figure 3.6. For good measure,

ISBN 1-256-52023-3

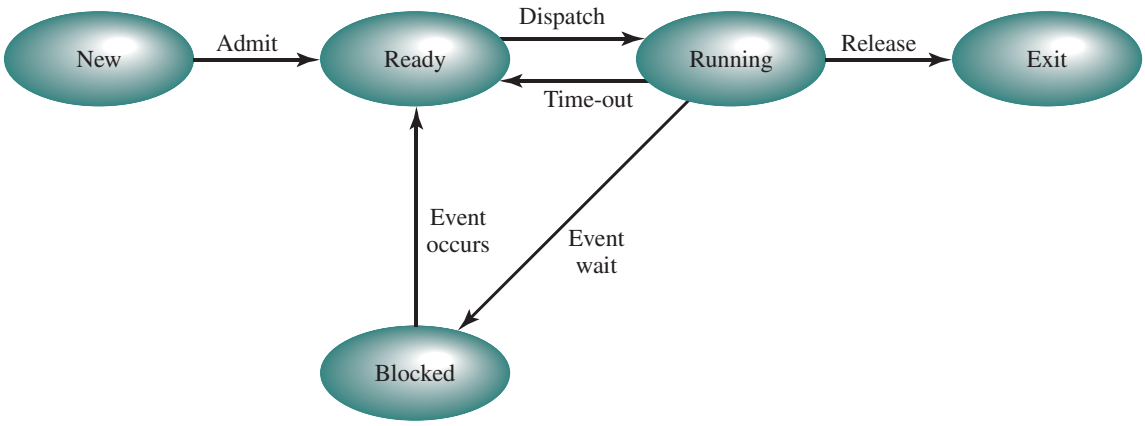


Figure 3.6 Five-State Process Model

we have added two additional states that will prove useful. The five states in this new diagram are:

- **Running:** The process that is currently being executed. For this chapter, we will assume a computer with a single processor, so at most one process at a time can be in this state.
- **Ready:** A process that is prepared to execute when given the opportunity.
- **Blocked/Waiting:**[4] A process that cannot execute until some event occurs, such as the completion of an I/O operation.
- **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the OS. Typically, a new process has not yet been loaded into main memory, although its process control block has been created.
- **Exit:** A process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason.

The New and Exit states are useful constructs for process management. The New state corresponds to a process that has just been defined. For example, if a new user attempts to log on to a time-sharing system or a new batch job is submitted for execution, the OS can define a new process in two stages. First, the OS performs the necessary housekeeping chores. An identifier is associated with the process. Any tables that will be needed to manage the process are allocated and built. At this point, the process is in the New state. This means that the OS has performed the necessary actions to create the process but has not committed itself to the execution of the process. For example, the OS may limit the number of processes that may be in the system for reasons of performance or main memory limitation. While a process is in the new state, information concerning the process that is needed by the OS is maintained in control tables in main memory. However, the process itself is

[4] *Waiting* is a frequently used alternative term for *Blocked* as a process state. Generally, we will use *Blocked*, but the terms are interchangeable.

ISBN 1-256-52023-3

not in main memory. That is, the code of the program to be executed is not in main memory, and no space has been allocated for the data associated with that program. While the process is in the New state, the program remains in secondary storage, typically disk storage.[5]

Similarly, a process exits a system in two stages. First, a process is terminated when it reaches a natural completion point, when it aborts due to an unrecoverable error, or when another process with the appropriate authority causes the process to abort. Termination moves the process to the exit state. At this point, the process is no longer eligible for execution. The tables and other information associated with the job are temporarily preserved by the OS, which provides time for auxiliary or support programs to extract any needed information. For example, an accounting program may need to record the processor time and other resources utilized by the process for billing purposes. A utility program may need to extract information about the history of the process for purposes related to performance or utilization analysis. Once these programs have extracted the needed information, the OS no longer needs to maintain any data relating to the process and the process is deleted from the system.

Figure 3.6 indicates the types of events that lead to each state transition for a process; the possible transitions are as follows:

- **Null → New:** A new process is created to execute a program. This event occurs for any of the reasons listed in Table 3.1.
- **New → Ready:** The OS will move a process from the New state to the Ready state when it is prepared to take on an additional process. Most systems set some limit based on the number of existing processes or the amount of virtual memory committed to existing processes. This limit assures that there are not so many active processes as to degrade performance.
- **Ready → Running:** When it is time to select a process to run, the OS chooses one of the processes in the Ready state. This is the job of the scheduler or dispatcher. Scheduling is explored in Part Four.
- **Running → Exit:** The currently running process is terminated by the OS if the process indicates that it has completed, or if it aborts. See Table 3.2.
- **Running → Ready:** The most common reason for this transition is that the running process has reached the maximum allowable time for uninterrupted execution; virtually all multiprogramming operating systems impose this type of time discipline. There are several other alternative causes for this transition, which are not implemented in all operating systems. Of particular importance is the case in which the OS assigns different levels of priority to different processes. Suppose, for example, that process A is running at a given priority level, and process B, at a higher priority level, is blocked. If the OS learns that the event upon which process B has been waiting has occurred, moving B to a ready state, then it can interrupt process A and dispatch process B. We

[5]In the discussion in this paragraph, we ignore the concept of virtual memory. In systems that support virtual memory, when a process moves from New to Ready, its program code and data are loaded into virtual memory. Virtual memory was briefly discussed in Chapter 2 and is examined in detail in Chapter 8.

ISBN 1-256-52023-3

say that the OS has **preempted** process A.[6] Finally, a process may voluntarily release control of the processor. An example is a background process that performs some accounting or maintenance function periodically.

- **Running → Blocked:** A process is put in the Blocked state if it requests something for which it must wait. A request to the OS is usually in the form of a system service call; that is, a call from the running program to a procedure that is part of the operating system code. For example, a process may request a service from the OS that the OS is not prepared to perform immediately. It can request a resource, such as a file or a shared section of virtual memory, that is not immediately available. Or the process may initiate an action, such as an I/O operation, that must be completed before the process can continue. When processes communicate with each other, a process may be blocked when it is waiting for another process to provide data or waiting for a message from another process.
- **Blocked → Ready:** A process in the Blocked state is moved to the Ready state when the event for which it has been waiting occurs.
- **Ready → Exit:** For clarity, this transition is not shown on the state diagram. In some systems, a parent may terminate a child' process at any time. Also, if a parent terminates, all child processes associated with that parent may be terminated.
- **Blocked → Exit:** The comments under the preceding item apply.

Returning to our simple example, Figure 3.7 shows the transition of each process among the states. Figure 3.8a suggests the way in which a queueing discipline might be implemented with two queues: a Ready queue and a Blocked queue. As each process is admitted to the system, it is placed in the Ready queue. When it is time for the OS to choose another process to run, it selects one from the Ready

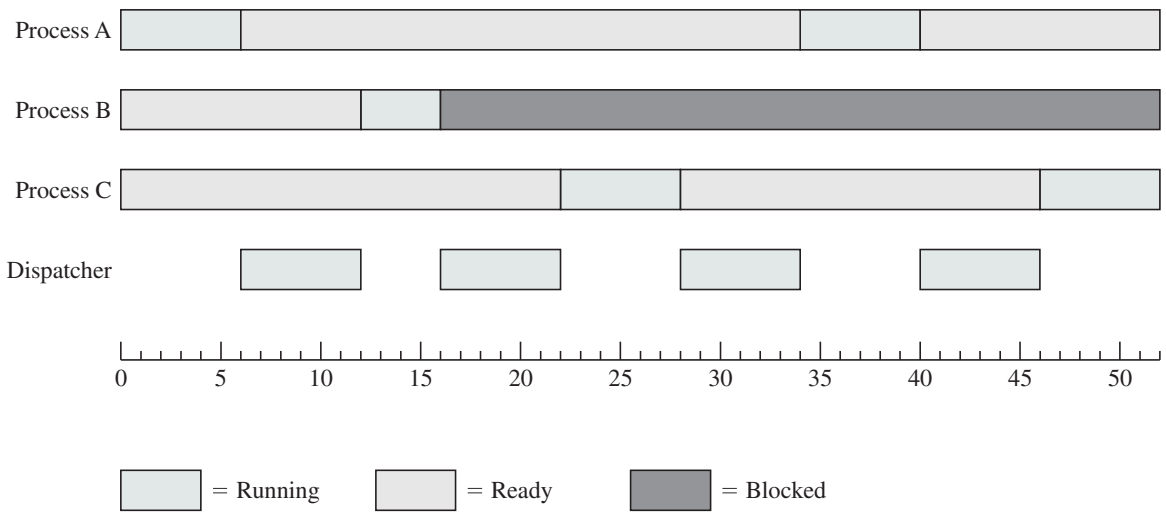
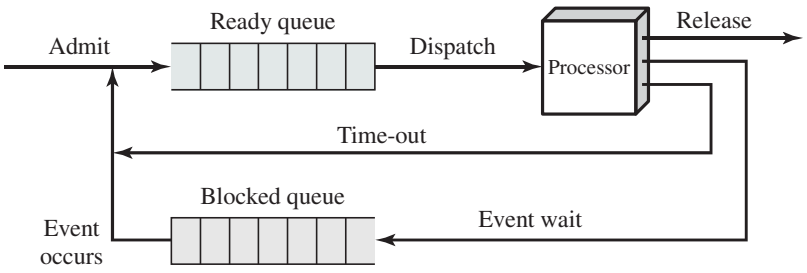


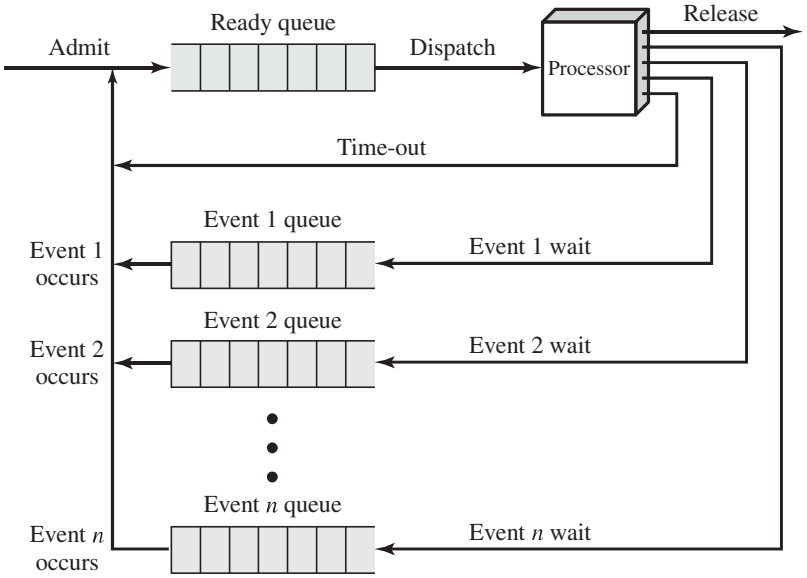
Figure 3.7 Process States for the Trace of Figure 3.4

[6]In general, the term *preemption* is defined to be the reclaiming of a resource from a process before the process has finished using it. In this case, the resource is the processor itself. The process is executing and could continue to execute, but is preempted so that another process can be executed.

ISBN 1-256-52023-3



(a) Single blocked queue



(b) Multiple blocked queues

Figure 3.8 Queueing Model for Figure 3.6

queue. In the absence of any priority scheme, this can be a simple first-in-first-out queue. When a running process is removed from execution, it is either terminated or placed in the Ready or Blocked queue, depending on the circumstances. Finally, when an event occurs, any process in the Blocked queue that has been waiting on that event only is moved to the Ready queue.

This latter arrangement means that, when an event occurs, the OS must scan the entire blocked queue, searching for those processes waiting on that event. In a large OS, there could be hundreds or even thousands of processes in that queue. Therefore, it would be more efficient to have a number of queues, one for each event. Then, when the event occurs, the entire list of processes in the appropriate queue can be moved to the Ready state (Figure 3.8b).

One final refinement: If the dispatching of processes is dictated by a priority scheme, then it would be convenient to have a number of Ready queues, one for each priority level. The OS could then readily determine which is the highest-priority ready process that has been waiting the longest.

ISBN 1-256-52023-3

Suspended Processes

THE NEED FOR SWAPPING The three principal states just described (Ready, Running, Blocked) provide a systematic way of modeling the behavior of processes and guide the implementation of the OS. Some operating systems are constructed using just these three states.

However, there is good justification for adding other states to the model. To see the benefit of these new states, consider a system that does not employ virtual memory. Each process to be executed must be loaded fully into main memory. Thus, in Figure 3.8b, all of the processes in all of the queues must be resident in main memory.

Recall that the reason for all of this elaborate machinery is that I/O activities are much slower than computation and therefore the processor in a uniprogramming system is idle most of the time. But the arrangement of Figure 3.8b does not entirely solve the problem. It is true that, in this case, memory holds multiple processes and that the processor can move to another process when one process is blocked. But the processor is so much faster than I/O that it will be common for all of the processes in memory to be waiting for I/O. Thus, even with multiprogramming, a processor could be idle most of the time.

What to do? Main memory could be expanded to accommodate more processes. But there are two flaws in this approach. First, there is a cost associated with main memory, which, though small on a per-byte basis, begins to add up as we get into the gigabytes of storage. Second, the appetite of programs for memory has grown as fast as the cost of memory has dropped. So larger memory results in larger processes, not more processes.

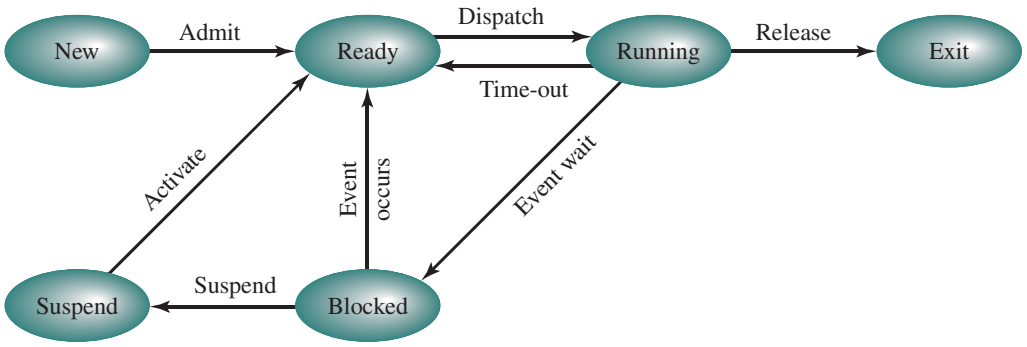
Another solution is swapping, which involves moving part or all of a process from main memory to disk. When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue. This is a queue of existing processes that have been temporarily kicked out of main memory, or suspended. The OS then brings in another process from the suspend queue, or it honors a new-process request. Execution then continues with the newly arrived process.

Swapping, however, is an I/O operation, and therefore there is the potential for making the problem worse, not better. But because disk I/O is generally the fastest I/O on a system (e.g., compared to tape or printer I/O), swapping will usually enhance performance.

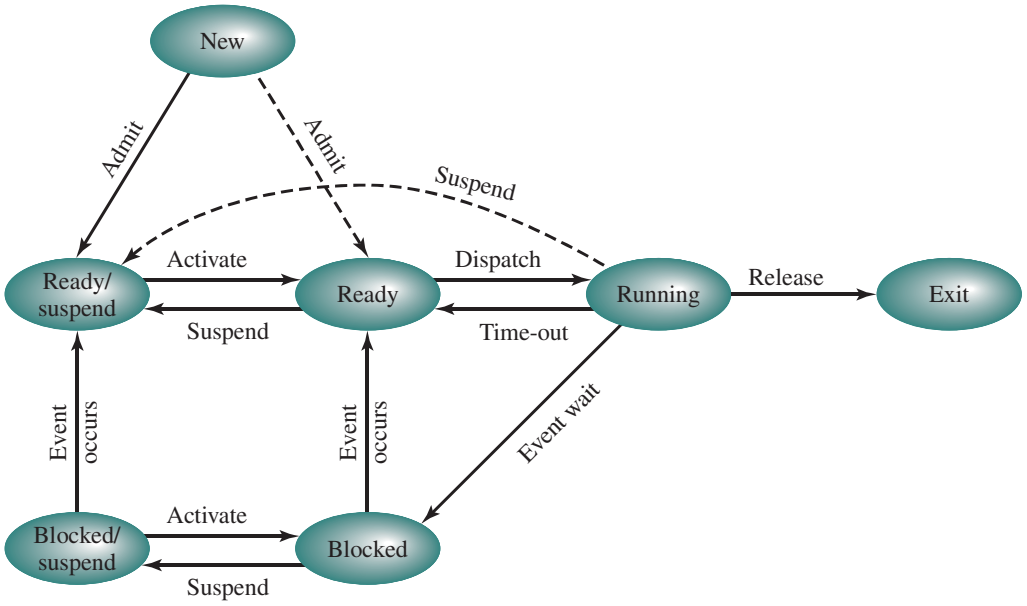
With the use of swapping as just described, one other state must be added to our process behavior model (Figure 3.9a): the Suspend state. When all of the processes in main memory are in the Blocked state, the OS can suspend one process by putting it in the Suspend state and transferring it to disk. The space that is freed in main memory can then be used to bring in another process.

When the OS has performed a swapping-out operation, it has two choices for selecting a process to bring into main memory: It can admit a newly created process or it can bring in a previously suspended process. It would appear that the preference should be to bring in a previously suspended process, to provide it with service rather than increasing the total load on the system.

ISBN 1-256-52023-3



(a) With one suspend state



(b) With two suspend states

Figure 3.9 Process State Transition Diagram with Suspend States

But this line of reasoning presents a difficulty. All of the processes that have been suspended were in the Blocked state at the time of suspension. It clearly would not do any good to bring a blocked process back into main memory, because it is still not ready for execution. Recognize, however, that each process in the Suspend state was originally blocked on a particular event. When that event occurs, the process is not blocked and is potentially available for execution.

Therefore, we need to rethink this aspect of the design. There are two independent concepts here: whether a process is waiting on an event (blocked or not) and whether a process has been swapped out of main memory (suspended or not). To accommodate this 2 × 2 combination, we need four states:

- **Ready:** The process is in main memory and available for execution
- **Blocked:** The process is in main memory and awaiting an event.

ISBN 1-256-52023-3

- **Blocked/Suspend:** The process is in secondary memory and awaiting an event.
- **Ready/Suspend:** The process is in secondary memory but is available for execution as soon as it is loaded into main memory.

Before looking at a state transition diagram that encompasses the two new suspend states, one other point should be mentioned. The discussion so far has assumed that virtual memory is not in use and that a process is either all in main memory or all out of main memory. With a virtual memory scheme, it is possible to execute a process that is only partially in main memory. If reference is made to a process address that is not in main memory, then the appropriate portion of the process can be brought in. The use of virtual memory would appear to eliminate the need for explicit swapping, because any desired address in any desired process can be moved in or out of main memory by the memory management hardware of the processor. However, as we shall see in Chapter 8, the performance of a virtual memory system can collapse if there is a sufficiently large number of active processes, all of which are partially in main memory. Therefore, even in a virtual memory system, the OS will need to swap out processes explicitly and completely from time to time in the interests of performance.

Let us look now, in Figure 3.9b, at the state transition model that we have developed. (The dashed lines in the figure indicate possible but not necessary transitions.) Important new transitions are the following:

- **Blocked → Blocked/Suspend:** If there are no ready processes, then at least one blocked process is swapped out to make room for another process that is not blocked. This transition can be made even if there are ready processes available, if the OS determines that the currently running process or a ready process that it would like to dispatch requires more main memory to maintain adequate performance.
- **Blocked/Suspend → Ready/Suspend:** A process in the Blocked/Suspend state is moved to the Ready/Suspend state when the event for which it has been waiting occurs. Note that this requires that the state information concerning suspended processes must be accessible to the OS.
- **Ready/Suspend → Ready:** When there are no ready processes in main memory, the OS will need to bring one in to continue execution. In addition, it might be the case that a process in the Ready/Suspend state has higher priority than any of the processes in the Ready state. In that case, the OS designer may dictate that it is more important to get at the higher-priority process than to minimize swapping.
- **Ready → Ready/Suspend:** Normally, the OS would prefer to suspend a blocked process rather than a ready one, because the ready process can now be executed, whereas the blocked process is taking up main memory space and cannot be executed. However, it may be necessary to suspend a ready process if that is the only way to free up a sufficiently large block of main memory. Also, the OS may choose to suspend a lower–priority ready process rather than a higher–priority blocked process if it believes that the blocked process will be ready soon.

ISBN 1-256-52023-3

Several other transitions that are worth considering are the following:

- **New → Ready/Suspend and New → Ready:** When a new process is created, it can either be added to the Ready queue or the Ready/Suspend queue. In either case, the OS must create a process control block and allocate an address space to the process. It might be preferable for the OS to perform these housekeeping duties at an early time, so that it can maintain a large pool of processes that are not blocked. With this strategy, there would often be insufficient room in main memory for a new process; hence the use of the (New → Ready/Suspend) transition. On the other hand, we could argue that a just-in-time philosophy of creating processes as late as possible reduces OS overhead and allows that OS to perform the process-creation duties at a time when the system is clogged with blocked processes anyway.
- **Blocked/Suspend → Blocked:** Inclusion of this transition may seem to be poor design. After all, if a process is not ready to execute and is not already in main memory, what is the point of bringing it in? But consider the following scenario: A process terminates, freeing up some main memory. There is a process in the (Blocked/Suspend) queue with a higher priority than any of the processes in the (Ready/Suspend) queue and the OS has reason to believe that the blocking event for that process will occur soon. Under these circumstances, it would seem reasonable to bring a blocked process into main memory in preference to a ready process.
- **Running → Ready/Suspend:** Normally, a running process is moved to the Ready state when its time allocation expires. If, however, the OS is preempting the process because a higher-priority process on the Blocked/Suspend queue has just become unblocked, the OS could move the running process directly to the (Ready/Suspend) queue and free some main memory.
- **Any State → Exit:** Typically, a process terminates while it is running, either because it has completed or because of some fatal fault condition. However, in some operating systems, a process may be terminated by the process that created it or when the parent process is itself terminated. If this is allowed, then a process in any state can be moved to the Exit state.

Other Uses of Suspension So far, we have equated the concept of a suspended process with that of a process that is not in main memory. A process that is not in main memory is not immediately available for execution, whether or not it is awaiting an event.

We can generalize the concept of a suspended process. Let us define a suspended process as having the following characteristics:

1. The process is not immediately available for execution.
2. The process may or may not be waiting on an event. If it is, this blocked condition is independent of the suspend condition, and occurrence of the blocking event does not enable the process to be executed immediately.

ISBN 1-256-52023-3

Table 3.3 Reasons for Process Suspension

Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

3. The process was placed in a suspended state by an agent: either itself, a parent process, or the OS, for the purpose of preventing its execution.
4. The process may not be removed from this state until the agent explicitly orders the removal.

Table 3.3 lists some reasons for the suspension of a process. One reason that we have discussed is to provide memory space either to bring in a Ready/Suspended process or to increase the memory allocated to other Ready processes. The OS may have other motivations for suspending a process. For example, an auditing or tracing process may be employed to monitor activity on the system; the process may be used to record the level of utilization of various resources (processor, memory, channels) and the rate of progress of the user processes in the system. The OS, under operator control, may turn this process on and off from time to time. If the OS detects or suspects a problem, it may suspend a process. One example of this is deadlock, which is discussed in Chapter 6. As another example, a problem is detected on a communications line, and the operator has the OS suspend the process that is using the line while some tests are run.

Another set of reasons concerns the actions of an interactive user. For example, if a user suspects a bug in the program, he or she may debug the program by suspending its execution, examining and modifying the program or data, and resuming execution. Or there may be a background process that is collecting trace or accounting statistics, which the user may wish to be able to turn on and off.

Timing considerations may also lead to a swapping decision. For example, if a process is to be activated periodically but is idle most of the time, then it should be swapped out between uses. A program that monitors utilization or user activity is an example.

Finally, a parent process may wish to suspend a descendent process. For example, process A may spawn process B to perform a file read. Subsequently, process B encounters an error in the file read procedure and reports this to process A. Process A suspends process B to investigate the cause.

In all of these cases, the activation of a suspended process is requested by the agent that initially requested the suspension.

ISBN 1-256-52023-3

3.3 PROCESS DESCRIPTION

The OS controls events within the computer system. It schedules and dispatches processes for execution by the processor, allocates resources to processes, and responds to requests by user processes for basic services. Fundamentally, we can think of the OS as that entity that manages the use of system resources by processes.

This concept is illustrated in Figure 3.10. In a multiprogramming environment, there are a number of processes (P_1,..., P_n) that have been created and exist in virtual memory. Each process, during the course of its execution, needs access to certain system resources, including the processor, I/O devices, and main memory. In the figure, process P_1 is running; at least part of the process is in main memory, and it has control of two I/O devices. Process P_2 is also in main memory but is blocked waiting for an I/O device allocated to P_1. Process P_n has been swapped out and is therefore suspended.

We explore the details of the management of these resources by the OS on behalf of the processes in later chapters. Here we are concerned with a more fundamental question: What information does the OS need to control processes and manage resources for them?

Operating System Control Structures

If the OS is to manage processes and resources, it must have information about the current status of each process and resource. The universal approach to providing this information is straightforward: The OS constructs and maintains tables of information about each entity that it is managing. A general idea of the scope of this effort is indicated in Figure 3.11, which shows four different types of tables maintained by the OS: memory, I/O, file, and process. Although the details will differ from one OS to another, fundamentally, all operating systems maintain information in these four categories.

Memory tables are used to keep track of both main (real) and secondary (virtual) memory. Some of main memory is reserved for use by the OS; the remainder is available for use by processes. Processes are maintained on secondary memory using some sort of virtual memory or simple swapping mechanism. The memory tables must include the following information:

- The allocation of main memory to processes
- The allocation of secondary memory to processes

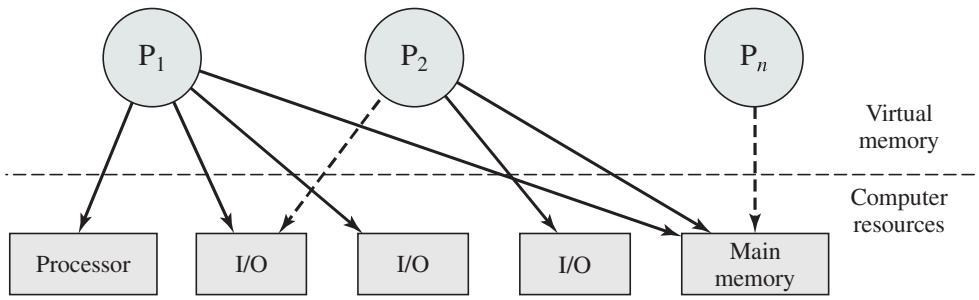


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

ISBN 1-256-52023-3

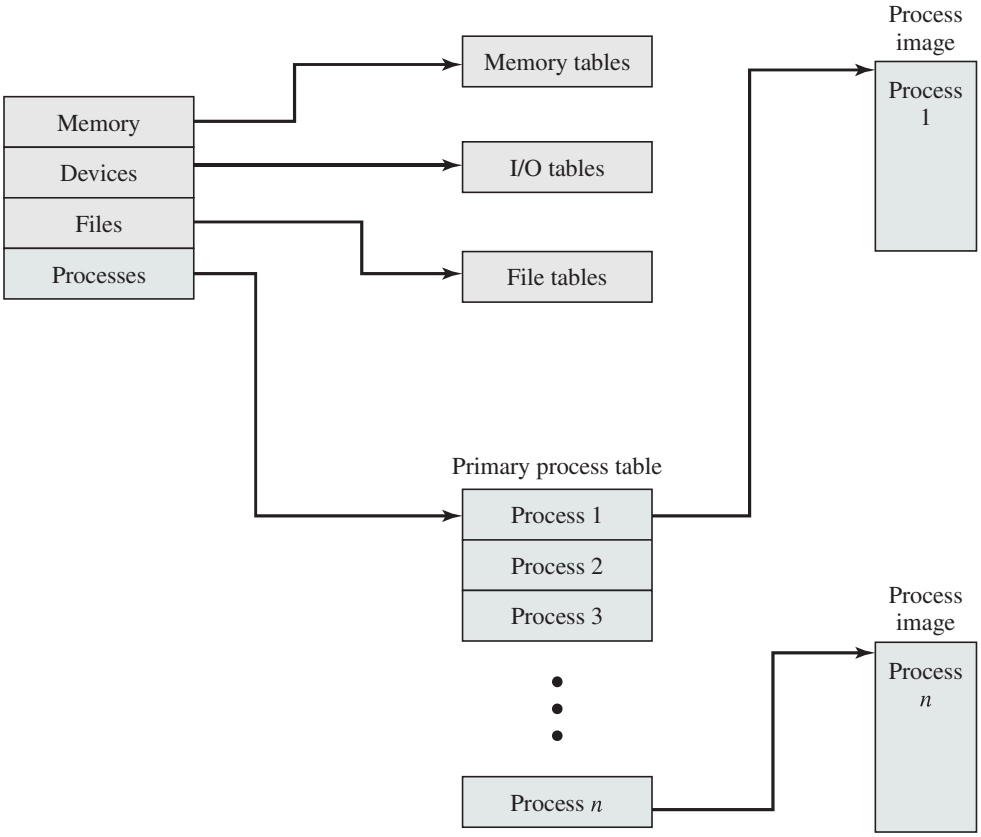


Figure 3.11 General Structure of Operating System Control Tables

- Any protection attributes of blocks of main or virtual memory, such as which processes may access certain shared memory regions
- Any information needed to manage virtual memory

We examine the information structures for memory management in detail in Part Three.

I/O tables are used by the OS to manage the I/O devices and channels of the computer system. At any given time, an I/O device may be available or assigned to a particular process. If an I/O operation is in progress, the OS needs to know the status of the I/O operation and the location in main memory being used as the source or destination of the I/O transfer. I/O management is examined in Chapter 11.

The OS may also maintain **file tables**. These tables provide information about the existence of files, their location on secondary memory, their current status, and other attributes. Much, if not all, of this information may be maintained and used by a file management system, in which case the OS has little or no knowledge of files. In other operating systems, much of the detail of file management is managed by the OS itself. This topic is explored in Chapter 12.

Finally, the OS must maintain **process tables** to manage processes. The remainder of this section is devoted to an examination of the required process tables. Before proceeding to this discussion, two additional points should be made. First, although Figure 3.11 shows four distinct sets of tables, it should be clear that these tables must be linked or cross-referenced in some fashion. Memory, I/O, and

ISBN 1-256-52023-3

files are managed on behalf of processes, so there must be some reference to these resources, directly or indirectly, in the process tables. The files referred to in the file tables are accessible via an I/O device and will, at some times, be in main or virtual memory. The tables themselves must be accessible by the OS and therefore are subject to memory management.

Second, how does the OS know to create the tables in the first place? Clearly, the OS must have some knowledge of the basic environment, such as how much main memory exists, what are the I/O devices and what are their identifiers, and so on. This is an issue of configuration. That is, when the OS is initialized, it must have access to some configuration data that define the basic environment, and these data must be created outside the OS, with human assistance or by some autoconfiguration software.

Process Control Structures

Consider what the OS must know if it is to manage and control a process. First, it must know where the process is located; second, it must know the attributes of the process that are necessary for its management (e.g., process ID and process state).

Process Location Before we can deal with the questions of where a process is located or what its attributes are, we need to address an even more fundamental question: What is the physical manifestation of a process? At a minimum, a process must include a program or set of programs to be executed. Associated with these programs is a set of data locations for local and global variables and any defined constants. Thus, a process will consist of at least sufficient memory to hold the programs and data of that process. In addition, the execution of a program typically involves a stack (see Appendix P) that is used to keep track of procedure calls and parameter passing between procedures. Finally, each process has associated with it a number of attributes that are used by the OS for process control. Typically, the collection of attributes is referred to as a *process control block*.[7] We can refer to this collection of program, data, stack, and attributes as the **process image** (Table 3.4).

The location of a process image will depend on the memory management scheme being used. In the simplest case, the process image is maintained as a

Table 3.4 Typical Elements of a Process Image

User Data The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.
User Program The program to be executed.
Stack Each process has one or more last-in-first-out (LIFO) stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.
Process Control Block Data needed by the OS to control the process (see Table 3.5).

[7]Other commonly used names for this data structure are *task control block*, *process descriptor*, and *task descriptor*.

ISBN 1-256-52023-3

contiguous, or continuous, block of memory. This block is maintained in secondary memory, usually disk. So that the OS can manage the process, at least a small portion of its image must be maintained in main memory. To execute the process, the entire process image must be loaded into main memory or at least virtual memory. Thus, the OS needs to know the location of each process on disk and, for each such process that is in main memory, the location of that process in main memory. We saw a slightly more complex variation on this scheme with the CTSS OS, in Chapter 2. With CTSS, when a process is swapped out, part of the process image may remain in main memory. Thus, the OS must keep track of which portions of the image of each process are still in main memory.

Modern operating systems presume paging hardware that allows noncontiguous physical memory to support partially resident processes.[8] At any given time, a portion of a process image may be in main memory, with the remainder in secondary memory.[9] Therefore, process tables maintained by the OS must show the location of each page of each process image.

Figure 3.11 depicts the structure of the location information in the following way. There is a primary process table with one entry for each process. Each entry contains, at least, a pointer to a process image. If the process image contains multiple blocks, this information is contained directly in the primary process table or is available by cross-reference to entries in memory tables. Of course, this depiction is generic; a particular OS will have its own way of organizing the location information.

PROCESS ATTRIBUTES A sophisticated multiprogramming system requires a great deal of information about each process. As was explained, this information can be considered to reside in a process control block. Different systems will organize this information in different ways, and several examples of this appear at the end of this chapter and the next. For now, let us simply explore the type of information that might be of use to an OS without considering in any detail how that information is organized.

Table 3.5 lists the typical categories of information required by the OS for each process. You may be somewhat surprised at the quantity of information required. As you gain a greater appreciation of the responsibilities of the OS, this list should appear more reasonable.

We can group the process control block information into three general categories:

- Process identification
- Processor state information
- Process control information

[8]A brief overview of the concepts of pages, segments, and virtual memory is provided in the subsection on memory management in Section 2.3.

[9]This brief discussion slides over some details. In particular, in a system that uses virtual memory, all of the process image for an active process is always in secondary memory. When a portion of the image is loaded into main memory, it is copied rather than moved. Thus, the secondary memory retains a copy of all segments and/or pages. However, if the main memory portion of the image is modified, the secondary copy will be out of date until the main memory portion is copied back onto disk.

ISBN 1-256-52023-3

Table 3.5 Typical Elements of a Process Control Block

Process Identification

Identifiers
Numeric identifiers that may be stored with the process control block include

- Identifier of this process
- Identifier of the process that created this process (parent process)
- User identifier

Processor State Information

User-Visible Registers
A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.

Control and Status Registers
These are a variety of processor registers that are employed to control the operation of the processor. These include

- **Program counter:** Contains the address of the next instruction to be fetched
- **Condition codes:** Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
- **Status information:** Includes interrupt enabled/disabled flags, execution mode

Stack Pointers
Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

Process Control Information

Scheduling and State Information
This is information that is needed by the operating system to perform its scheduling function. Typical items of information:

- **Process state:** Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
- **Priority:** One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable).
- **Scheduling-related information:** This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- **Event:** Identity of event the process is awaiting before it can be resumed.

Data Structuring
A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent–child (creator–created) relationship with another process. The process control block may contain pointers to other processes to support these structures.

Interprocess Communication
Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

Process Privileges
Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

Memory Management
This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

Resource Ownership and Utilization
Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

ISBN 1-256-52023-3

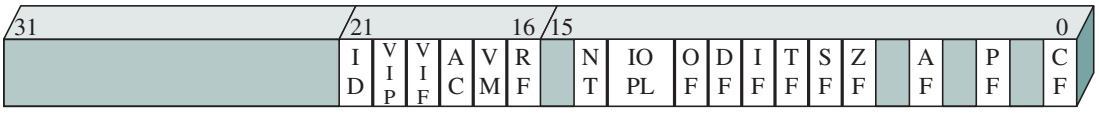
With respect to **process identification**, in virtually all operating systems, each process is assigned a unique numeric identifier, which may simply be an index into the primary process table (Figure 3.11); otherwise there must be a mapping that allows the OS to locate the appropriate tables based on the process identifier. This identifier is useful in several ways. Many of the other tables controlled by the OS may use process identifiers to cross-reference process tables. For example, the memory tables may be organized so as to provide a map of main memory with an indication of which process is assigned to each region. Similar references will appear in I/O and file tables. When processes communicate with one another, the process identifier informs the OS of the destination of a particular communication. When processes are allowed to create other processes, identifiers indicate the parent and descendents of each process.

In addition to these process identifiers, a process may be assigned a user identifier that indicates the user responsible for the job.

Processor state information consists of the contents of processor registers. While a process is running, of course, the information is in the registers. When a process is interrupted, all of this register information must be saved so that it can be restored when the process resumes execution. The nature and number of registers involved depend on the design of the processor. Typically, the register set will include user-visible registers, control and status registers, and stack pointers. These are described in Chapter 1.

Of particular note, all processor designs include a register or set of registers, often known as the program status word (PSW), that contains status information. The PSW typically contain condition codes plus other status information. A good example of a processor status word is that on Intel x86 processors, referred to as the EFLAGS register (shown in Figure 3.12 and Table 3.6). This structure is used by any OS (including UNIX and Windows) running on an x86 processor.

The third major category of information in the process control block can be called, for want of a better name, **process control information**. This is the additional information needed by the OS to control and coordinate the various active processes. The last part of Table 3.5 indicates the scope of this information. As



ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag
DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

Figure 3.12 x86 EFLAGS Register

ISBN 1-256-52023-3

Table 3.6 Pentium EFLAGS Register Bits

Control Bits

AC (Alignment check)
Set if a word or doubleword is addressed on a nonword or non-doubleword boundary.

ID (Identification flag)
If this bit can be set and cleared, this processor supports the CPUID instruction. This instruction provides information about the vendor, family, and model.

RF (Resume flag)
Allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.

IOPL (I/O privilege level)
When set, causes the processor to generate an exception on all accesses to I/O devices during protected mode operation.

DF (Direction flag)
Determines whether string processing instructions increment or decrement the 16-bit half-registers SI and DI (for 16-bit operations) or the 32-bit registers ESI and EDI (for 32-bit operations).

IF (Interrupt enable flag)
When set, the processor will recognize external interrupts.

TF (Trap flag)
When set, causes an interrupt after the execution of each instruction. This is used for debugging.

Operating Mode Bits

NT (Nested task flag)
Indicates that the current task is nested within another task in protected mode operation.

VM (Virtual 8086 mode)
Allows the programmer to enable or disable virtual 8086 mode, which determines whether the processor runs as an 8086 machine.

VIP (Virtual interrupt pending)
Used in virtual 8086 mode to indicate that one or more interrupts are awaiting service.

VIF (Virtual interrupt flag)
Used in virtual 8086 mode instead of IF.

Condition Codes

AF (Auxiliary carry flag)
Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation using the AL register.

CF (Carry flag)
Indicates carrying out or borrowing into the leftmost bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.

OF (Overflow flag)
Indicates an arithmetic overflow after an addition or subtraction.

PF (Parity flag)
Parity of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.

SF (Sign flag)
Indicates the sign of the result of an arithmetic or logic operation.

ZF (Zero flag)
Indicates that the result of an arithmetic or logic operation is 0.

ISBN 1-256-52023-3

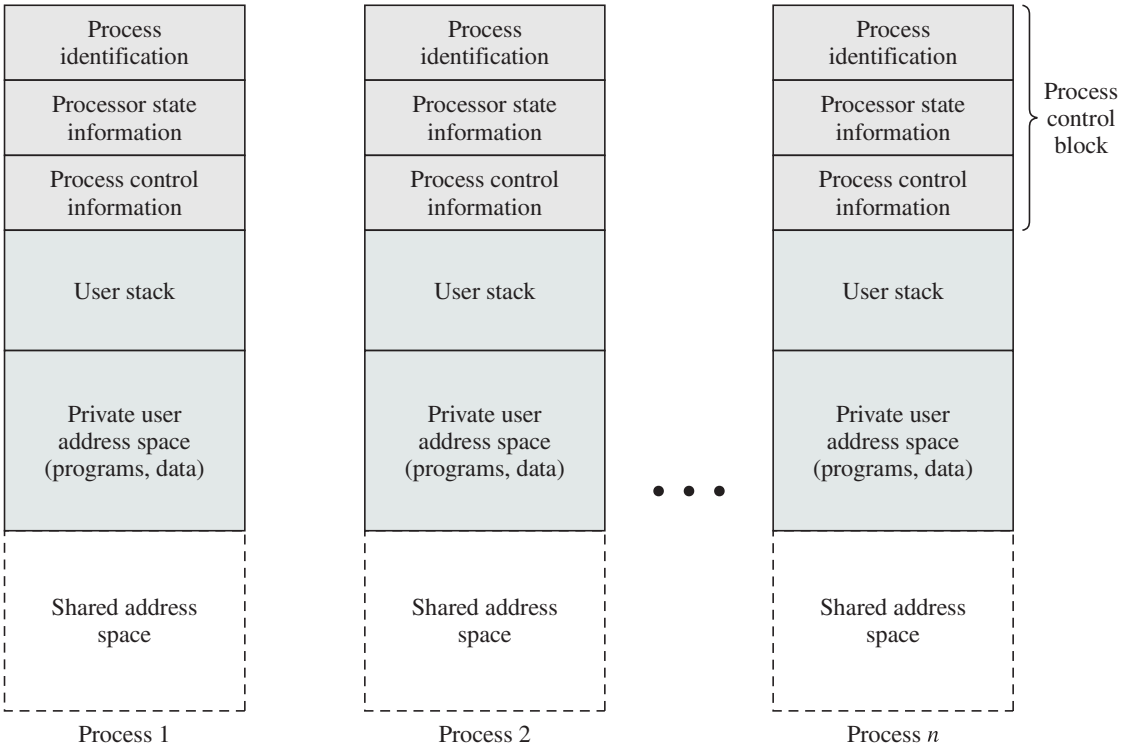


Figure 3.13 **User Processes in Virtual Memory**

we examine the details of operating system functionality in succeeding chapters, the need for the various items on this list should become clear.

Figure 3.13 suggests the structure of process images in virtual memory. Each process image consists of a process control block, a user stack, the private address space of the process, and any other address space that the process shares with other processes. In the figure, each process image appears as a contiguous range of addresses. In an actual implementation, this may not be the case; it will depend on the memory management scheme and the way in which control structures are organized by the OS.

As indicated in Table 3.5, the process control block may contain structuring information, including pointers that allow the linking of process control blocks. Thus, the queues that were described in the preceding section could be implemented as linked lists of process control blocks. For example, the queueing structure of Figure 3.8a could be implemented as suggested in Figure 3.14.

The Role of the Process Control Block The process control block is the most important data structure in an OS. Each process control block contains all of the information about a process that is needed by the OS. The blocks are read and/or modified by virtually every module in the OS, including those involved with scheduling, resource allocation, interrupt processing, and performance monitoring and analysis. One can say that the set of process control blocks defines the state of the OS.

This brings up an important design issue. A number of routines within the OS will need access to information in process control blocks. The provision of direct

ISBN 1-256-52023-3

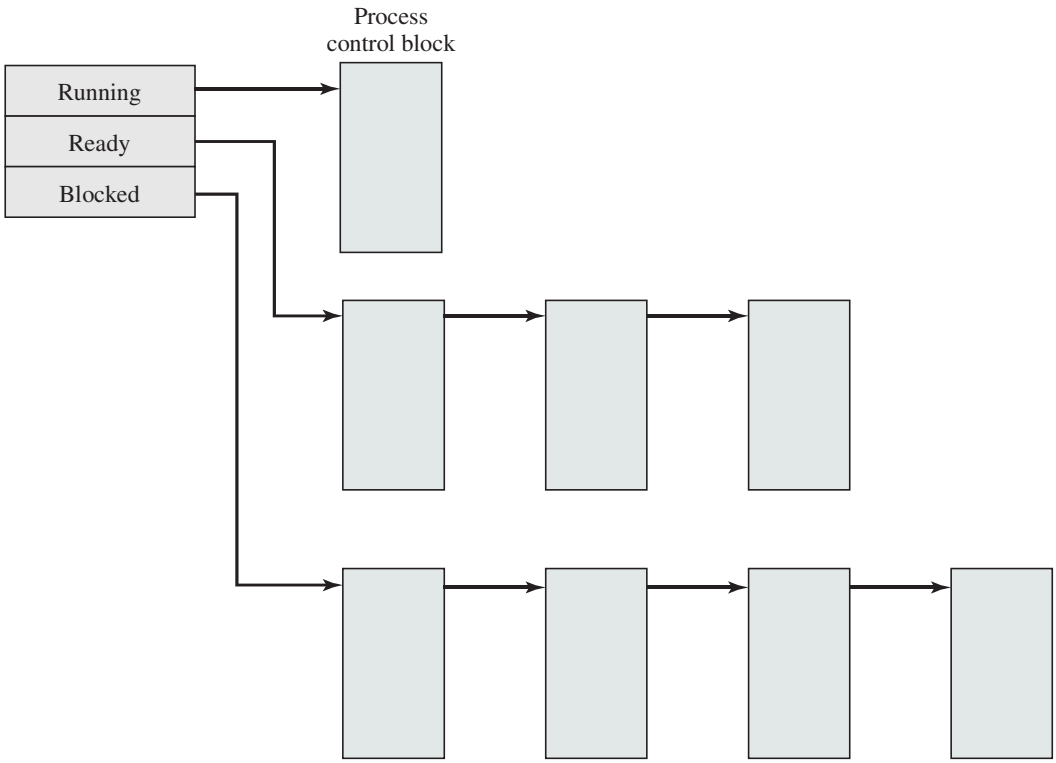


Figure 3.14 **Process List Structures**

access to these tables is not difficult. Each process is equipped with a unique ID, and this can be used as an index into a table of pointers to the process control blocks. The difficulty is not access but rather protection. Two problems present themselves:

- A bug in a single routine, such as an interrupt handler, could damage process control blocks, which could destroy the system's ability to manage the affected processes.
- A design change in the structure or semantics of the process control block could affect a number of modules in the OS.

These problems can be addressed by requiring all routines in the OS to go through a handler routine, the only job of which is to protect process control blocks, and which is the sole arbiter for reading and writing these blocks. The trade-off in the use of such a routine involves performance issues and the degree to which the remainder of the system software can be trusted to be correct.

3.4 PROCESS CONTROL

Modes of Execution

Before continuing with our discussion of the way in which the OS manages processes, we need to distinguish between the mode of processor execution normally associated with the OS and that normally associated with user programs. Most

ISBN 1-256-52023-3

processors support at least two modes of execution. Certain instructions can only be executed in the more-privileged mode. These would include reading or altering a control register, such as the program status word; primitive I/O instructions; and instructions that relate to memory management. In addition, certain regions of memory can only be accessed in the more-privileged mode.

The less-privileged mode is often referred to as the **user mode,** because user programs typically would execute in this mode. The more-privileged mode is referred to as the **system mode, control mode,** or **kernel mode.** This last term refers to the kernel of the OS, which is that portion of the OS that encompasses the important system functions. Table 3.7 lists the functions typically found in the kernel of an OS.

The reason for using two modes should be clear. It is necessary to protect the OS and key operating system tables, such as process control blocks, from interference by user programs. In the kernel mode, the software has complete control of the processor and all its instructions, registers, and memory. This level of control is not necessary and for safety is not desirable for user programs.

Two questions arise: How does the processor know in which mode it is to be executing and how is the mode changed? Regarding the first question, typically there is a bit in the program status word (PSW) that indicates the mode of execution. This bit is changed in response to certain events. Typically, when a user makes a call to an operating system service or when an interrupt triggers execution of an operating system routine, the mode is set to the kernel mode and, upon return from the service to the user process, the mode is set to user mode. As an example, consider the Intel Itanium processor, which implements the 64-bit IA-64 architecture. The processor has a processor status register (psr) that includes a 2-bit cpl (current privilege level) field. Level 0 is the most privileged level, while level 3 is the least privileged level. Most operating systems, such as Linux, use level 0 for the kernel and one other level

Table 3.7 Typical Functions of an Operating System Kernel

Process Management
• Process creation and termination
• Process scheduling and dispatching
• Process switching
• Process synchronization and support for interprocess communication
• Management of process control blocks
Memory Management
• Allocation of address space to processes
• Swapping
• Page and segment management
I/O Management
• Buffer management
• Allocation of I/O channels and devices to processes
Support Functions
• Interrupt handling
• Accounting
• Monitoring

ISBN 1-256-52023-3

for user mode. When an interrupt occurs, the processor clears most of the bits in the psr, including the cpl field. This automatically sets the cpl to level 0. At the end of the interrupt-handling routine, the final instruction that is executed is irt (interrupt return). This instruction causes the processor to restore the psr of the interrupted program, which restores the privilege level of that program. A similar sequence occurs when an application places a system call. For the Itanium, an application places a system call by placing the system call identifier and the system call arguments in a predefined area and then executing a special instruction that has the effect of interrupting execution at the user level and transferring control to the kernel.

Process Creation

In Section 3.2, we discussed the events that lead to the creation of a new process. Having discussed the data structures associated with a process, we are now in a position to describe briefly the steps involved in actually creating the process.

Once the OS decides, for whatever reason (Table 3.1), to create a new process, it can proceed as follows:

1. **Assign a unique process identifier to the new process.** At this time, a new entry is added to the primary process table, which contains one entry per process.
2. **Allocate space for the process.** This includes all elements of the process image. Thus, the OS must know how much space is needed for the private user address space (programs and data) and the user stack. These values can be assigned by default based on the type of process, or they can be set based on user request at job creation time. If a process is spawned by another process, the parent process can pass the needed values to the OS as part of the process-creation request. If any existing address space is to be shared by this new process, the appropriate linkages must be set up. Finally, space for a process control block must be allocated.
3. **Initialize the process control block.** The process identification portion contains the ID of this process plus other appropriate IDs, such as that of the parent process. The processor state information portion will typically be initialized with most entries zero, except for the program counter (set to the program entry point) and system stack pointers (set to define the process stack boundaries). The process control information portion is initialized based on standard default values plus attributes that have been requested for this process. For example, the process state would typically be initialized to Ready or Ready/Suspend. The priority may be set by default to the lowest priority unless an explicit request is made for a higher priority. Initially, the process may own no resources (I/O devices, files) unless there is an explicit request for these or unless they are inherited from the parent.
4. **Set the appropriate linkages.** For example, if the OS maintains each scheduling queue as a linked list, then the new process must be put in the Ready or Ready/Suspend list.
5. **Create or expand other data structures.** For example, the OS may maintain an accounting file on each process to be used subsequently for billing and/or performance assessment purposes.

ISBN 1-256-52023-3

Process Switching

On the face of it, the function of process switching would seem to be straightforward. At some time, a running process is interrupted and the OS assigns another process to the Running state and turns control over to that process. However, several design issues are raised. First, what events trigger a process switch? Another issue is that we must recognize the distinction between mode switching and process switching. Finally, what must the OS do to the various data structures under its control to achieve a process switch?

When to Switch Processes A process switch may occur any time that the OS has gained control from the currently running process. Table 3.8 suggests the possible events that may give control to the OS.

First, let us consider system interrupts. Actually, we can distinguish, as many systems do, two kinds of system interrupts, one of which is simply referred to as an interrupt, and the other as a trap. The former is due to some sort of event that is external to and independent of the currently running process, such as the completion of an I/O operation. The latter relates to an error or exception condition generated within the currently running process, such as an illegal file access attempt. With an ordinary **interrupt,** control is first transferred to an interrupt handler, which does some basic housekeeping and then branches to an OS routine that is concerned with the particular type of interrupt that has occurred. Examples include the following:

- **Clock interrupt:** The OS determines whether the currently running process has been executing for the maximum allowable unit of time, referred to as a **time slice**. That is, a time slice is the maximum amount of time that a process can execute before being interrupted. If so, this process must be switched to a Ready state and another process dispatched.
- **I/O interrupt:** The OS determines what I/O action has occurred. If the I/O action constitutes an event for which one or more processes are waiting, then the OS moves all of the corresponding blocked processes to the Ready state (and Blocked/Suspend processes to the Ready/Suspend state). The OS must then decide whether to resume execution of the process currently in the Running state or to preempt that process for a higher-priority Ready process.
- **Memory fault:** The processor encounters a virtual memory address reference for a word that is not in main memory. The OS must bring in the block

Table 3.8 Mechanisms for Interrupting the Execution of a Process

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

ISBN 1-256-52023-3

(page or segment) of memory containing the reference from secondary memory to main memory. After the I/O request is issued to bring in the block of memory, the process with the memory fault is placed in a blocked state; the OS then performs a process switch to resume execution of another process. After the desired block is brought into memory, that process is placed in the Ready state.

With a **trap,** the OS determines if the error or exception condition is fatal. If so, then the currently running process is moved to the Exit state and a process switch occurs. If not, then the action of the OS will depend on the nature of the error and the design of the OS. It may attempt some recovery procedure or simply notify the user. It may do a process switch or resume the currently running process.

Finally, the OS may be activated by a **supervisor call** from the program being executed. For example, a user process is running and an instruction is executed that requests an I/O operation, such as a file open. This call results in a transfer to a routine that is part of the operating system code. The use of a system call may place the user process in the Blocked state.

MODE SWITCHING In Chapter 1, we discussed the inclusion of an interrupt stage as part of the instruction cycle. Recall that, in the interrupt stage, the processor checks to see if any interrupts are pending, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch stage and fetches the next instruction of the current program in the current process. If an interrupt is pending, the processor does the following:

1. It sets the program counter to the starting address of an interrupt handler program.
2. It switches from user mode to kernel mode so that the interrupt processing code may include privileged instructions.

The processor now proceeds to the fetch stage and fetches the first instruction of the interrupt handler program, which will service the interrupt. At this point, typically, the context of the process that has been interrupted is saved into that process control block of the interrupted program.

One question that may now occur to you is, What constitutes the context that is saved? The answer is that it must include any information that may be altered by the execution of the interrupt handler and that will be needed to resume the program that was interrupted. Thus, the portion of the process control block that was referred to as processor state information must be saved. This includes the program counter, other processor registers, and stack information.

Does anything else need to be done? That depends on what happens next. The interrupt handler is typically a short program that performs a few basic tasks related to an interrupt. For example, it resets the flag or indicator that signals the presence of an interrupt. It may send an acknowledgment to the entity that issued the interrupt, such as an I/O module. And it may do some basic housekeeping relating to the effects of the event that caused the interrupt. For example, if the interrupt relates to an I/O event, the interrupt handler will check for an error condition. If an error

ISBN 1-256-52023-3

has occurred, the interrupt handler may send a signal to the process that originally requested the I/O operation. If the interrupt is by the clock, then the handler will hand control over to the dispatcher, which will want to pass control to another process because the time slice allotted to the currently running process has expired.

What about the other information in the process control block? If this interrupt is to be followed by a switch to another process, then some work will need to be done. However, in most operating systems, the occurrence of an interrupt does not necessarily mean a process switch. It is possible that, after the interrupt handler has executed, the currently running process will resume execution. In that case, all that is necessary is to save the processor state information when the interrupt occurs and restore that information when control is returned to the program that was running. Typically, the saving and restoring functions are performed in hardware.

CHANGE OF PROCESS STATE It is clear, then, that the mode switch is a concept distinct from that of the process switch.[10] A mode switch may occur without changing the state of the process that is currently in the Running state. In that case, the context saving and subsequent restoral involve little overhead. However, if the currently running process is to be moved to another state (Ready, Blocked, etc.), then the OS must make substantial changes in its environment. The steps involved in a full process switch are as follows:

1. Save the context of the processor, including program counter and other registers.
2. Update the process control block of the process that is currently in the Running state. This includes changing the state of the process to one of the other states (Ready; Blocked; Ready/Suspend; or Exit). Other relevant fields must also be updated, including the reason for leaving the Running state and accounting information.
3. Move the process control block of this process to the appropriate queue (Ready; Blocked on Event *i*; Ready/Suspend).
4. Select another process for execution; this topic is explored in Part Four.
5. Update the process control block of the process selected. This includes changing the state of this process to Running.
6. Update memory management data structures. This may be required, depending on how address translation is managed; this topic is explored in Part Three.
7. Restore the context of the processor to that which existed at the time the selected process was last switched out of the Running state, by loading in the previous values of the program counter and other registers.

Thus, the process switch, which involves a state change, requires more effort than a mode switch.

[10]The term *context switch* is often found in OS literature and textbooks. Unfortunately, although most of the literature uses this term to mean what is here called a process switch, other sources use it to mean a mode switch or even a thread switch (defined in the next chapter). To avoid ambiguity, the term is not used in this book.

ISBN 1-256-52023-3

3.5 EXECUTION OF THE OPERATING SYSTEM

In Chapter 2, we pointed out two intriguing facts about operating systems:

- The OS functions in the same way as ordinary computer software in the sense that the OS is a set of programs executed by the processor.
- The OS frequently relinquishes control and depends on the processor to restore control to the OS.

If the OS is just a collection of programs and if it is executed by the processor just like any other program, is the OS a process? If so, how is it controlled? These interesting questions have inspired a number of design approaches. Figure 3.15 illustrates a range of approaches that are found in various contemporary operating systems.

Nonprocess Kernel

One traditional approach, common on many older operating systems, is to execute the kernel of the OS outside of any process (Figure 3.15a). With this approach, when the currently running process is interrupted or issues a supervisor call, the mode context of this process is saved and control is passed to the kernel. The OS has its own region of memory to use and its own system stack for controlling procedure calls and returns. The OS can perform any desired functions and restore the context of the interrupted process, which causes execution to resume in the interrupted

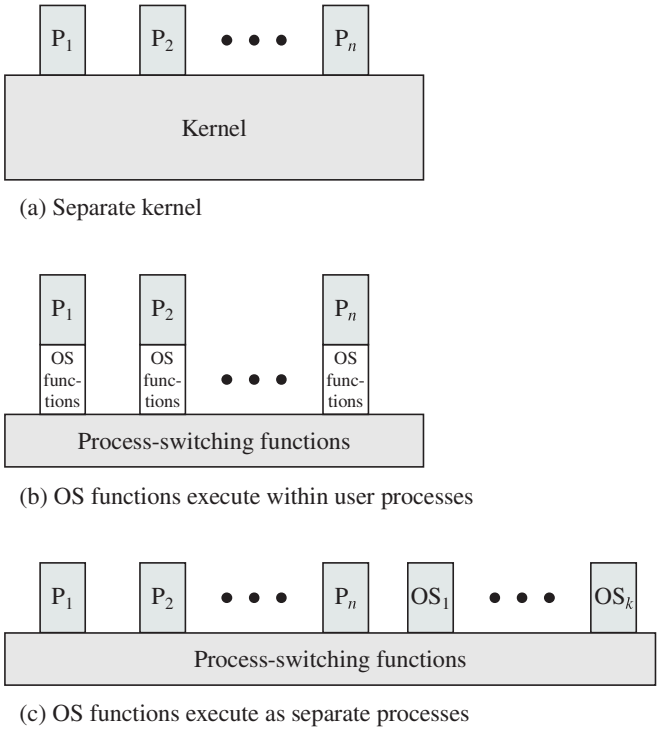


Figure 3.15 Relationship between Operating System and User Processes

ISBN 1-256-52023-3

user process. Alternatively, the OS can complete the function of saving the environment of the process and proceed to schedule and dispatch another process. Whether this happens depends on the reason for the interruption and the circumstances at the time.

In any case, the key point here is that the concept of process is considered to apply only to user programs. The operating system code is executed as a separate entity that operates in privileged mode.

Execution within User Processes

An alternative that is common with operating systems on smaller computers (PCs, workstations) is to execute virtually all OS software in the context of a user process. The view is that the OS is primarily a collection of routines that the user calls to perform various functions, executed within the environment of the user's process. This is illustrated in Figure 3.15b. At any given point, the OS is managing *n* process images. Each image includes not only the regions illustrated in Figure 3.13, but also program, data, and stack areas for kernel programs.

Figure 3.16 suggests a typical process image structure for this strategy. A separate kernel stack is used to manage calls/returns while the process is in kernel mode.

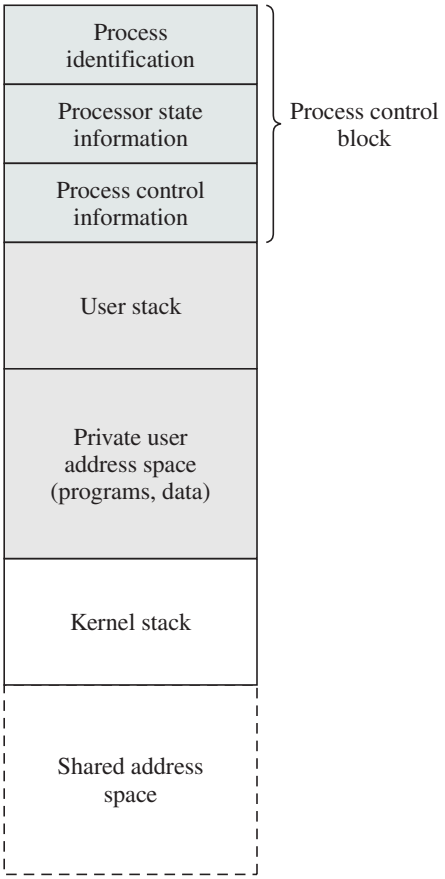


Figure 3.16 **Process Image: Operating System Executes within User Space**

ISBN 1-256-52023-3

Operating system code and data are in the shared address space and are shared by all user processes.

When an interrupt, trap, or supervisor call occurs, the processor is placed in kernel mode and control is passed to the OS. To pass control from a user program to the OS, the mode context is saved and a mode switch takes place to an operating system routine. However, execution continues within the current user process. Thus, a process switch is not performed, just a mode switch within the same process.

If the OS, upon completion of its work, determines that the current process should continue to run, then a mode switch resumes the interrupted program within the current process. This is one of the key advantages of this approach: A user program has been interrupted to employ some operating system routine, and then resumed, and all of this has occurred without incurring the penalty of two process switches. If, however, it is determined that a process switch is to occur rather than returning to the previously executing program, then control is passed to a process-switching routine. This routine may or may not execute in the current process, depending on system design. At some point, however, the current process has to be placed in a nonrunning state and another process designated as the running process. During this phase, it is logically most convenient to view execution as taking place outside of all processes.

In a way, this view of the OS is remarkable. Simply put, at certain points in time, a process will save its state information, choose another process to run from among those that are ready, and relinquish control to that process. The reason this is not an arbitrary and indeed chaotic situation is that during the critical time, the code that is executed in the user process is shared operating system code and not user code. Because of the concept of user mode and kernel mode, the user cannot tamper with or interfere with the operating system routines, even though they are executing in the user's process environment. This further reminds us that there is a distinction between the concepts of process and program and that the relationship between the two is not one to one. Within a process, both a user program and operating system programs may execute, and the operating system programs that execute in the various user processes are identical.

Process-Based Operating System

Another alternative, illustrated in Figure 3.15c, is to implement the OS as a collection of system processes. As in the other options, the software that is part of the kernel executes in a kernel mode. In this case, however, major kernel functions are organized as separate processes. Again, there may be a small amount of process-switching code that is executed outside of any process.

This approach has several advantages. It imposes a program design discipline that encourages the use of a modular OS with minimal, clean interfaces between the modules. In addition, some noncritical operating system functions are conveniently implemented as separate processes. For example, we mentioned earlier a monitor program that records the level of utilization of various resources (processor, memory, channels) and the rate of progress of the user processes in the system. Because this program does not provide a particular service to any active process, it can only be invoked by the OS. As a process, the function can run at an assigned priority

ISBN 1-256-52023-3

level and be interleaved with other processes under dispatcher control. Finally, implementing the OS as a set of processes is useful in a multiprocessor or multicomputer environment, in which some of the operating system services can be shipped out to dedicated processors, improving performance.

3.6 SECURITY ISSUES

An OS associates a set of privileges with each process. These privileges dictate what resources the process may access, including regions of memory, files, privileged system instructions, and so on. Typically, a process that executes on behalf of a user has the privileges that the OS recognizes for that user. A system or utility process may have privileges assigned at configuration time.

On a typical system, the highest level of privilege is referred to as administrator, supervisor, or root access.[11] Root access provides access to all the functions and services of the operating system. With root access, a process has complete control of the system and can add or change programs and files, monitor other processes, send and receive network traffic, and alter privileges.

A key security issue in the design of any OS is to prevent, or at least detect, attempts by a user or a piece of malicious software (malware) from gaining unauthorized privileges on the system and, in particular, from gaining root access. In this section, we briefly summarize the threats and countermeasures related to this security issue. Part Seven provides more detail.

System Access Threats

System access threats fall into two general categories: intruders and malicious software.

INTRUDERS One of the most common threats to security is the intruder (the other is viruses), often referred to as a hacker or cracker. In an important early study of intrusion, Anderson [ANDE80] identified three classes of intruders:

- **Masquerader:** An individual who is not authorized to use the computer and who penetrates a system's access controls to exploit a legitimate user's account
- **Misfeasor:** A legitimate user who accesses data, programs, or resources for which such access is not authorized, or who is authorized for such access but misuses his or her privileges
- **Clandestine user:** An individual who seizes supervisory control of the system and uses this control to evade auditing and access controls or to suppress audit collection

The masquerader is likely to be an outsider; the misfeasor generally is an insider; and the clandestine user can be either an outsider or an insider.

Intruder attacks range from the benign to the serious. At the benign end of the scale, there are many people who simply wish to explore internets and see what is

[11] On UNIX systems, the administrator, or *superuser*, account is called root; hence the term *root access*.

ISBN 1-256-52023-3