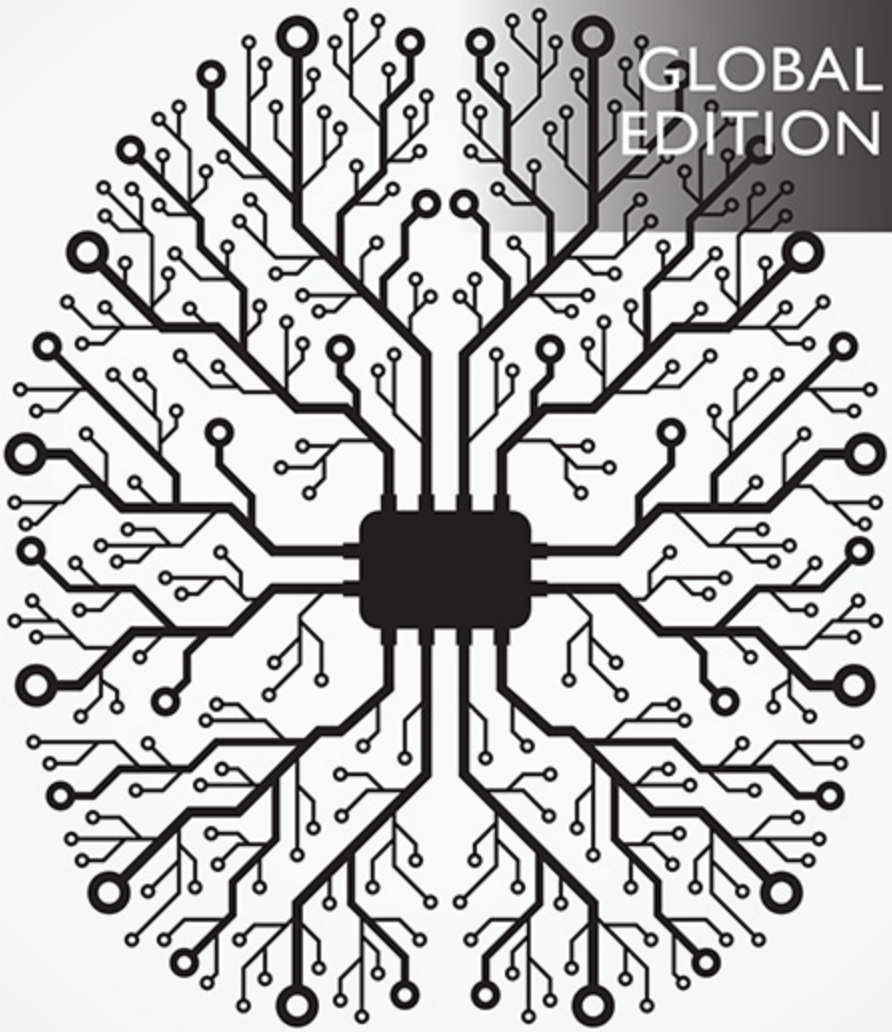
GLOBAL EDITION



Operating Systems

Internals and Design Principles

NINTH EDITION

William Stallings

Pearson

Operating Systems

This page intentionally left blank

Operating Systems

Internals and Design Principles

Ninth Edition

Global Edition

William Stallings

Pearson

Senior Vice President Courseware Portfolio Management: Marcia J. Horton
Director, Portfolio Management: Engineering, Computer Science & Global Editions: Julian Partridge
Higher Ed Portfolio Management: Tracy Johnson (Dunkelberger)
Acquisitions Editor, Global Editions: Sourabh Maheshwari
Portfolio Management Assistant: Kristy Alaura
Managing Content Producer: Scott Disanno
Content Producer: Robert Engelhardt
Project Editor, Global Editions: K.K. Neelakantan
Web Developer: Steve Wright
Rights and Permissions Manager: Ben Ferrini
Manufacturing Buyer, Higher Ed, Lake Side Communications Inc (LSC): Maura Zaldivar-Garcia
Senior Manufacturing Controller, Global Editions: Trudy Kimber
Media Production Manager, Global Editions: Vikram Kumar
Inventory Manager: Ann Lam
Marketing Manager: Demetrius Hall
Product Marketing Manager: Yvonne Vannatta
Marketing Assistant: Jon Bryant
Cover Designer: Lumina Datamatics
Cover Art: Shai_Halud/Shutterstock
Full-Service Project Management: Bhanuprakash Sherla, SPi Global

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on page CL-1.

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

Authorized adaptation from the United States edition, entitled Operating Systems: Internals and Design Principles, 9th Edition, ISBN 978-0-13-467095-9, by William Stallings published by Pearson Education © 2018.

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1

ISBN 10: 1-292-21429-5
ISBN 13: 978-1-292-21429-0

Typeset by SPi Global

Printed and bound in Malaysia.

For Tricia

This page intentionally left blank

CONTENTS

This page intentionally left blank

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.

VIDEONOTES

Locations of VideoNotes

http://www.pearsonglobaleditions.com/stallings

This page intentionally left blank

PREFACE

WHAT'S NEW IN THE NINTH EDITION

Since the eighth edition of this book was published, the field of operating systems has seen continuous innovations and improvements. In this new edition, I have tried to capture these changes while maintaining a comprehensive coverage of the entire field. To begin the process of revision, the eighth 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.

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:

- **Updated Linux coverage:** The Linux material has been updated and expanded to reflect changes in the Linux kernel since the eighth edition.
- **Updated Android coverage:** The Android material has been updated and expanded to reflect changes in the Android kernel since the eighth edition.
- **New Virtualization coverage:** The chapter on virtual machines has been completely rewritten to provide better organization and an expanded and more up-to-date treatment. In addition, a new section has been added on the use of containers.
- **New Cloud operating systems:** New to this edition is the coverage of cloud operating systems, including an overview of cloud computing, a discussion of the principles and requirements for a cloud operating system, and a discussion of a OpenStack, a popular open-source Cloud OS.
- **New IoT operating systems:** New to this edition is the coverage of operating systems for the Internet of Things. The coverage includes an overview of the IoT, a discussion of the principles and requirements for an IoT operating system, and a discussion of a RIOT, a popular open-source IoT OS.
- **Updated and Expanded Embedded operating systems:** This chapter has been substantially revised and expanded including:
  - The section on embedded systems has been expanded and now includes discussions of microcontrollers and deeply embedded systems.
  - The overview section on embedded OSs has been expanded and updated.
  - The treatment of embedded Linux has been expanded, and a new discussion of a popular embedded Linux system, μClinux, has been added.
- **Concurrency:** New projects have been added to the Projects Manual to better help the student understand the principles of concurrency.

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 confined to the capacity and speed of machines, but in applications and system support requirements. Second, the rapid pace of change that has always characterized computer systems continues without respite. 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, four operating systems have been chosen as running examples:

- **Windows:** A multitasking operating system for personal computers, workstations, servers, and mobile devices. 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 object-oriented design principles. This book covers the technology used in the most recent version of Windows, known as Windows 10.
- **Android:** Android is tailored for embedded devices, especially mobile phones. Focusing on the unique requirements of the embedded environment, the book provides details of Android internals.
- **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 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. For convenience, all of the material for each of the example systems is also available as an online document.

SUPPORT OF ACM/IEEE COMPUTER SCIENCE CURRICULA 2013

The book is intended for both an academic and a professional audience. As a textbook, it is intended as a one-semester or two-semester undergraduate course in operating systems for computer science, computer engineering, and electrical engineering majors. This edition is designed to support the recommendations of the current (December 2013) version of the ACM/IEEE Computer Science Curricula 2013 (CS2013). The CS2013 curriculum recommendation includes Operating Systems (OS) as one of the Knowledge Areas in the Computer Science Body of Knowledge. CS2013 divides all course work into three categories: Core-Tier 1 (all topics should be included in the curriculum), Core-Tier 2 (all or almost all topics should be included), and Elective (desirable to provide breadth and depth). In the OS area, CS2013 includes two Tier 1 topics, four Tier 2 topics, and six Elective topics, each of which has a number of subtopics. This text covers all of the topics and subtopics listed by CS2013 in these three categories.

Table P.1 shows the support for the OS Knowledge Areas provided in this textbook. A detailed list of subtopics for each topic is available as the file CS2013-OS .pdf at box.com/OS9e.

PLAN OF THE TEXT

The book is divided into six parts:

1. Background
2. Processes
3. Memory
4. Scheduling
5. Input/Output and files
6. Advanced topics (embedded OSs, virtual machines, OS security, and cloud and IoT operating systems)

The book includes a number of pedagogic features, including the use of animations and videonotes and numerous figures and tables to clarify the discussion. Each chapter includes a list of key words, review questions, and homework problems. 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.

Table P.1 Coverage of CS2013 Operating Systems (OSs) Knowledge Area

Topic	Coverage in Book
Overview of Operating Systems (Tier 1)	Chapter 2: Operating System Overview
Operating System Principles (Tier 1)	Chapter 1: Computer System Overview Chapter 2: Operating System Overview
Concurrency (Tier 2)	Chapter 5: Mutual Exclusion and Synchronization Chapter 6: Deadlock and Starvation Appendix A: Topics in Concurrency Chapter 18: Distributed Process Management
Scheduling and Dispatch (Tier 2)	Chapter 9: Uniprocessor Scheduling Chapter 10: Multiprocessor and Real-Time Scheduling
Memory Management (Tier 2)	Chapter 7: Memory Management Chapter 8: Virtual Memory
Security and Protection (Tier 2)	Chapter 15: Operating System Security
Virtual Machines (Elective)	Chapter 14: Virtual Machines
Device Management (Elective)	Chapter 11: I/O Management and Disk Scheduling
File System (Elective)	Chapter 12: File Management
Real Time and Embedded Systems (Elective)	Chapter 10: Multiprocessor and Real-Time Scheduling Chapter 13: Embedded Operating Systems Material on Android throughout the text
Fault Tolerance (Elective)	Section 2.5: Fault Tolerance
System Performance Evaluation (Elective)	Performance issues related to memory management, scheduling, and other areas addressed throughout the text

INSTRUCTOR SUPPORT MATERIALS

The major goal of this text is to make it as effective a teaching tool as possible for this fundamental yet evolving subject. This goal is reflected both in the structure of the book and in the supporting material. The text is accompanied by the following supplementary material to aid the instructor:

- **Solutions manual:** Solutions to end-of-chapter Review Questions and Problems.
- **Projects manual:** Suggested project assignments for all of the project categories listed in this Preface.
- **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 with a separate file of answers.

- **VideoNotes on concurrency:** Professors perennially cite concurrency as perhaps the most difficult concept in the field of operating systems for students to grasp. The edition is accompanied by a number of VideoNotes lectures discussing the various concurrency algorithms defined in the book. This icon appears next to each algorithm definition in the book to indicate that a VideoNote is available:
- **Sample syllabuses:** The text contains more material that can be conveniently covered in one semester. Accordingly, instructors are provided with several sample syllabuses that guide the use of the text within limited time. These samples are based on real-world experience by professors with the seventh edition.

All of these support materials are available at the **Instructor Resource Center (IRC)** for this textbook, which can be reached through the publisher's website http://www.pearsonglobaleditions.com/stallings. To gain access to the IRC, please contact your local Pearson sales representative.

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 has incorporated a projects component in the course as a result of an overwhelming support it received. In the online portion of the text, two major programming projects are defined. In addition, the instructor's support materials available through Pearson not only includes 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 later.
- **Simulation projects:** Described later.
- **Semaphore projects:** Designed to help students understand concurrency concepts, including race conditions, starvation, and deadlock.
- **Kernel projects:** The IRC includes complete instructor support for two different sets of Linux kernel programming projects, as well as a set of kernel programming projects for Android.
- **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

This edition provides 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 full-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.

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 distributed to students to help them begin using OS/161.
3. A set of exercises using OS/161, to be distributed 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 the student can read the textbook material then do the corresponding OS/161 project.

SIMULATIONS

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 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

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.

PROGRAMMING PROJECTS

This edition provides support for 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 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.

Finally, the project manual provided at the IRC includes a series of programming projects that cover a broad range of topics and that can be implemented in any suitable language on any platform.

ONLINE DOCUMENTS AND VIDEONOTES FOR STUDENTS

For this new edition, a substantial amount of original supporting material for students has been made available online, at two online locations. The **book's website**, at http://www.pearsonglobaleditions.com/stallings (click on *Student Resources* link), includes a list of relevant links organized by chapter and an errata sheet for the book.

Purchasing this textbook new also grants the reader twelve months of access to the **Companion Website**, which includes the following materials:

- **Online chapters:** To limit the size and cost of the book, 5 chapters of the book, covering security, 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 15 online appendices cover these topics for the interested student. The appendices are listed in this book's table of contents.
- **Homework problems and solutions:** To aid the student in understanding the material, a separate set of homework problems with solutions is available.

- **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.
- **VideoNotes:** VideoNotes are step-by-step video tutorials specifically designed to enhance the programming concepts presented in this textbook. The book is accompanied by a number of VideoNotes lectures discussing the various concurrency algorithms defined in the book.

To access the Premium Content site, click on the Companion website link at www.pearsonglobaleditions.com/stallings and enter the student access code found on the card in the front of the book.

ACKNOWLEDGMENTS

I would like to thank the following for their contributions. Rami Rosen contributed most of the new material on Linux. Vineet Chadha made a major contribution to the new chapter on virtual machines. Durgadoss Ramanathan provided the new material on Android ART.

Through its multiple editions this book has benefited from review by hundreds of instructors and professionals, who generously spared their precious time and shared their expertise. Here I acknowledge those whose help contributed to this latest edition.

The following instructors reviewed all or a large part of the manuscript for this edition: Jiang Guo (California State University, Los Angeles), Euripides Montagne (University of Central Florida), Kihong Park (Purdue University), Mohammad Abdus Salam (Southern University and A&M College), Robert Marmorstein (Longwood University), Christopher Diaz (Seton Hill University), and Barbara Bracken (Wilkes University).

Thanks also to all those who provided detailed technical reviews of one or more chapters: Nischay Anikar, Adri Jovin, Ron Munitz, Fatih Eyup Nar, Atte Peltomaki, Durgadoss Ramanathan, Carlos Villavieja, Wei Wang, Serban Constantinescu and Chen Yang.

Thanks also to those who provided detailed reviews of the example systems. Reviews of the Android material were provided by Kristopher Micinski, Ron Munitz, Atte Peltomaki, Durgadoss Ramanathan, Manish Shakya, Samuel Simon, Wei Wang, and Chen Yang. The Linux reviewers were Tigran Aivazian, Kaiwan Billimoria, Peter Huewe, Manmohan Manoharan, Rami Rosen, Neha Naik, and Hualing Yu. The Windows material was reviewed by Francisco Cotrina, Sam Haidar, Christopher Kuleci, Benny Olsson, and Dave Probert. The RIOT material was reviewed by Emmanuel Baccelli and Kaspar Schleiser, and OpenStack was reviewed by Bob Callaway. Nick Garnett of eCosCentric reviewed the material on eCos; and Philip Levis, one of the developers of TinyOS reviewed the material on TinyOS. Sid Young reviewed the material on container virtualization.

Andrew Peterson of the University of Toronto prepared the OS/161 supplements for the IRC. James Craig Burley authored and recorded the VideoNotes.

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 thank the many people responsible for the publication of this book, all of whom did their usual excellent job. This includes the staff at Pearson, particularly my editor Tracy Johnson, her assistant Kristy Alaura, program manager Carole Snyder, and project manager Bob Engelhardt. Thanks also to the marketing and sales staffs at Pearson, without whose efforts this book would not be in front of you.

ACKNOWLEDGMENTS FOR THE GLOBAL EDITION

Pearson would like to thank and acknowledge Moumita Mitra Manna (Bangabasi College) for contributing to the Global Edition, and A. Kannamal (Coimbatore Institute of Technology), Kumar Shashi Prabh (Shiv Nadar University), and Khyat Sharma for reviewing the Global Edition.

This page intentionally left blank

About the Author

Dr. William Stallings has authored 18 titles, and including the revised editions, over 40 books on computer security, computer networking, and computer architecture. His writings have appeared in numerous publications, including the *Proceedings of the IEEE, ACM Computing Reviews* and *Cryptologia*.

He has received the Best Computer Science textbook of the Year award 13 times 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 created and maintains the *Computer Science Student Resource Site* at 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 Ph.D. from M.I.T. in Computer Science and a B.S. from Notre Dame in electrical engineering.

This page intentionally left blank

PART 1 Background

CHAPTER 1

COMPUTER SYSTEM OVERVIEW

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 systems and multicore computers.
- 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. Additional topics are covered in Appendix C. For a more detailed treatment, see [STAL16a].

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).
- **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*.

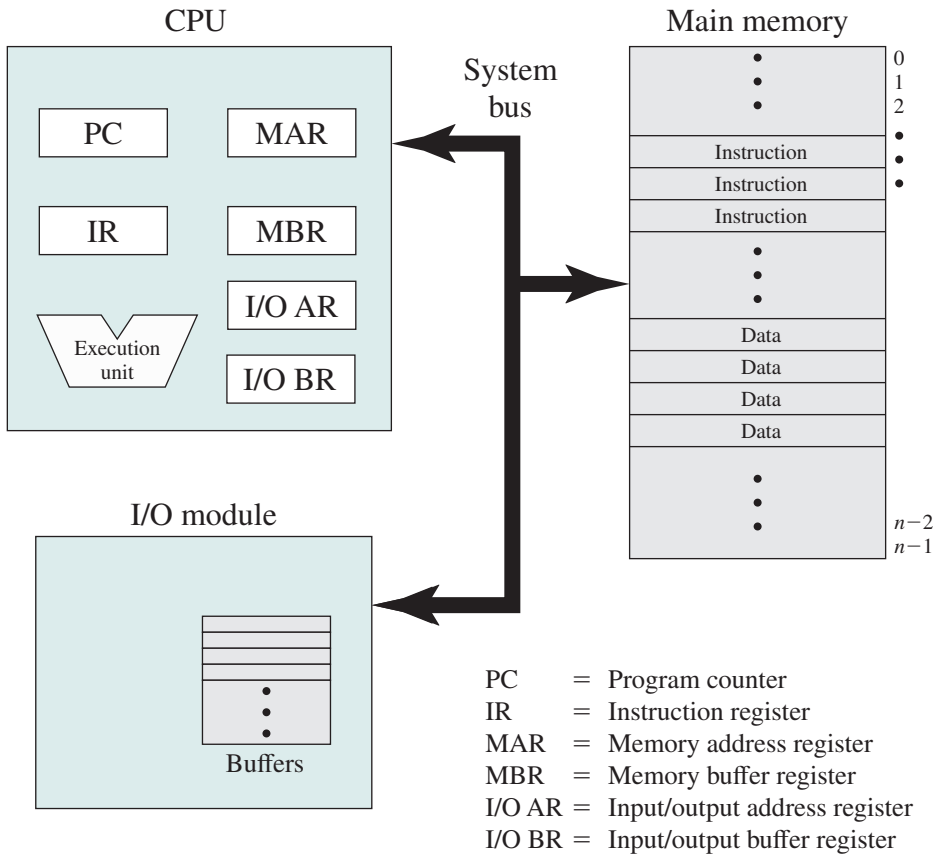


Figure 1.1 **Computer Components: Top-Level View**

- **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 receives 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 storing 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.

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

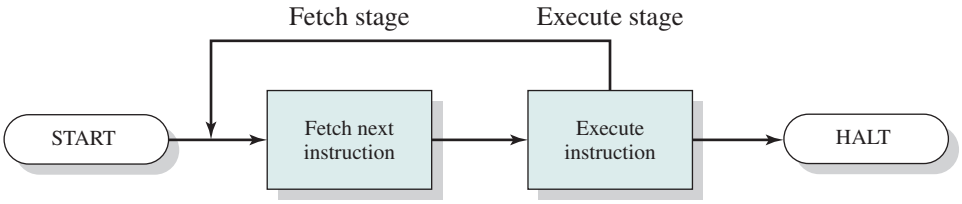


Figure 1.2 **Basic Instruction Cycle**

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 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 specifies that the next instruction 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

(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

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) 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.

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

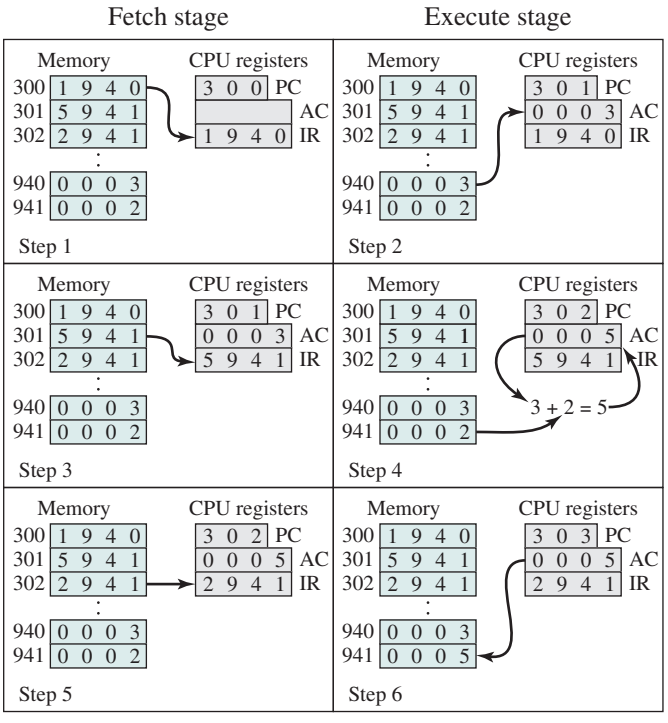


Figure 1.4 **Example of Program Execution (contents of memory and registers in hexadecimal)**

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.

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.

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, or 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.

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.

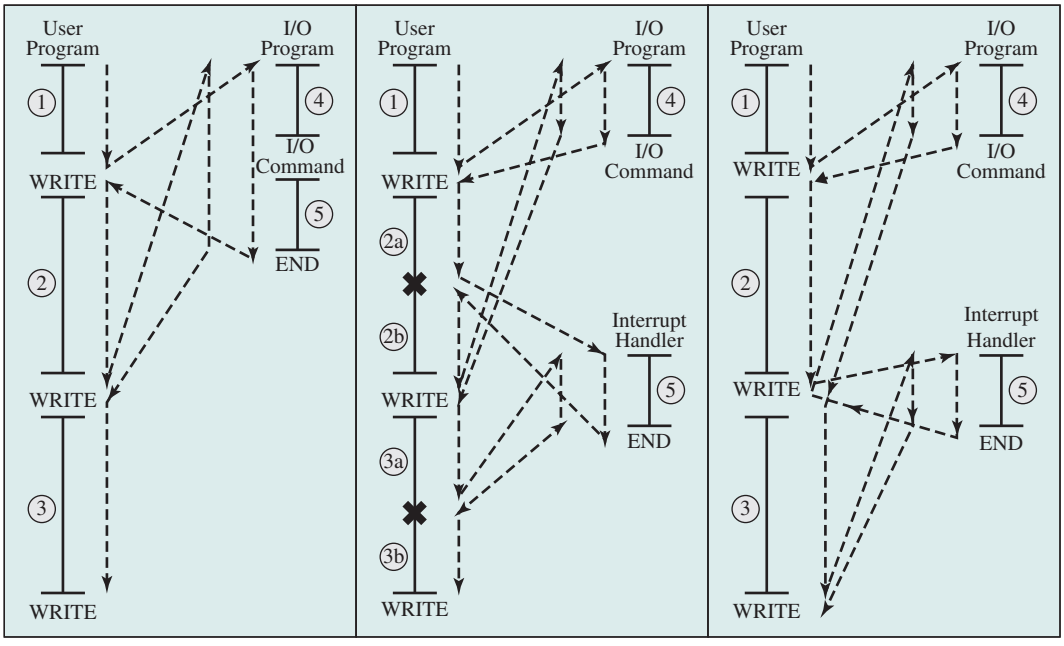
To give a specific example, consider a PC that operates at 1 GHz, which would allow roughly 10^9 instructions per second.[2] A typical hard disk has a rotational speed of 7200 revolutions per minute for a half-track rotation time of 4 ms, which is 4 million times slower than the processor.

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 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 of, 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.

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

[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.



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

✖ = interrupt occurs during course of execution of user program

Figure 1.5 Program Flow of Control Without and With Interrupts

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

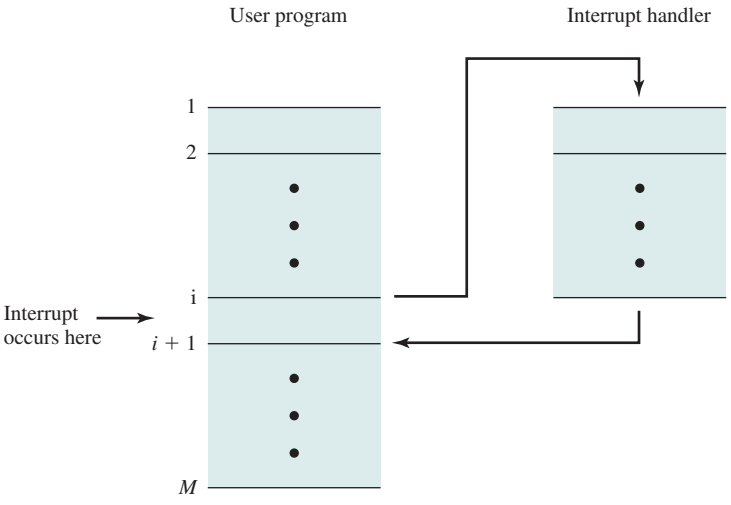


Figure 1.6 **Transfer of Control via Interrupts**

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 (see 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, 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 with Figure 1.2). In the interrupt stage, the processor checks to see if any interrupts have occurred, indicated by the presence of an

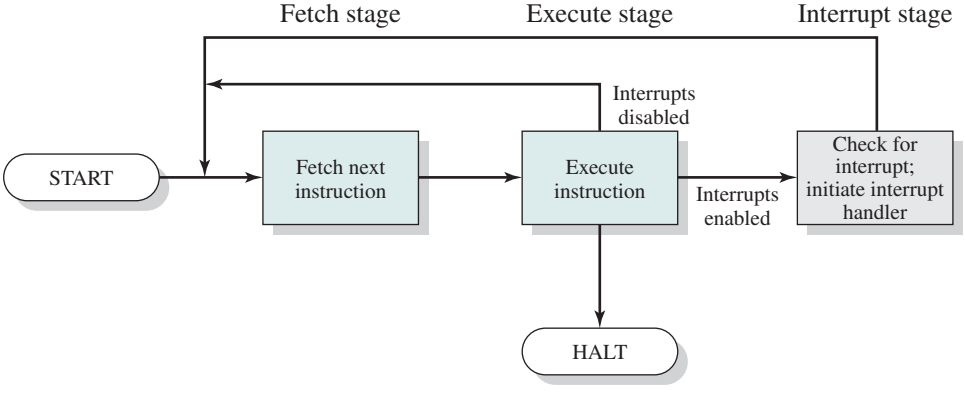


Figure 1.7 **Instruction Cycle with Interrupts**

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, 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 assume

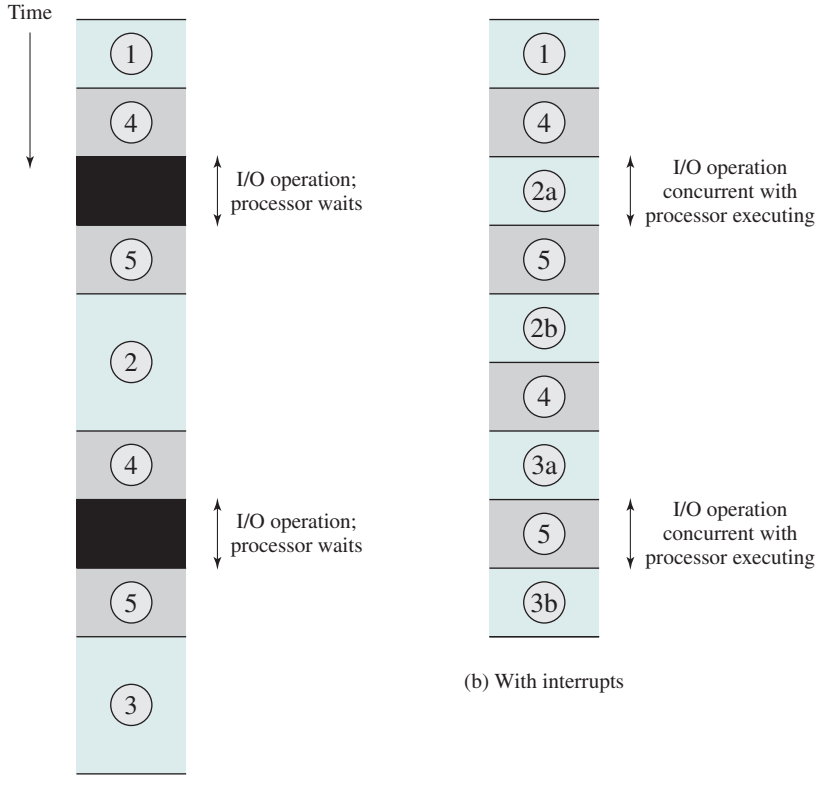


Figure 1.8 **Program Timing: Short I/O Wait**

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 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.

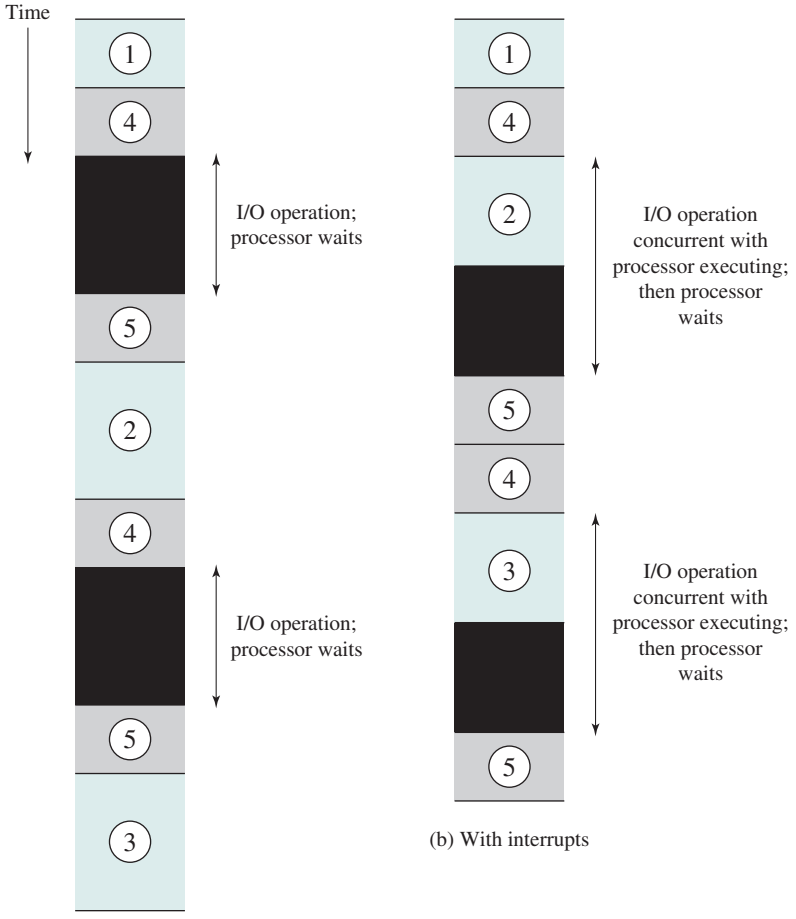


Figure 1.9 **Program Timing: Long I/O Wait**

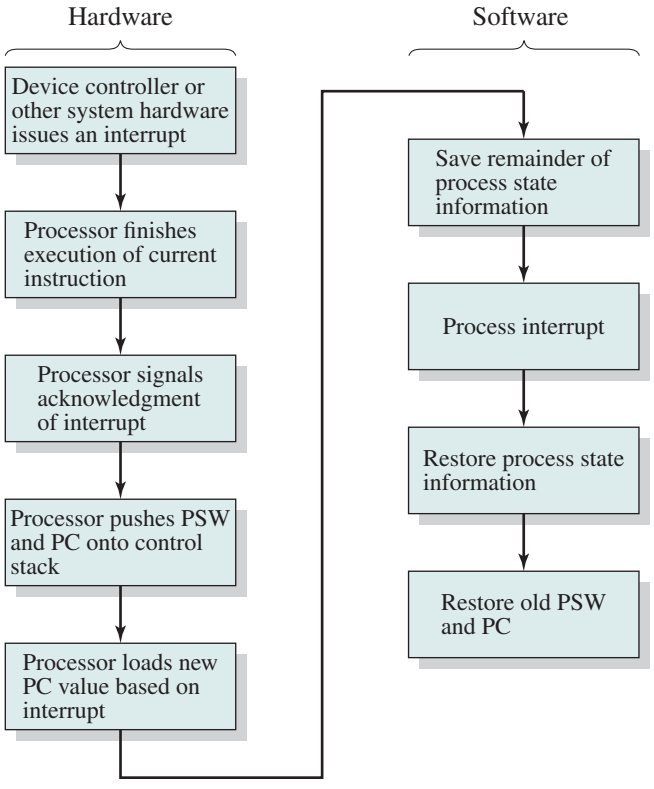


Figure 1.10 **Simple Interrupt Processing**

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 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 is

[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.

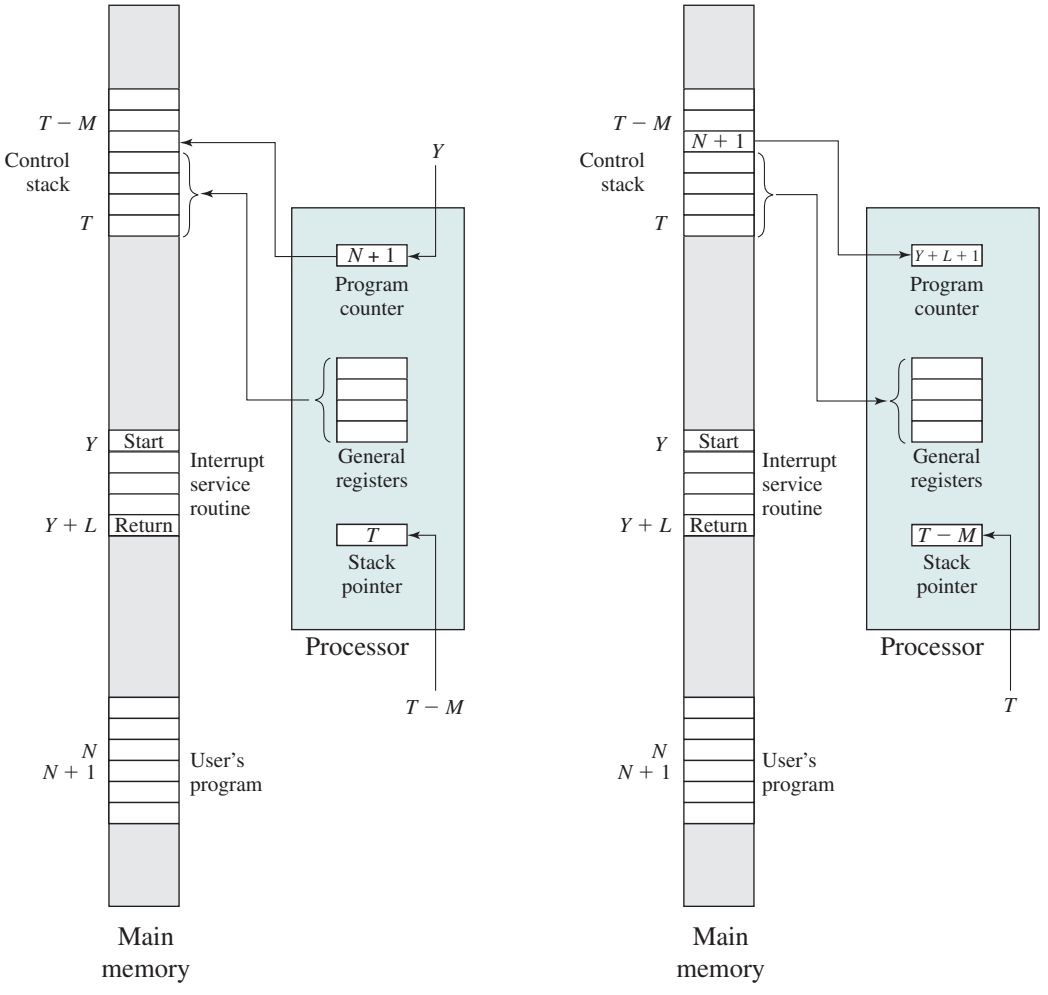
contained in the program counter (PC). These can be pushed onto a control stack (see Appendix P).

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 will be 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 (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.



(a) Interrupt occurs after instruction at location N

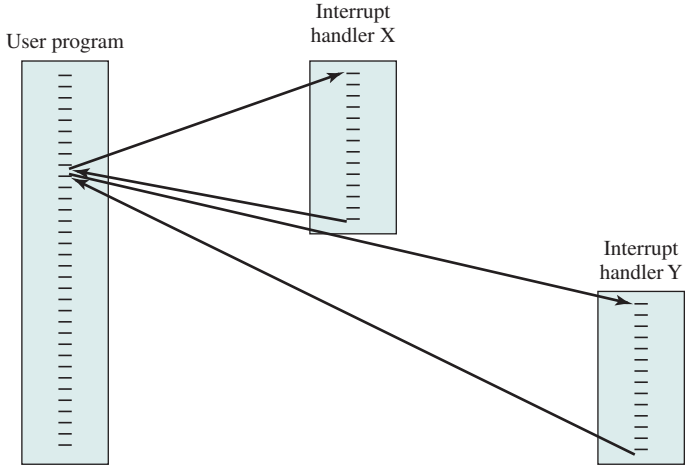
(b) Return from interrupt

Figure 1.11 Changes in Memory and Registers for an Interrupt

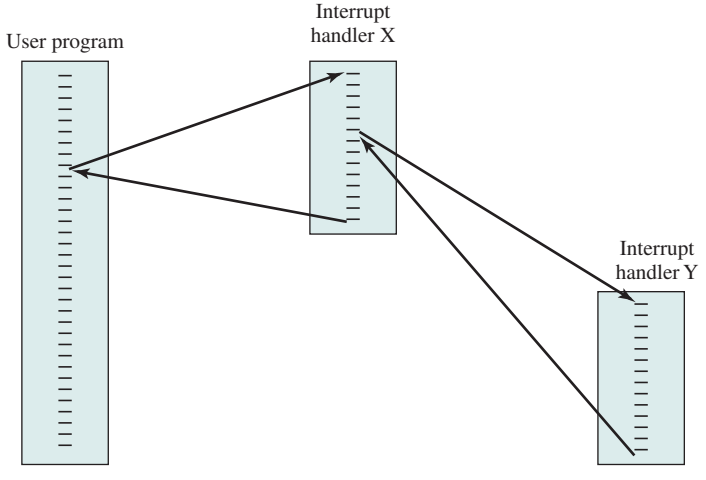
Multiple Interrupts

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 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.

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 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 (see Figure 1.12a).



(a) Sequential interrupt processing



(b) Nested interrupt processing

Figure 1.12 Transfer of Control with Multiple Interrupts

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 (see 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 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.

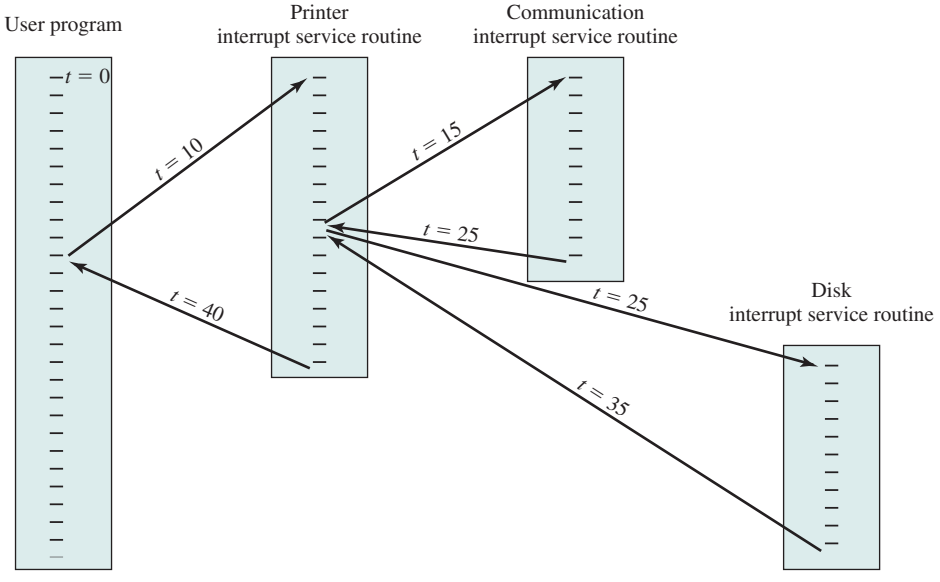


Figure 1.13 Example Time Sequence of Multiple Interrupts

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: 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 the processor has access to two levels of memory. Level 1 contains 1000 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, the byte is first transferred to level 1, 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. 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

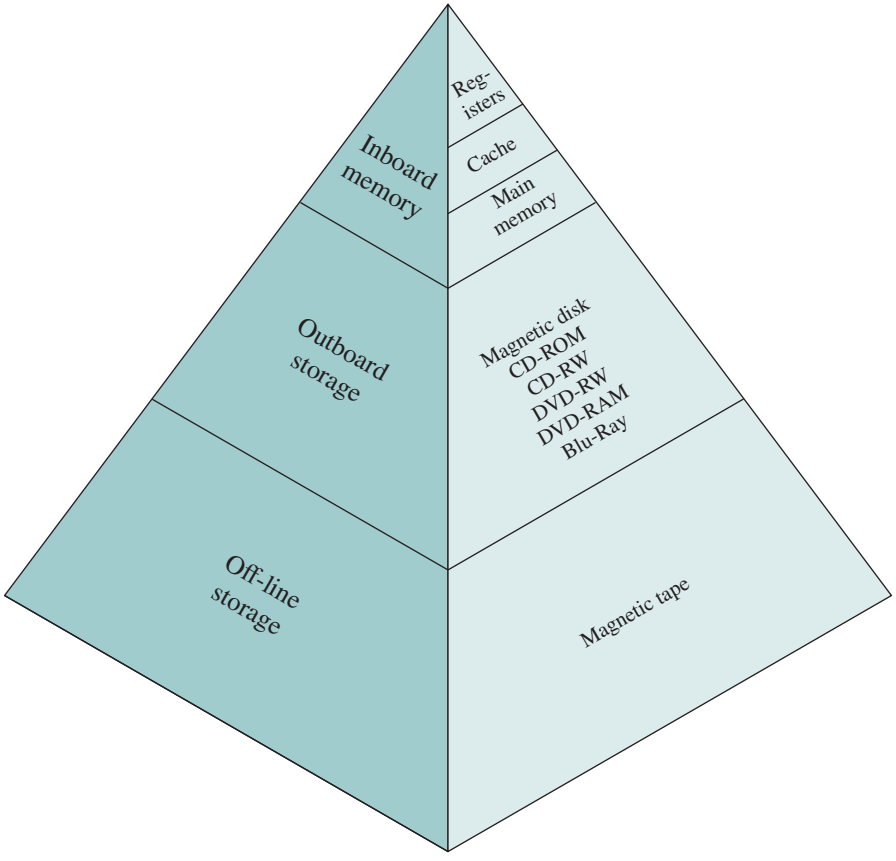


Figure 1.14 **The Memory Hierarchy**

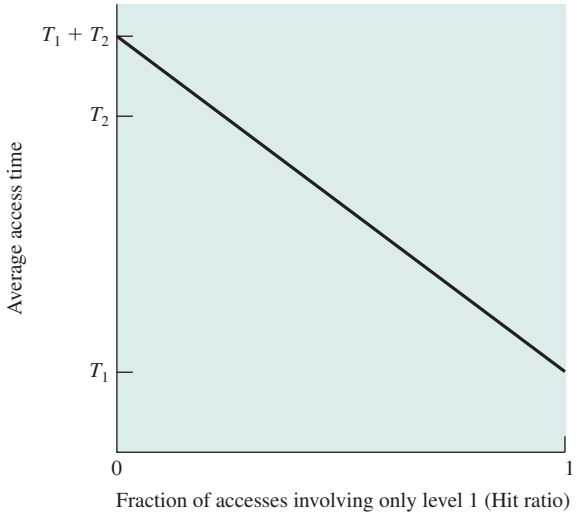


Figure 1.15 **Performance of a Simple Two-Level Memory**

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 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.

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

[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.

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 will be 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 (to be examined in detail in Chapter 11), improves performance in two ways:

1. 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.
2. 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 (to be 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.

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

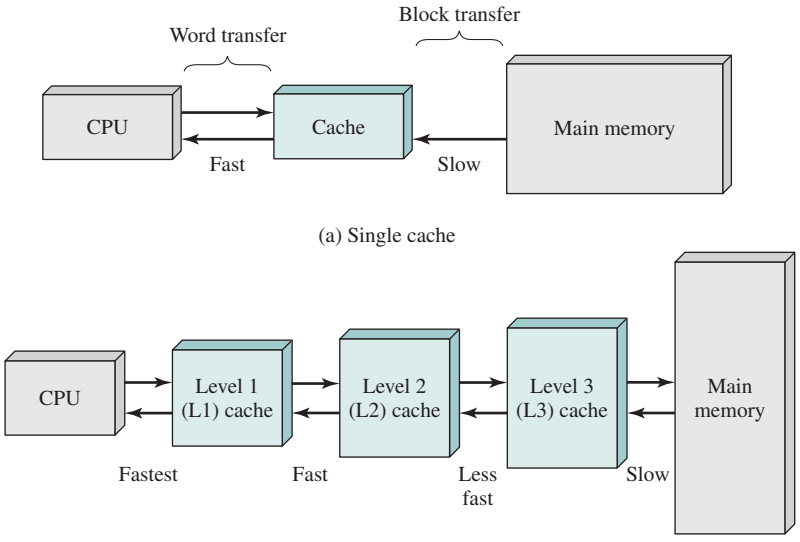


Figure 1.16 Cache and Main Memory

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 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 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.

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

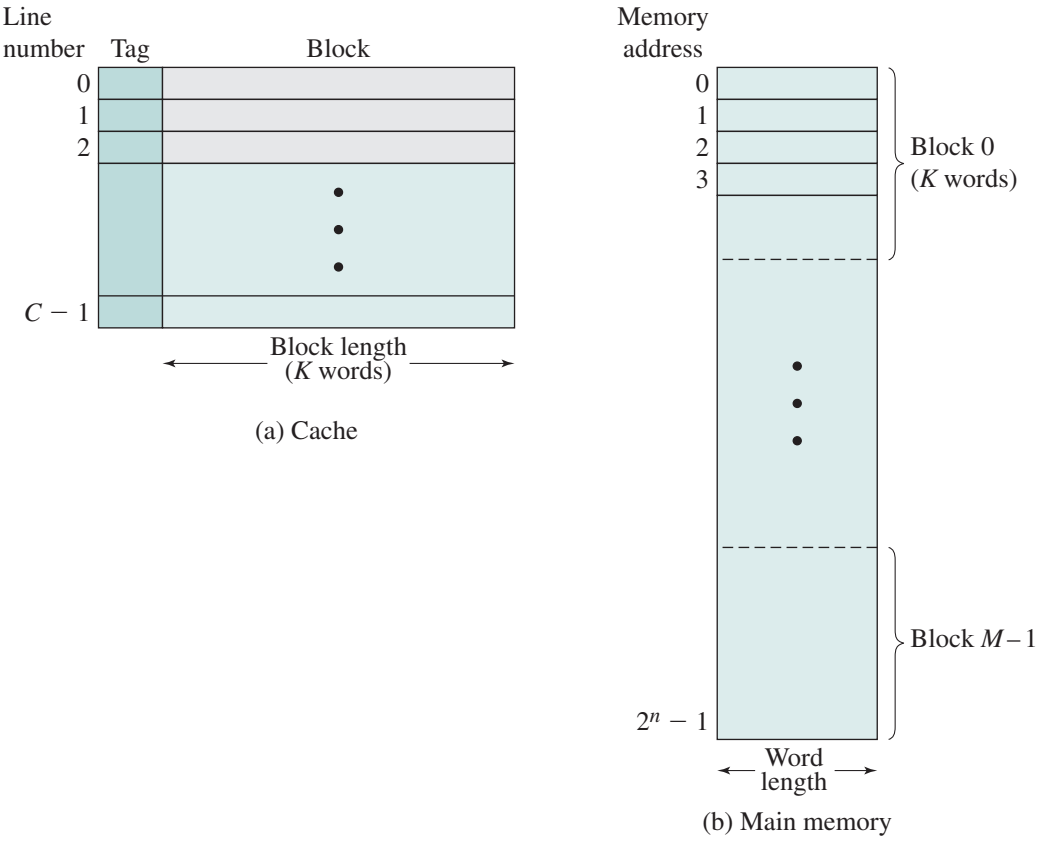


Figure 1.17 **Cache/Main Memory Structure**

As a simple example, suppose 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 addressed in dealing with virtual memory and disk cache design. They fall into the following categories:

- Cache size
- Block size

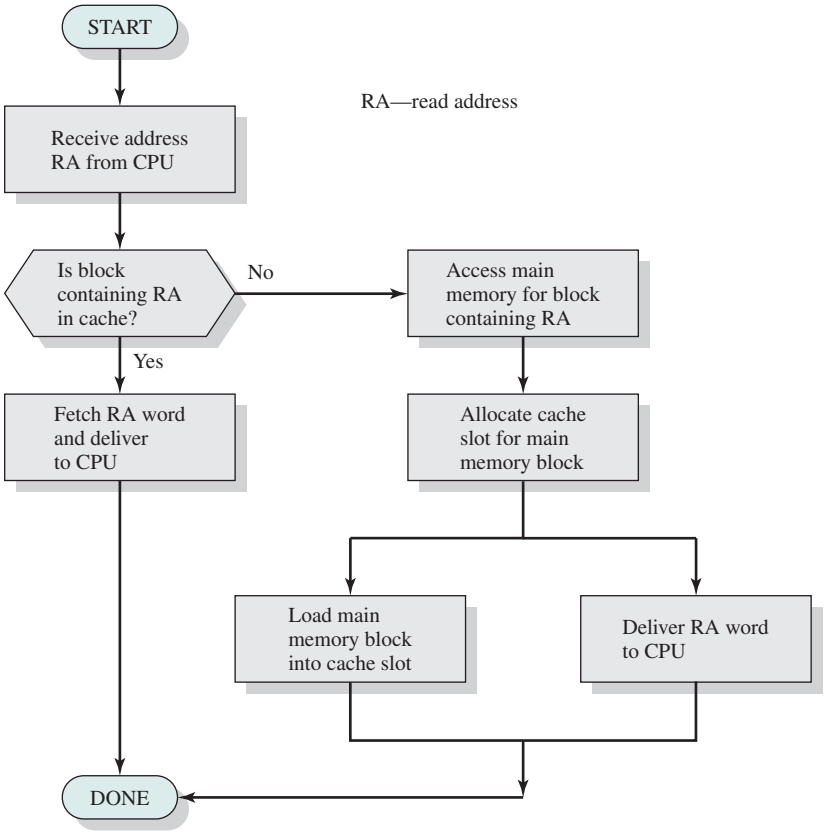


Figure 1.18 **Cache Read Operation**

- 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. Consider beginning with a relatively small block size, then increasing the size. As the block size increases, more useful data are brought into the cache with each block transfer. The result will be that the hit ratio increases 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. 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 L3. A discussion of the performance benefits of multiple cache levels is beyond our current scope (see [STAL16a] 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 will 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, 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, 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 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 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 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 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.

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 will examine three approaches to providing parallelism by replicating processors: symmetric multiprocessors (SMPs), multicore computers, and clusters. SMPs and multicore computers are discussed in this section; clusters will be 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 such 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.
- **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 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 typically has two dedicated levels of cache, designated L1 and L2. As Figure 1.19 indicates, each processor and its dedicated caches are housed on a separate chip. 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

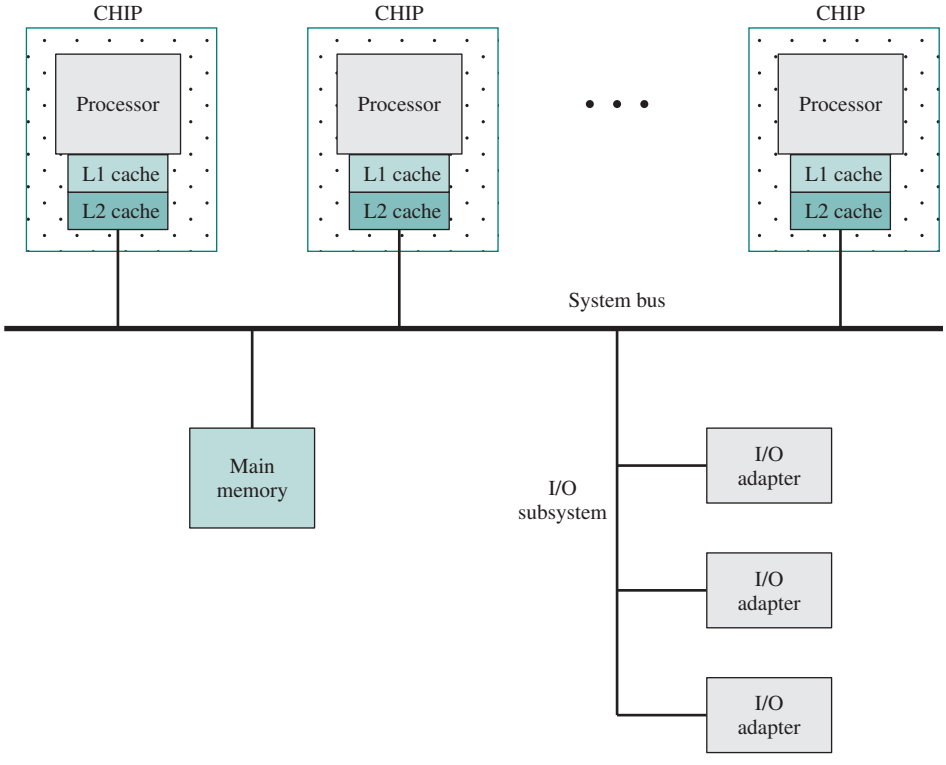


Figure 1.19 Symmetric Multiprocessor Organization

memory (messages and status information left in shared address spaces). It may also be possible for processors to exchange signals directly. The memory is often organized so 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

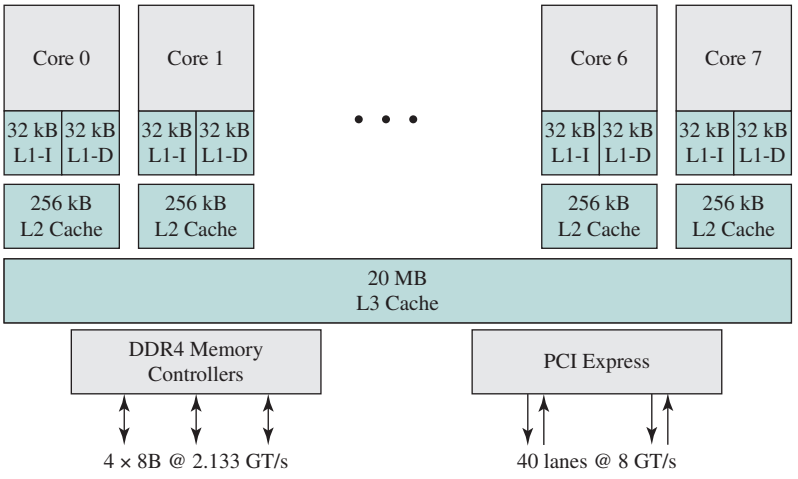
A **multicore** computer, also known as a **chip multiprocessor**, combines two or more processors (called cores) on a single piece of silicon (called a die). Typically, each core consists of all of the components of an independent processor, such as registers, ALU, pipeline hardware, and control unit, 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 current scope, but is summarized in Appendix C.

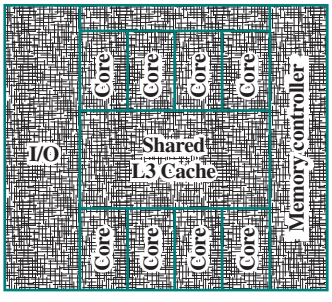
An example of a multicore system is the Intel Core i7-5960X, which includes six x86 processors, each with a dedicated L2 cache, and with a shared L3 cache (see Figure 1.20a). One mechanism Intel uses to make its caches more effective is prefetching, in which the hardware examines memory access patterns and attempts to fill the caches speculatively with data that's likely to be requested soon. Figure 1.20b shows the physical layout of the 5960X in its chip.

The Core i7-5960X chip supports two forms of external communications to other chips. The **DDR4 memory controller** brings the memory controller for the DDR (double data rate) main memory onto the chip. The interface supports four channels that are 8 bytes wide for a total bus width of 256 bits, for an aggregate data rate of up to 64 GB/s. With the memory controller on the chip, the Front Side Bus is eliminated. The **PCI Express** is a peripheral bus and enables high-speed communications among connected processor chips. The PCI Express link operates at 8 GT/s (transfers per second). At 40 bits per transfer, that adds up to 40 GB/s.

[6] A description of hardware-based cache coherency schemes is provided in [STAL16a].



(a) Block diagram



(b) Physical layout on chip

Figure 1.20 **Intel Core i7-5960X Block Diagram**

1.9 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

address register	instruction cycle	programmed I/O
auxiliary memory	instruction register	register
block	interrupt	replacement algorithm
cache memory	interrupt-driven I/O	secondary memory
cache slot	I/O module	slot
central processing unit	locality of reference	spatial locality
chip multiprocessor	main memory	stack
data register	memory hierarchy	stack frame
direct memory access (DMA)	miss	stack pointer
hit	multicore	system bus
hit ratio	multiprocessor	temporal locality
input/output	processor	
instruction	program counter	

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 can multiple interrupts be serviced by setting priorities?
1.6. What characteristics are observed while going up the memory hierarchy?
1.7. What are the trade-offs that determine the size of the 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
0100 = SUB from AC

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

1. Load AC from device 7.
2. SUB from AC contents of memory location 880.
3. Store AC to memory location 881.

Assume that the next value retrieved from device 7 is 6 and that location 880 contains a value of 5.

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 64-bit microprocessor having 64-bit instructions composed of two fields. The first 4 bytes contain the opcode, and the remainder an immediate operand or an operand address.

a. What is the maximum directly addressable memory capacity?
b. What ideal size of microprocessor address buses should be used? How will system speed be affected for data buses of 64 bits, 32 bits and 16 bits?
c. How many bits should the instruction register contain if the instruction register is to contain only the opcode, and how many if the instruction register is to contain the whole instruction?

1.4. Consider a hypothetical microprocessor generating a 16-bit address (e.g., assume 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 64-bit microprocessor, with a 32-bit external data bus, driven by a 16 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 64 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 10800 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. 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. Extend Equations (1.1) and (1.2) in Appendix 1A to 3-level memory hierarchies.

1.12. Consider a memory system with cache having the following parameters:

$$Sc = 32\text{ KB} \quad Cc = 0.1\text{ cents/bytes} \quad T_C = 10\text{ ns}$$
$$Sm = 256\text{ MB} \quad Cm = 0.0001\text{ cents/bytes} \quad Tm = 100\text{ ns}$$

a. What was the total cost prior to addition of cache?
b. What is the total cost after addition of cache?
c. What is the percentage decrease in time due to inclusion of cache with respect to a system without cache memory considering a cache hit ratio of 0.85?

1.13. Suppose that a large file is being accessed by a computer memory system comprising of a cache and a main memory. The cache access time is 60 ns. Time to access main memory (including cache access) is 300 ns. The file can be opened either in read or in write mode. A write operation involves accessing both main memory and the cache (write-through cache). A read operation accesses either only the cache or both the cache and main memory depending upon whether the access word is found in the cache or not. It is estimated that read operations comprise of 80% of all operations. If the cache hit ratio for read operations is 0.9, what is the average access time of 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 will look at some of the performance characteristics of two-level memories that are common to all three approaches.

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

Table 1.3 Relative Dynamic Frequency of High-Level Language Operations

Study	[HUCK83]	[KNUT71]	[PATT82]		[TANE78]
Language	Pascal	FORTRAN	Pascal	C	SAL
Workload	Scientific	Student	System	System	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

programs for typesetting, computer-aided design (CAD), sorting, and file 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].

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

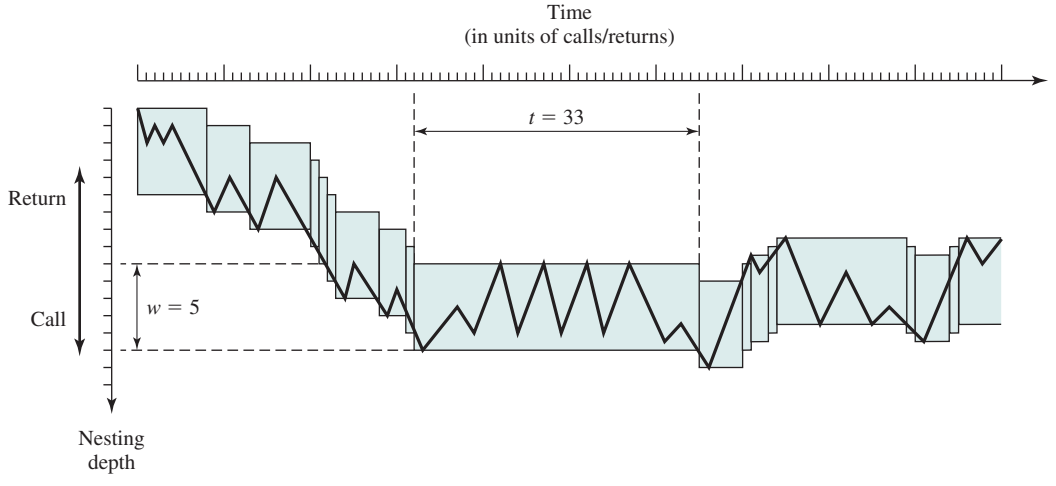


Figure 1.21 Example Call-Return Behavior of a Program

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 temporary storage 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

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

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]

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

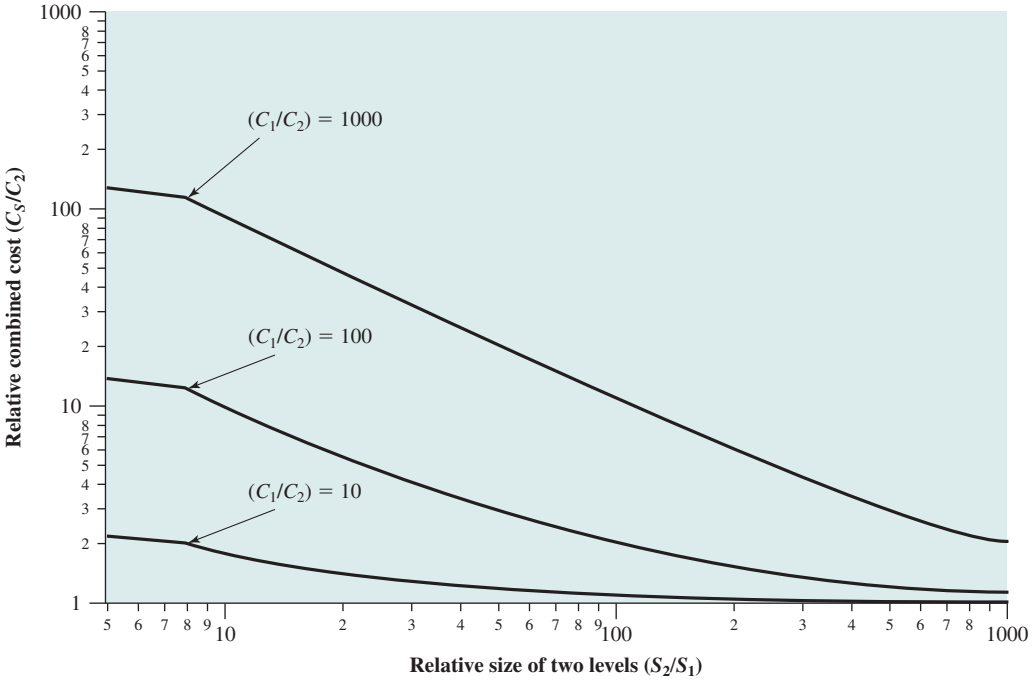


Figure 1.22 **Relationship of Average Memory Cost to Relative Memory Size for a Two-Level Memory**

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?

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),

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

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.

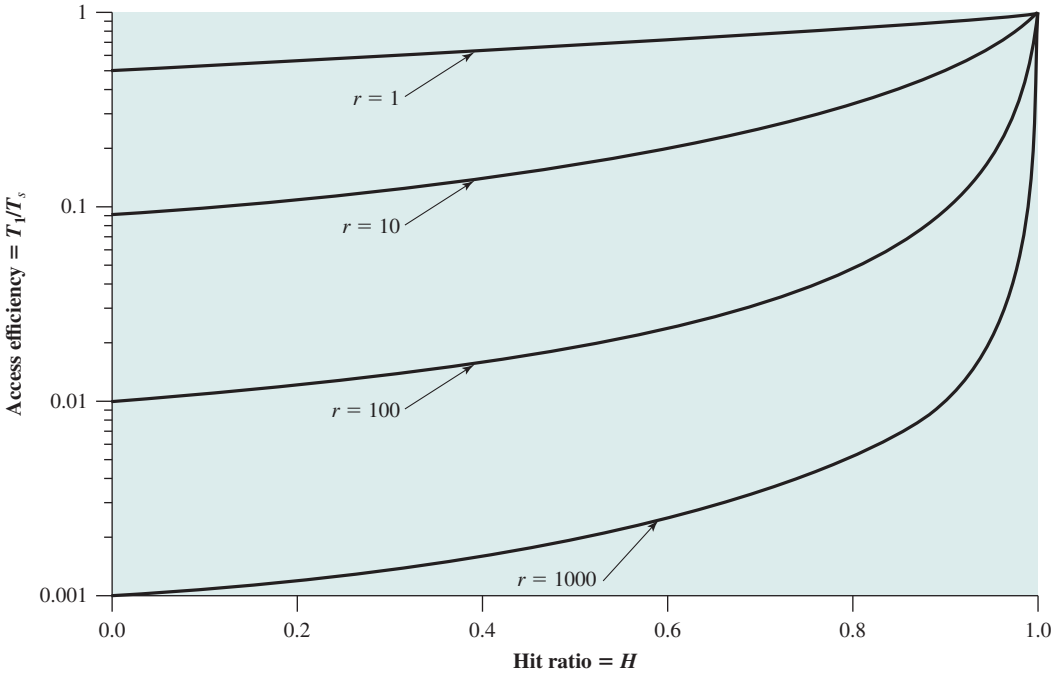


Figure 1.23 **Access Efficiency as a Function of Hit Ratio (r = T2/T1)**

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 also stored in M1. Now suppose 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 have shown that rather small cache sizes will yield a hit ratio above 0.75 *regardless of the size of main memory* ([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

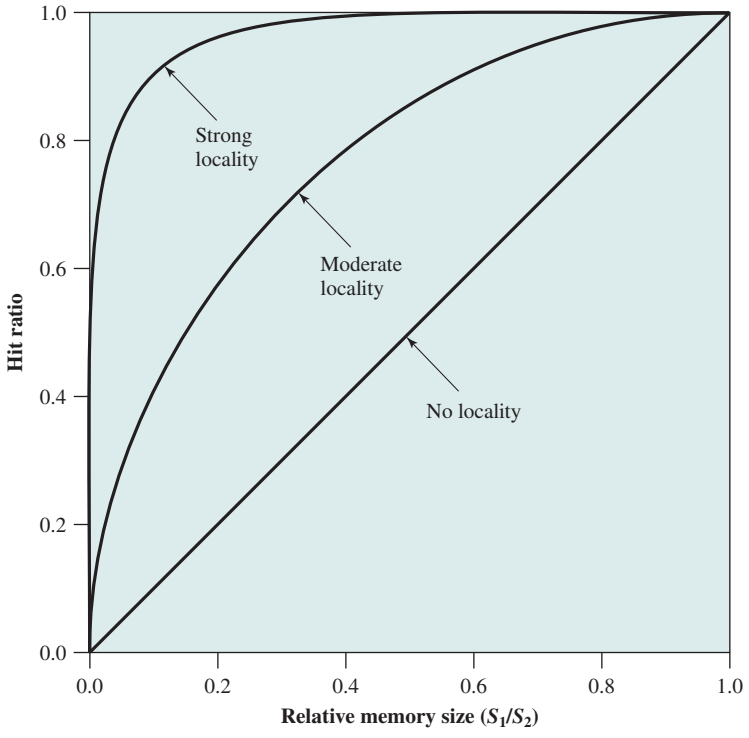


Figure 1.24 **Hit Ratio as a Function of Relative Memory Size**

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. Please note that with L2 cache (or even L2 and L3 caches) involved, analysis is much more complex. See [PEIR99] and [HAND98] for discussions.

CHAPTER 2

Operating System Overview

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.
- 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 will look at how operating systems have evolved from primitive batch systems to sophisticated multitasking, multiuser systems. The remainder of the chapter will look at the history and general characteristics of the two operating systems that serve as examples throughout this book.

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 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 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 a 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

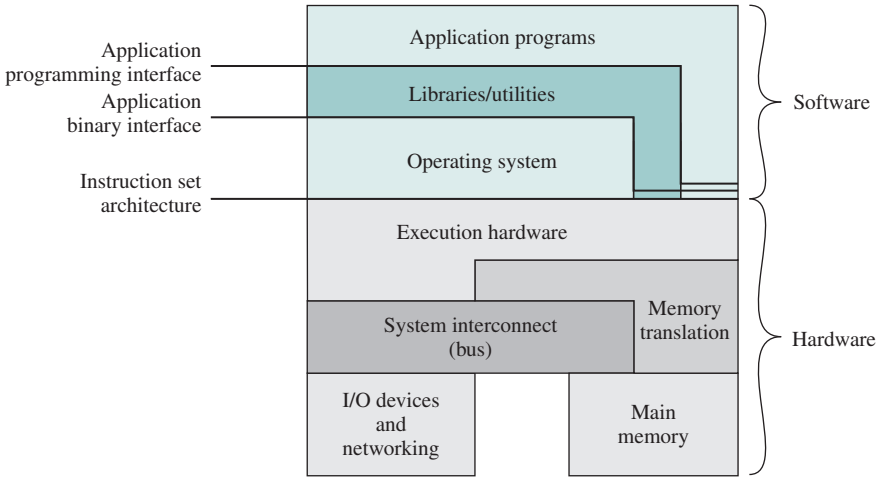


Figure 2.1 Computer Hardware and Software Structure

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 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, or 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.

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 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

The OS is responsible for controlling the use of a computer's resources, such as I/O, main and secondary memory, and processor execution time. 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 consists of instructions executed by the processor. While executing, the OS decides how processor time is to be allocated and which computer resources are available for use. But in order for the processor to act on these decisions, it must cease executing the OS program and execute other programs. Thus, the OS relinquishes control for the processor to do some "useful" work, 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.

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 and utility programs and data. The OS and the memory management hardware in the processor 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.

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, 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.

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

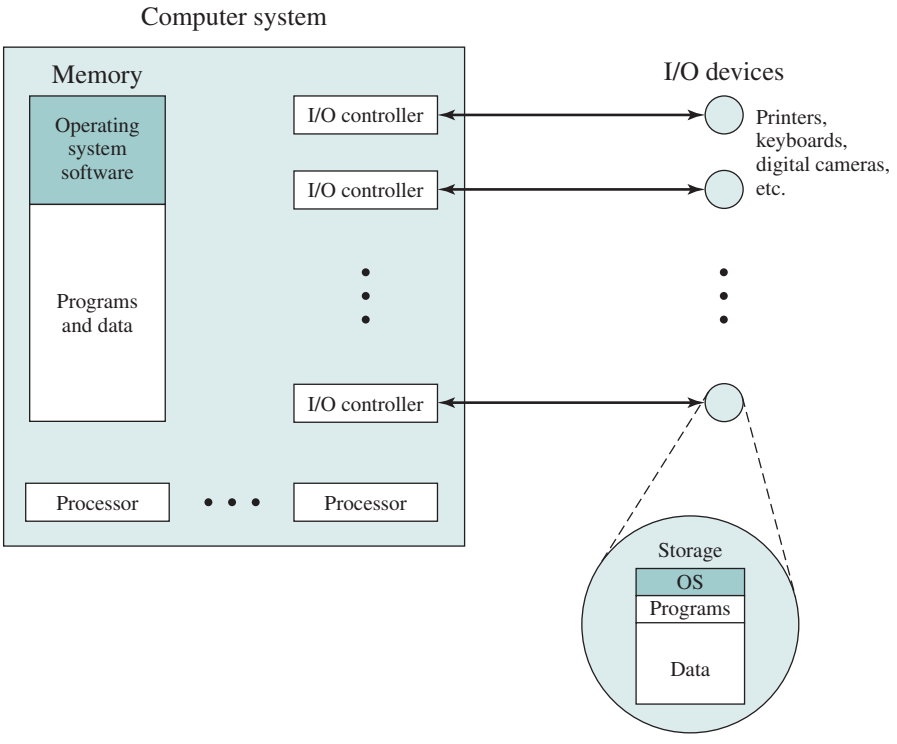


Figure 2.2 **The Operating System as Resource Manager**

- **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 regularly update an OS 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 will 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), then loading and linking together the object program and common functions. Each of these steps could 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

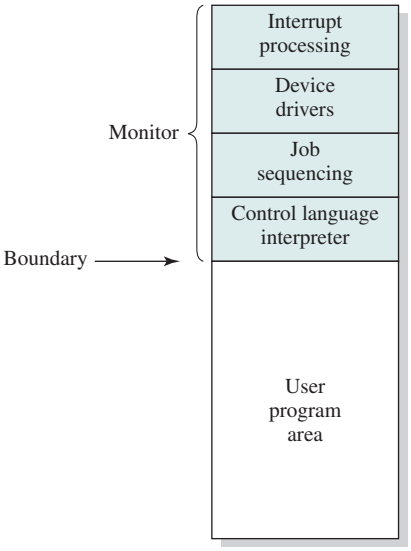


Figure 2.3 Memory Layout for a Resident Monitor

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 (see 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 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 the processor is now fetching and executing instructions in a user program, and "control is returned to the monitor" means 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
.
$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 as 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.
- **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 monitors 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 I/O devices are slow compared to the processor.

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

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

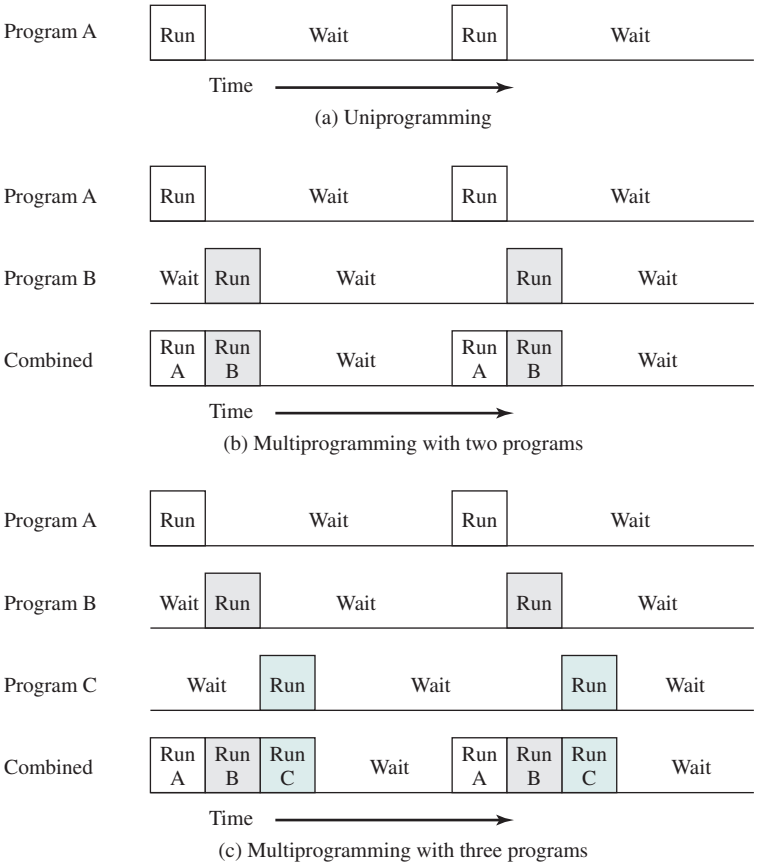


Figure 2.5 Multiprogramming Example

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

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 there must be enough memory to hold the OS (resident monitor) and one user program. Suppose 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 (see Figure 2.5b). Furthermore, we might expand memory to hold three, four, or more programs and switch among all of them (see Figure 2.5c). The approach is known as **multiprogramming**, or **multitasking**. It is the central theme of modern operating systems.

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 the 5 minutes are over, 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 the jobs are run concurrently under a multiprogramming OS. Because there is little resource contention between the jobs, all three can run in

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

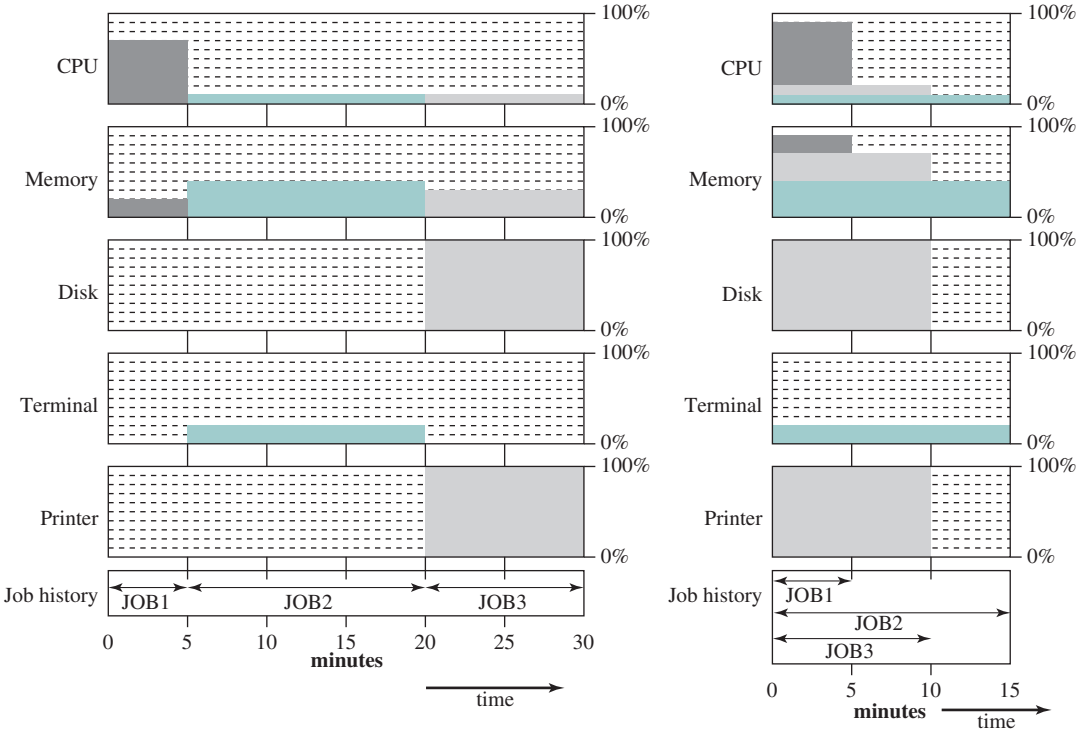


Figure 2.6 Utilization Histograms

nearly minimum time while coexisting with the others in the computer (assuming 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 after the interrupt is handled.

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, and this decision requires an algorithm for scheduling. These concepts will be 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.

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 ported to 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 5,000 of those. 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 5,000th 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.

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

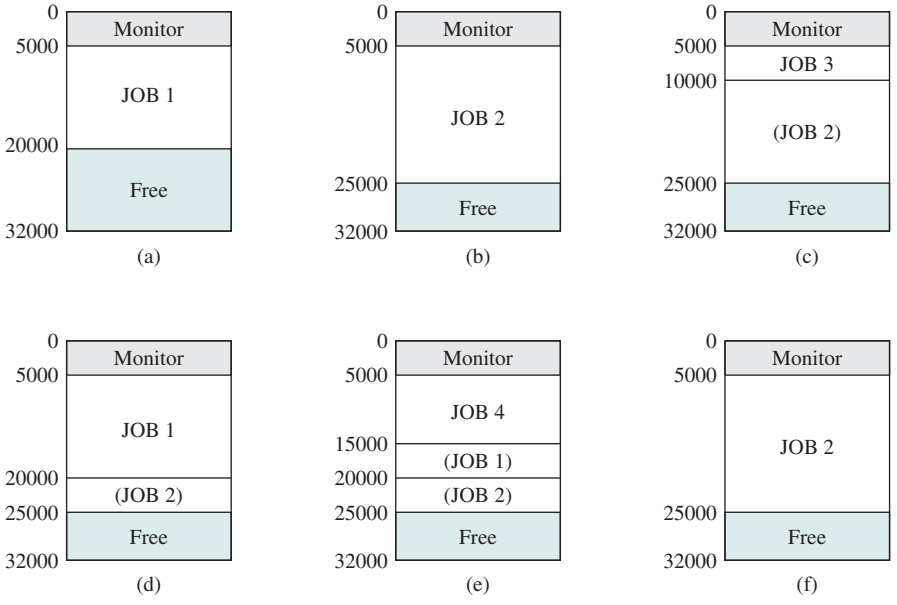


Figure 2.7 CTSS Operation

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 there are four interactive users with the following memory requirements, in words:

- JOB1: 15,000
- JOB2: 20,000
- JOB3: 5,000
- JOB4: 10,000

Initially, the monitor loads JOB1 and transfers control to it (Figure 2.7a). 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 (Figure 2.7b). 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 (Figure 2.7c). 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 (Figure 2.7d). When JOB4 is loaded, part of JOB1 and the portion of JOB2 remaining in memory are retained (Figure 2.7e). 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 the missing portion of JOB2 be read in (Figure 2.7f).

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 only authorized users have access 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 four areas span many of the key design and implementation issues of modern operating systems. The brief review of these four 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.

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, 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 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:

1. An executable program
2. The associated data needed by the program (variables, work space, buffers, etc.)
3. 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 the OS needs to manage the process, and 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

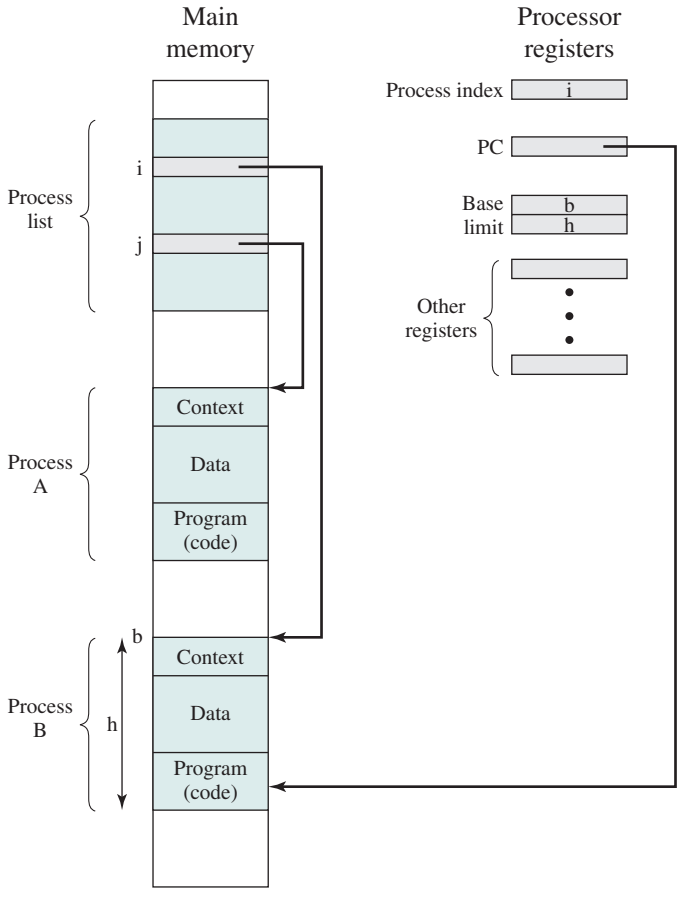


Figure 2.8 Typical Process Implementation

points to the next instruction in that process to be executed. The base and limit registers define the region in memory occupied 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 the execution of process A. The process switch consists of saving 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 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:

1. **Process isolation:** The OS must prevent independent processes from interfering with each other's memory, both data and instructions.
2. **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.
3. **Support of modular programming:** Programmers should be able to define program modules, and to dynamically create, destroy, and alter the size of modules.
4. **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.
5. **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 concurrently reside in main memory, so 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 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 simultaneously reside in main memory. 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, in coordination with the OS, arranges for the missing page to be loaded. Such a scheme is referred to as **virtual memory** and is depicted in Figure 2.9.

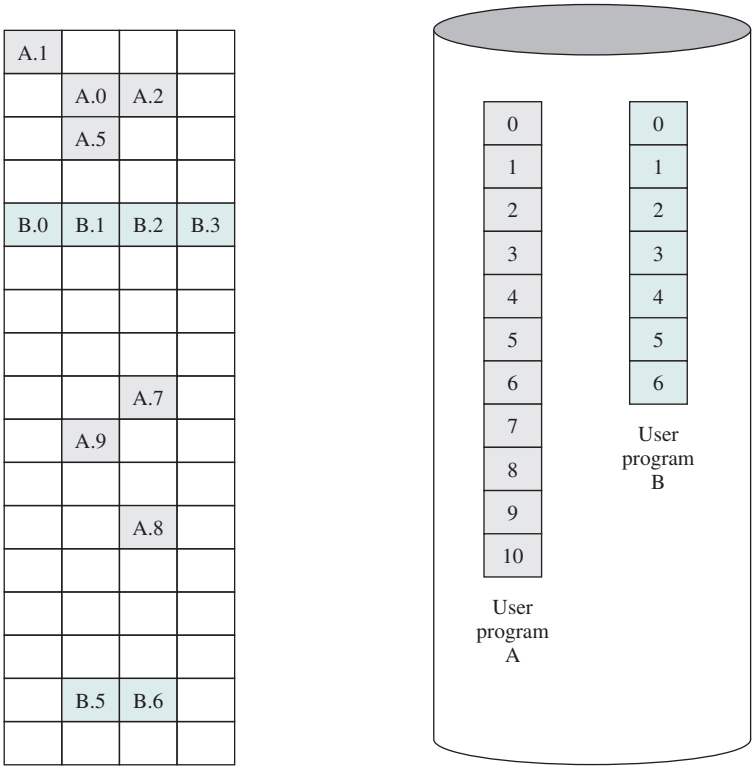


Figure 2.9 Virtual Memory Concepts

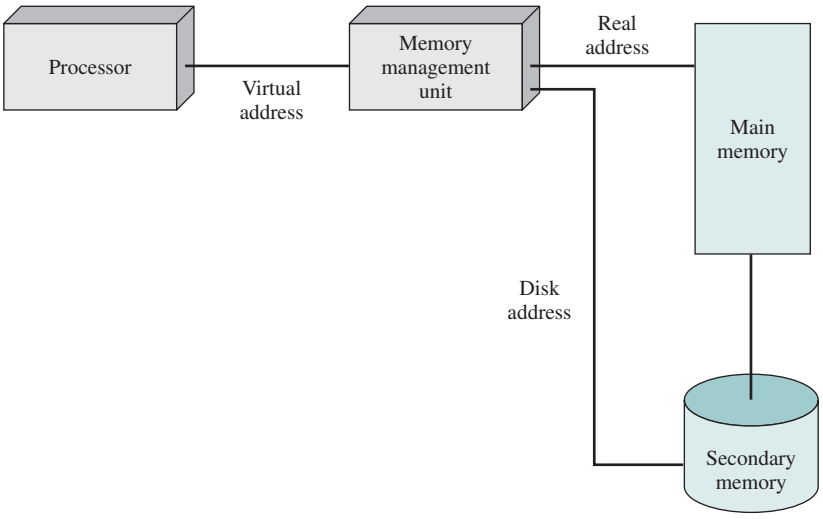


Figure 2.10 Virtual Memory Addressing

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 (a 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 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:

1. **Availability:** Concerned with protecting the system against interruption.
2. **Confidentiality:** Assures that users cannot read data for which access is unauthorized.
3. **Data integrity:** Protection of data from unauthorized modification.
4. **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:

1. **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.
2. **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; the process can then immediately use the device, then release it for later demands from other processes.
3. **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,

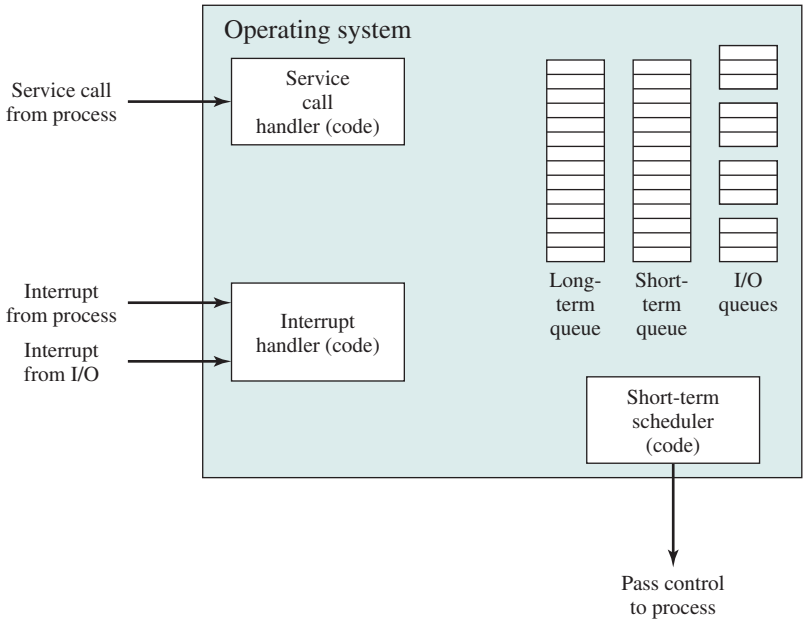


Figure 2.11 **Key Elements of an Operating System for Multiprogramming**

or 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.

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

Until recently, most operating systems 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 space management, 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

stack (to enable subroutine branching). A thread executes sequentially and is interruptible so 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 will be 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 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

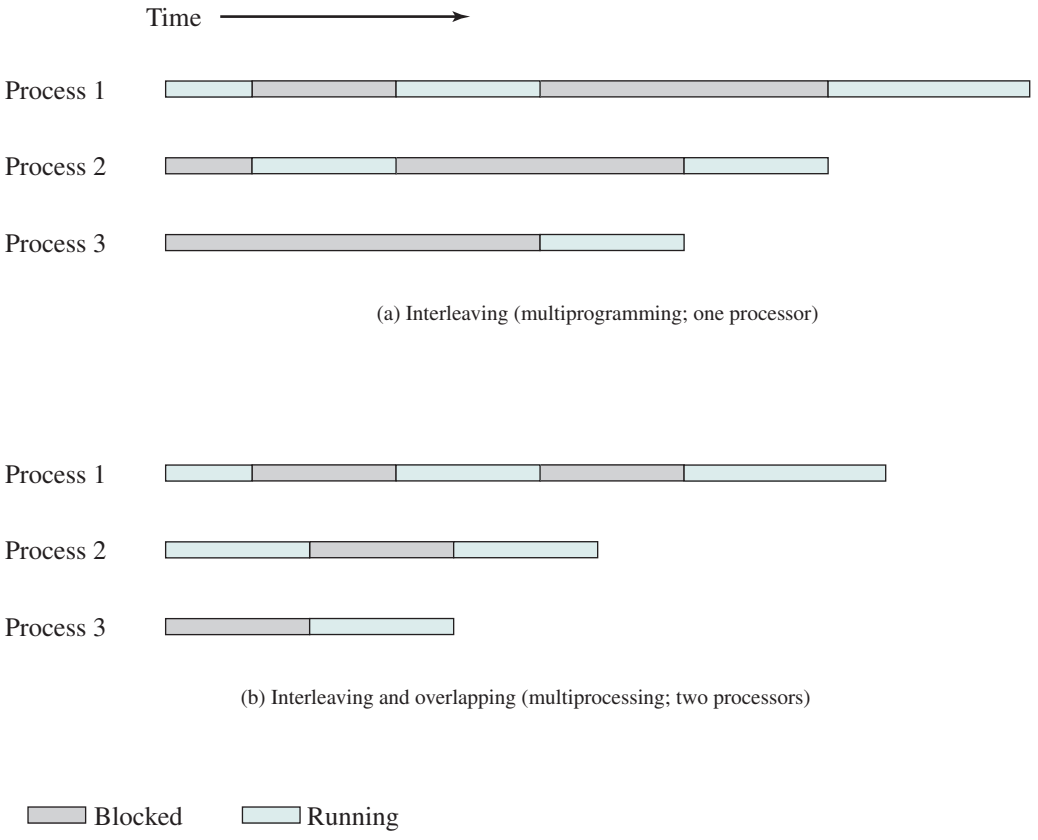


Figure 2.12 Multiprogramming and Multiprocessing

on 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 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 behind that of uniprocessor and SMP operating systems. We will 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.

2.5 FAULT TOLERANCE

Fault tolerance refers to the ability of a system or component to continue normal operation despite the presence of hardware or software faults. This typically involves some degree of redundancy. Fault tolerance is intended to increase the reliability of a system. Typically, increased fault tolerance (and therefore increased reliability) comes with a cost, either in financial terms or performance, or both. Thus, the extent adoption of fault tolerance measures must be determined by how critical the resource is.

Fundamental Concepts

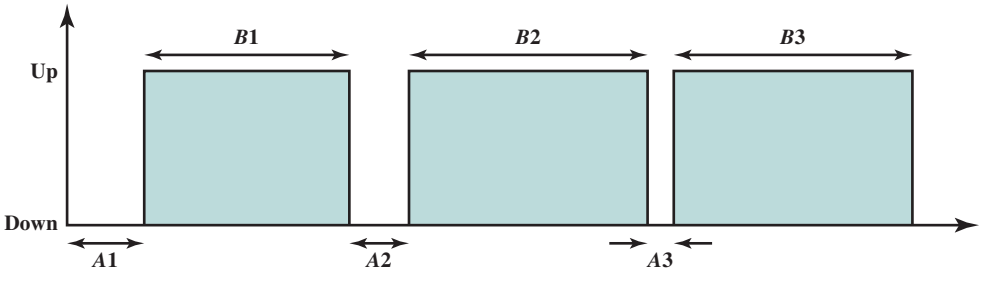
The three basic measures of the quality of the operation of a system that relate to fault tolerance are reliability, mean time to failure (MTTF), and availability. These concepts were developed with specific reference to hardware faults, but apply more generally to hardware and software faults.

The **reliability** $R(t)$ of a system is defined as the probability of its correct operation up to time t given that the system was operating correctly at time $t = 0$. For computer systems and operating systems, the term *correct operation* means the correct execution of a set of programs, and the protection of data from unintended modification. The **mean time to failure (MTTF)** is defined as

$$\text{MTTF} = \int_0^\infty R(t)$$

The **mean time to repair (MTTR)** is the average time it takes to repair or replace a faulty element. Figure 2.13 illustrates the relationship between MTTF and MTTR.

The **availability** of a system or service is defined as the fraction of time the system is available to service users' requests. Equivalently, availability is the probability that an entity is operating correctly under given conditions at a given instant of time. The time during which the system is not available is called **downtime**; the time during



$$MTTF = \frac{B1 + B2 + B3}{3} \qquad MTTR = \frac{A1 + A2 + A3}{3}$$

Figure 2.13 System Operational States

Table 2.4 Availability Classes

Class	Availability	Annual Downtime
Continuous	1.0	0
Fault tolerant	0.99999	5 minutes
Fault resilient	0.9999	53 minutes
High availability	0.999	8.3 hours
Normal availability	0.99–0.995	44–87 hours

which the system is available is called **uptime**. The availability A of a system can be expressed as follows:

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Table 2.4 shows some commonly identified availability levels and the corresponding annual downtime.

Often, the mean uptime, which is MTTF, is a better indicator than availability. A small downtime and a small uptime combination may result in a high availability measure, but the users may not be able to get any service if the uptime is less than the time required to complete a service.

Faults

The IEEE Standards Dictionary defines a **fault** as an erroneous hardware or software state resulting from component failure, operator error, physical interference from the environment, design error, program error, or data structure error. The standard also states that a fault manifests itself as (1) a defect in a hardware device or component; for example, a short circuit or broken wire, or (2) an incorrect step, process, or data definition in a computer program.

We can group faults into the following categories:

- **Permanent:** A fault that, after it occurs, is always present. The fault persists until the faulty component is replaced or repaired. Examples include disk head crashes, software bugs, and a burnt-out communications component.
- **Temporary:** A fault that is not present all the time for all operating conditions. Temporary faults can be further classified as follows:

  —**Transient:** A fault that occurs only once. Examples include bit transmission errors due to an impulse noise, power supply disturbances, and radiation that alters a memory bit.

  —**Intermittent:** A fault that occurs at multiple, unpredictable times. An example of an intermittent fault is one caused by a loose connection.

In general, fault tolerance is built into a system by adding redundancy. Methods of redundancy include the following:

- **Spatial (physical) redundancy:** Physical redundancy involves the use of multiple components that either perform the same function simultaneously, or are

configured so one component is available as a backup in case of the failure of another component. An example of the former is the use of multiple parallel circuitry with the majority result produced as output. An example of the latter is a backup name server on the Internet.

- **Temporal redundancy:** Temporal redundancy involves repeating a function or operation when an error is detected. This approach is effective with temporary faults, but not useful for permanent faults. An example is the retransmission of a block of data when an error is detected, such as is done with data link control protocols.
- **Information redundancy:** Information redundancy provides fault tolerance by replicating or coding data in such a way that bit errors can be both detected and corrected. An example is the error-control coding circuitry used with memory systems, and error-correction techniques used with RAID disks, as will be described in subsequent chapters.

Operating System Mechanisms

A number of techniques can be incorporated into OS software to support fault tolerance. A number of examples will be evident throughout the book. The following list provides examples:

- **Process isolation:** As was mentioned earlier in this chapter, processes are generally isolated from one another in terms of main memory, file access, and flow of execution. The structure provided by the OS for managing processes provides a certain level of protection for other processes from a process that produces a fault.
- **Concurrency controls:** Chapters 5 and 6 will discuss some of the difficulties and faults that can occur when processes communicate or cooperate. These chapters will also discuss techniques used to ensure correct operation and to recover from fault conditions, such as deadlock.
- **Virtual machines:** Virtual machines, as will be discussed in Chapter 14, provide a greater degree of application isolation and hence fault isolation. Virtual machines can also be used to provide redundancy, with one virtual machine serving as a backup for another.
- **Checkpoints and rollbacks:** A checkpoint is a copy of an application's state saved in some storage that is immune to the failures under consideration. A rollback restarts the execution from a previously saved checkpoint. When a failure occurs, the application's state is rolled back to the previous checkpoint and restarted from there. This technique can be used to recover from transient as well as permanent hardware failures, and certain types of software failures. Database and transaction processing systems typically have such capabilities built in.

A much wider array of techniques could be discussed, but a full treatment of OS fault tolerance is beyond our current scope.

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 (such as data structures) and coordinating actions (such as 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 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 will be 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, and will be described in Chapter 5.
- **Memory management:** Memory management on a multiprocessor must deal with all of the issues found on uniprocessor computers, and will be 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.

- **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 ten or more 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 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. 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, 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 will introduce two general strategies in this section, and will 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.

One of the most effective initiatives to support developers is Grand Central Dispatch (GCD), implemented in the latest release of the UNIX-based Mac OS X and the iOS operating systems. GCD is a multicore support capability. It 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 will 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, 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.

2.7 MICROSOFT WINDOWS OVERVIEW

Background

Microsoft initially used the name Windows in 1985, for an operating environment extension to the primitive MS-DOS operating system, which was a successful OS used on early personal computers. This Windows/MS-DOS combination was ultimately replaced by a new version of Windows, known as Windows NT, first released in 1993, and intended for laptop and desktop systems. Although the basic internal architecture has remained roughly the same since Windows NT, the OS has continued to evolve with new functions and features. The latest release at the time of this writing is Windows 10. Windows 10 incorporates features from the preceding desktop/laptop release, Windows 8.1, as well as from versions of Windows intended for mobile devices for the Internet of Things (IoT). Windows 10 also incorporates software from the Xbox One system. The resulting unified Windows 10 supports desktops, laptops, smart phones, tablets, and Xbox One.

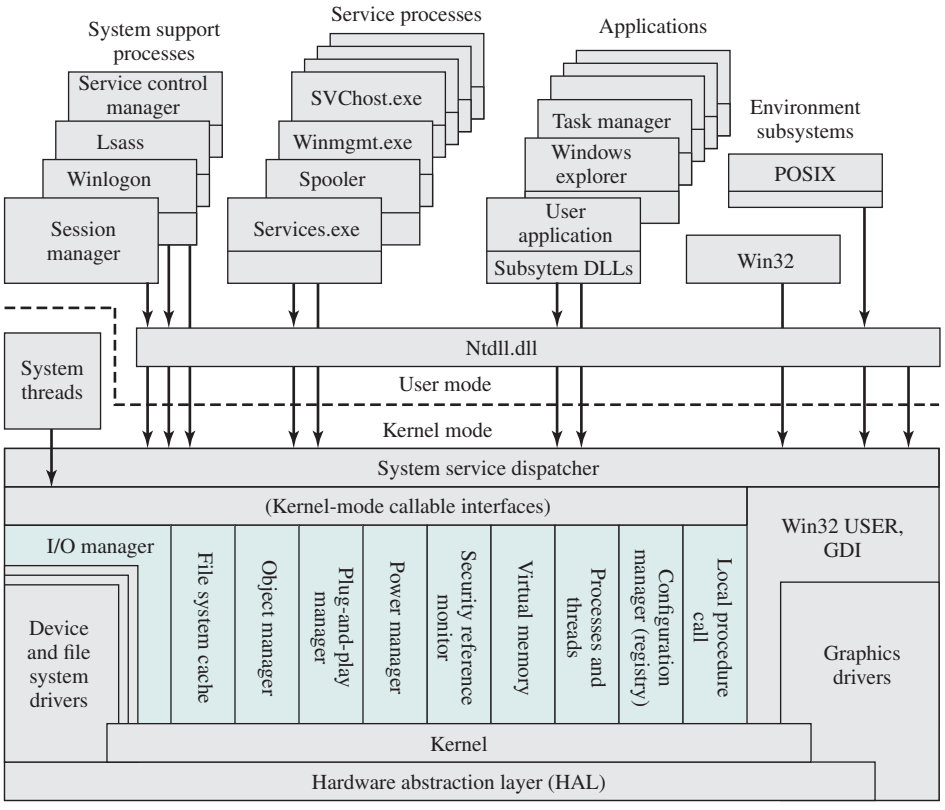
Architecture

Figure 2.14 illustrates the overall structure of Windows. 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 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 levels, 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,



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

Colored area indicates Executive

Figure 2.14 Windows Internals Architecture [RUSS11]

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 will be 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 by 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 process's handle table, which consist of access control information and a pointer to the object. Windows objects will be discussed later in this section.
- **Plug-and-play manager:** Determines which drivers are required to support a particular device and loads those drivers.
- **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 will be 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 will be described in Chapter 8.
- **Process/thread manager:** Creates, manages, and deletes process and thread objects. Windows process and thread management will be 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 Windows supports four basic types of user-mode processes:

1. **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.
2. **Service processes:** The printer spooler, the event logger, user-mode components that cooperate with device drivers, various network services, and 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.
3. **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.
4. **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.

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 applications 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 will be 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.
- **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 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 are common to all object types (e.g., object name, security parameters, usage count), 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 process 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 process 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 will be 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 2.5 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.

Table 2.5 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 codes can be profiled.

2.8 TRADITIONAL UNIX SYSTEMS

History

UNIX was initially developed at Bell Labs and became operational on a PDP-7 in 1970. 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 on 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.

Description

The classic UNIX architecture can be pictured as in three levels: hardware, kernel, and user. The OS is often called the system kernel, or simply the kernel, to emphasize its isolation from the user and applications. It interacts directly with the hardware. 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, which supports system calls from applications, other interface software, and the components of the C compiler (compiler, assembler, loader). The level above this consists of user applications and the user interface to the C compiler.

A look at the kernel is provided in Figure 2.15. 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 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.

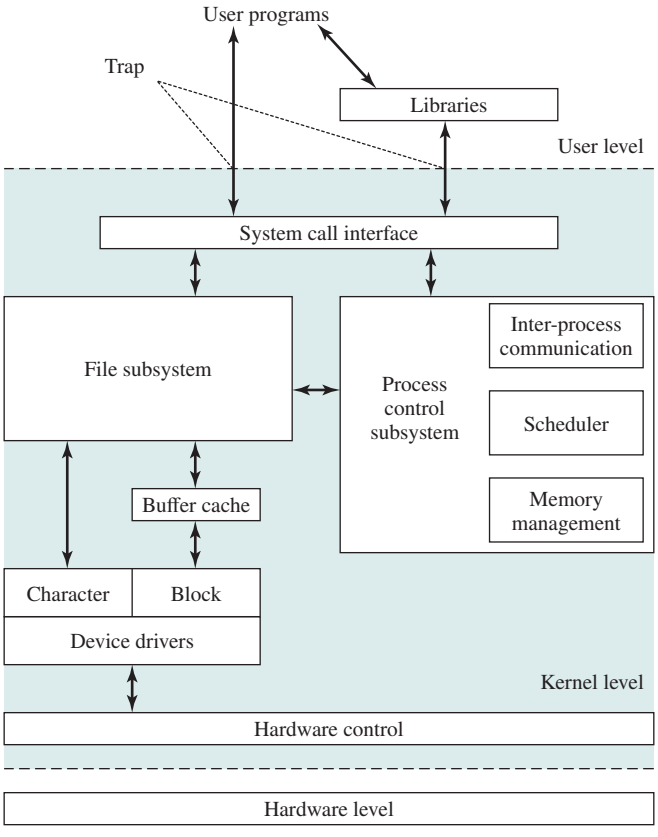


Figure 2.15 **Traditional UNIX Architecture**

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 2.16. There is a small core of facilities, written in 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 (see Figure 2.17).

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,

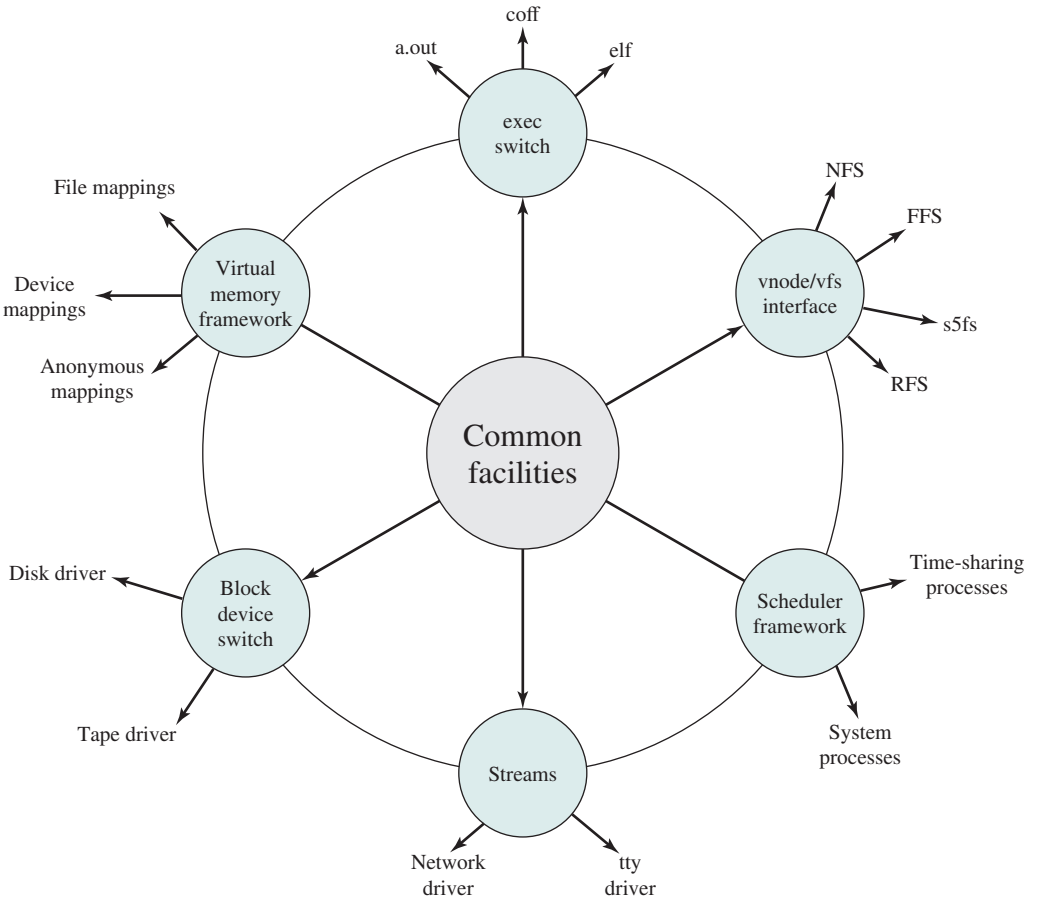


Figure 2.16 **Modern UNIX Kernel**

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 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.

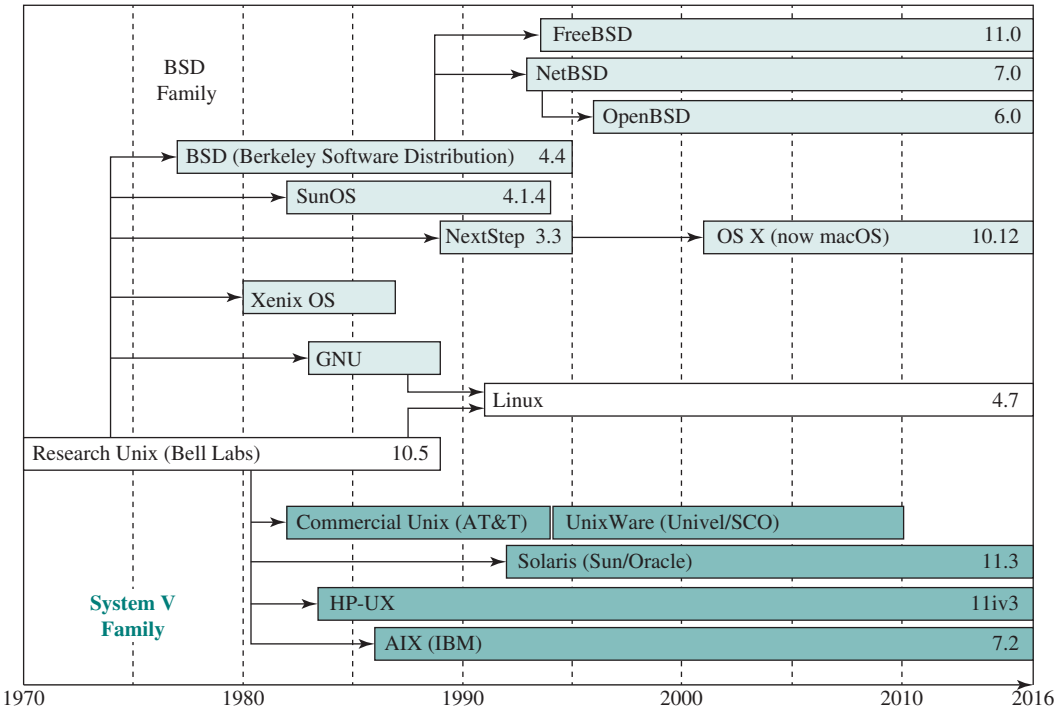


Figure 2.17 UNIX Family Tree

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.

There are several widely used, open-source versions of BSD. FreeBSD is popular for Internet-based servers and firewalls and is used in a number of embedded systems. NetBSD is available for many platforms, including large-scale server systems, desktop systems, and handheld devices, and is often used in embedded systems. OpenBSD is an open-source OS that places special emphasis on security.

The latest version of the Macintosh OS, originally known as OS X and now called MacOS, is based on FreeBSD 5.0 and the Mach 3.0 microkernel.

Solaris 11

Solaris is Oracle's SVR4-based UNIX release, with the latest version being 11. 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 one the most widely used and most successful commercial UNIX implementations.