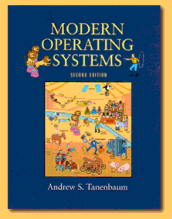
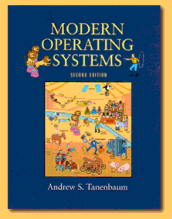
**MODERN OPERATING** 

**MODERN OPERATING**

**SYSTEMS**

**SYSTEMS**

**SECOND EDITION**

**SECOND EDITION**

**by Andrew S. Tanenbaum**

**by Andrew S. Tanenbaum**

Publisher: Prentice Hall PTR

Publisher: Prentice Hall PTR

ISBN 0-13-031358-0, 976 pages

ISBN 0-13-031358-0, 976 pages

Table of Contents

Table of Contents

About the Author

About the Author

**MODERN OPERATING SYSTEMS**

PREFACE

1 INTRODUCTION

1.1 WHAT IS AN OPERATING SYSTEM?

1.1.1 The Operating System as an Extended Machine

1.1.2 The Operating System as a Resource Manager

1.2 HISTORY OF OPERATING SYSTEMS

1.2.1 The First Generation (1945-55) Vacuum Tubes and Plugboards 1.2.2 The Second Generation (1955-65) Transistors and Batch Systems 1.2.3 The Third Generation (1965-1980) ICs and Multiprogramming 1.2.4 The Fourth Generation (1980-Present) Personal Computers 1.2.5 Ontogeny Recapitulates Phytogeny

1.3 THE OPERATING SYSTEM ZOO

1.3.1 Mainframe Operating Systems

1.3.2 Server Operating Systems

1.3.3 Multiprocessor Operating Systems

1.3.4 Personal Computer Operating Systems

1.3.5 Real-Time Operating Systems

1.3.6 Embedded Operating Systems

1.3.7 Smart Card Operating Systems

1.4 COMPUTER HARDWARE REVIEW

1.4.1 Processors

1.4.2 Memory

1.4.3 I/O Devices

1.4.4 Buses

1.5 OPERATING SYSTEM CONCEPTS

1.5.1 Processes

1.5.3 Memory Management

1.5.4 Input/Output

1.5.5 Files

1.5.6 Security

1.5.8 Recycling of Concepts

1.6 SYSTEM CALLS

1.6.1 System Calls for Process Management 1.6.2 System Calls for File Management

1.6.3 System Calls for Directory Management 1.6.4 Miscellaneous System Calls

1.6.5 The Windows Win32 API

1.7 OPERATING SYSTEM STRUCTURE 1.7.1 Monolithic Systems

1.7.2 Layered Systems

1.7.3 Virtual Machines

1.7.4 Exokernels

1.7.5 Client-Server Model

1.8 RESEARCH ON OPERATING SYSTEMS 1.9 OUTLINE OF THE REST OF THIS BOOK 1.10 METRIC UNITS

1.11 SUMMARY

2 PROCESSES AND THREADS

2.1 PROCESSES

2.1.1 The Process Model

2.1.2 Process Creation

2.1.3 Process Termination

2.1.4 Process Hierarchies

2.1.5 Process States

2.1.6 Implementation of Processes

2.2 THREADS

2.2.1 The Thread Model

2.2.2 Thread Usage

2.2.3 Implementing Threads in User Space 2.2.4 Implementing Threads in the Kernel 2.2.5 Hybrid Implementations

2.2.6 Scheduler Activations

2.2.7 Pop-Up Threads

2.2.8 Making Single-Threaded Code Multithreaded

2.3 INTERPROCESS COMMUNICATION

2.3.1 Race Conditions

2.3.2 Critical Regions

2.3.3 Mutual Exclusion with Busy Waiting

Disabling Interrupts

Lock Variables

Strict Alternation

Peterson’s Solution

The TSL Instruction

2.3.4 Sleep and Wakeup

The Producer-Consumer Problem

2.3.5 Semaphores

Solving the Producer-Consumer Problem using Semaphores

2.3.6 Mutexes

2.3.7 Monitors

2.3.8 Message Passing

Design Issues for Message Passing Systems

The Producer-Consumer Problem with Message Passing 2.3.9 Barriers

2.4 CLASSICAL IPC PROBLEMS

2.4.1 The Dining Philosophers Problem

2.4.2 The Readers and Writers Problem

2.4.3 The Sleeping Barber Problem

2.5 SCHEDULING

2.5.1 Introduction to Scheduling

Process Behavior

When to Schedule

Categories of Scheduling Algorithms

Scheduling Algorithm Goals

2.5.2 Scheduling in Batch Systems

First-Come First-Served

Shortest Job First

Shortest Remaining Time Next

Three-Level Scheduling

2.5.3 Scheduling in Interactive Systems

Round-Robin Scheduling

Priority Scheduling

Multiple Queues

Shortest Process Next

Guaranteed Scheduling

Lottery Scheduling

Fair-Share Scheduling

2.5.4 Scheduling in Real-Time Systems

2.5.5 Policy versus Mechanism

2.5.6 Thread Scheduling

2.6 RESEARCH ON PROCESSES AND THREADS

2.7 SUMMARY

3 DEADLOCKS

3.1 RESOURCES

3.1.1 Preemptable and Nonpreemptable Resources

3.1.2 Resource Acquisition

3.2 INTRODUCTION TO DEADLOCKS

3.2.1 Conditions for Deadlock

3.2.2 Deadlock Modeling

3.3 THE OSTRICH ALGORITHM

3.4 DEADLOCK DETECTION AND RECOVERY

3.4.1 Deadlock Detection with One Resource of Each Type 3.4.2 Deadlock Detection with Multiple Resource of Each Type 3.4.3 Recovery from Deadlock

Recovery through Preemption

Recovery through Rollback

Recovery through Killing Processes

3.5 DEADLOCK AVOIDANCE

3.5.1 Resource Trajectories

3.5.2 Safe and Unsafe States

3.5.3 The Banker’s Algorithm for a Single Resource 3.5.4 The Banker’s Algorithm for Multiple Resources

3.6 DEADLOCK PREVENTION

3.6.1 Attacking the Mutual Exclusion Condition

3.6.2 Attacking the Hold and Wait Condition

3.6.3 Attacking the No Preemption Condition

3.6.4 Attacking the Circular Wait Condition

3.7 OTHER ISSUES

3.7.1 Two-Phase Locking

3.7.2 Nonresource Deadlocks

3.7.3 Starvation

3.8 RESEARCH ON DEADLOCKS

3.9 SUMMARY

4 MEMORY MANAGEMENT

4.1 BASIC MEMORY MANAGEMENT

4.1.1 Monoprogramming without Swapping or Paging 4.1.2 Multiprogramming with Fixed Partitions

4.1.3 Modeling Multiprogramming

4.1.4 Analysis of Multiprogramming System Performance 4.1.5 Relocation and Protection

4.2 SWAPPING

4.2.1 Memory Management with Bitmaps

4.2.2 Memory Management with Linked Lists

4.3 VIRTUAL MEMORY

4.3.1 Paging

4.3.2 Page Tables

Multilevel Page Tables

Structure of a Page Table Entry

4.3.3 TLBs—Translation Lookaside Buffers

Software TLB Management

4.3.4 Inverted Page Tables

4.4 PAGE REPLACEMENT ALGORITHMS

4.4.1 The Optimal Page Replacement Algorithm

4.4.2 The Not Recently Used Page Replacement Algorithm 4.4.3 The First-In, First-Out (FIFO) Page Replacement Algorithm 4.4.4 The Second Chance Page Replacement Algorithm 4.4.5 The Clock Page Replacement Algorithm

4.4.6 The Least Recently Used (LRU) Page Replacement Algorithm 4.4.7 Simulating LRU in Software

4.4.8 The Working Set Page Replacement Algorithm

4.4.9 The WSClock Page Replacement Algorithm

4.4.10 Summary of Page Replacement Algorithms

4.5 MODELING PAGE REPLACEMENT ALGORITHMS 4.5.1 Belady’s Anomaly

4.5.2 Stack Algorithms

4.5.3 The Distance String

4.5.4 Predicting Page Fault Rates

4.6 DESIGN ISSUES FOR PAGING SYSTEMS

4.6.1 Local versus Global Allocation Policies

4.6.2 Load Control

4.6.3 Page Size

4.6.4 Separate Instruction and Data Spaces

4.6.5 Shared Pages

4.6.6 Cleaning Policy

4.6.7 Virtual Memory Interface

4.7 IMPLEMENTATION ISSUES

4.7.1 Operating System Involvement with Paging

4.7.2 Page Fault Handling

4.7.3 Instruction Backup

4.7.4 Locking Pages in Memory

4.7.5 Backing Store

4.7.6 Separation of Policy and Mechanism

4.8 SEGMENTATION

4.8.1 Implementation of Pure Segmentation

4.8.2 Segmentation with Paging: MULTICS

4.8.3 Segmentation with Paging: The Intel Pentium

4.9 RESEARCH ON MEMORY MANAGEMENT 4.10 SUMMARY

5 INPUT/OUTPUT

5.1 PRINCIPLES OF I/O HARDWARE

5.1.1 I/O Devices

5.1.2 Device Controllers

5.1.3 Memory-Mapped I/O

5.1.4 Direct Memory Access (DMA)

5.1.5 Interrupts Revisited

5.2 PRINCIPLES OF I/O SOFTWARE

5.2.1 Goals of the I/O Software

5.2.2 Programmed I/O

5.2.3 Interrupt-Driven I/O

5.2.4 I/O Using DMA

5.3 I/O SOFTWARE LAYERS

5.3.1 Interrupt Handlers

5.3.2 Device Drivers

5.3.3 Device-Independent I/O Software

Uniform Interfacing for Device Drivers

Buffering

Error Reporting

Allocating and Releasing Dedicated Devices

Device-Independent Block Size

5.3.4 User-Space I/O Software

5.4 DISKS

5.4.1 Disk Hardware

Magnetic Disks

RAID

CD-ROMs

CD-Recordables

CD-Rewritables

DVD

5.4.2 Disk Formatting

5.4.3 Disk Arm Scheduling Algorithms

5.4.4 Error Handling

5.4.5 Stable Storage

5.5 CLOCKS

5.5.1 Clock Hardware

5.5.2 Clock Software

5.5.3 Soft Timers

5.6 CHARACTER-ORIENTED TERMINALS

5.6.1 RS-232 Terminal Hardware

5.6.2 Input Software

5.6.3 Output Software

5.7 GRAPHICAL USER INTERFACES

5.7.1 Personal Computer Keyboard, Mouse, and Display Hardware 5.7.2 Input Software

5.7.3 Output Software for Windows

BitMaps

Fonts

5.8 NETWORK TERMINALS

5.8.1 The X Window System

5.8.2 The SLIM Network Terminal

5.9 POWER MANAGEMENT

5.9.1 Hardware Issues

5.9.2 Operating System Issues

The Display

The Hard Disk

The CPU

The Memory

Wireless Communication

Thermal Management

Battery Management

Driver Interface

5.9.3 Degraded Operation

5.10 RESEARCH ON INPUT/OUTPUT

5.11 SUMMARY

6 FILE SYSTEMS

6.1 FILES

6.1.1 File Naming

6.1.2 File Structure

6.1.3 File Types

6.1.4 File Access

6.1.5 File Attributes

6.1.6 File Operations

6.1.7 An Example Program Using File System Calls 6.1.8 Memory-Mapped Files

6.2 DIRECTORIES

6.2.1 Single-Level Directory Systems

6.2.2 Two-level Directory Systems

6.2.3 Hierarchical Directory Systems

6.2.4 Path Names

6.2.5 Directory Operations

6.3 FILE SYSTEM IMPLEMENTATION

6.3.1 File System Layout

6.3.2 Implementing Files

Contiguous Allocation

Linked List Allocation

Linked List Allocation Using a Table in Memory I-nodes

6.3.3 Implementing Directories

6.3.4 Shared Files

6.3.5 Disk Space Management

Block Size

Keeping Track of Free Blocks

Disk Quotas

6.3.6 File System Reliability

Backups

File System Consistency

6.3.7 File System Performance

Caching

Block Read Ahead

Reducing Disk Arm Motion

6.3.8 Log-Structured File Systems

6.4 EXAMPLE FILE SYSTEMS

6.4.1 CD-ROM File Systems

The ISO 9660 File System

Rock Ridge Extensions

Joliet Extensions

6.4.2 The CP/M File System

6.4.3 The MS-DOS File System

6.4.4 The Windows 98 File System

6.4.5 The UNIX V7 File System

6.5 RESEARCH ON FILE SYSTEMS 6.6 SUMMARY

7 MULTIMEDIA OPERATING SYSTEMS 7.1 INTRODUCTION TO MULTIMEDIA 7.2 MULTIMEDIA FILES

7.2.1 Audio Encoding

7.2.2 Video Encoding

7.3 VIDEO COMPRESSION

7.3.1 The JPEG Standard

7.3.2 The MPEG Standard

7.4 MULTIMEDIA PROCESS SCHEDULING

7.4.1 Scheduling Homogeneous Processes

7.4.2 General Real-Time Scheduling

7.4.3 Rate Monotonic Scheduling

7.4.4 Earliest Deadline First Scheduling

7.5 MULTIMEDIA FILE SYSTEM PARADIGMS

7.5.1 VCR Control Functions

7.5.2 Near Video on Demand

7.5.3 Near Video on Demand with VCR Functions

7.6 FILE PLACEMENT

7.6.1 Placing a File on a Single Disk

7.6.2 Two Alternative File Organization Strategies

7.6.3 Placing Files for Near Video on Demand

7.6.4 Placing Multiple Files on a Single Disk

7.6.5 Placing Files on Multiple Disks

7.7 CACHING

7.7.1 Block Caching

7.7.2 File Caching

7.8 DISK SCHEDULING FOR MULTIMEDIA

7.8.1 Static Disk Scheduling

7.8.2 Dynamic Disk Scheduling

7.9 RESEARCH ON MULTIMEDIA

7.10 SUMMARY

8 MULTIPLE PROCESSOR SYSTEMS

8.1 MULTIPROCESSORS

8.1.1 Multiprocessor Hardware

UMA Bus-Based SMP Architectures

UMA Multiprocessors Using Crossbar Switches

UMA Multiprocessors Using Multistage Switching Networks NUMA Multiprocessors

8.1.2 Multiprocessor Operating System Types Each CPU Has Its Own Operating System

Master-Slave Multiprocessors

Symmetric Multiprocessors

8.1.3 Multiprocessor Synchronization

Spinning versus Switching

8.1.4 Multiprocessor Scheduling

Timesharing

Space Sharing

Gang Scheduling

8.2 MULTICOMPUTERS

8.2.1 Multicomputer Hardware

Interconnection Technology

Network Interfaces

8.2.2 Low-Level Communication Software

Node to Network Interface Communication

8.2.3 User-Level Communication Software

Send and Receive

Blocking versus Nonblocking Calls

8.2.4 Remote Procedure Call

Implementation Issues

8.2.5 Distributed Shared Memory

Replication

False Sharing

Achieving Sequential Consistency

8.2.6 Multicomputer Scheduling

8.2.7 Load Balancing

A Graph-Theoretic Deterministic Algorithm

A Sender-Initiated Distributed Heuristic Algorithm

A Receiver-Initialed Distributed Heuristic Algorithm A Bidding Algorithm

8.3 DISTRIBUTED SYSTEMS

8.3.1 Network Hardware

Ethernet

The Internet

8.3.2 Network Services and Protocols

Network Services

Network Protocols

8.3.3 Document-Based Middleware

8.3.4 File System-Based Middleware

Transfer Model

The Directory Hierarchy

Naming Transparency

Semantics of File Sharing

AFS

8.3.5 Shared Object-Based Middleware

CORBA

Globe

8.3.6 Coordination-Based Middleware

Linda

Publish/Subscribe

Jini

8.4 RESEARCH ON MULTIPLE PROCESSOR SYSTEMS 8.5 SUMMARY

9 SECURITY

9.1 THE SECURITY ENVIRONMENT

9.1.1 Threats

9.1.2 Intruders

9.1.3 Accidental Data Loss

9.2 BASICS OF CRYPTOGRAPHY

9.2.1 Secret-Key Cryptography

9.2.2 Public-Key Cryptography

9.2.3 One-Way Functions

9.2.4 Digital Signatures

9.3 USER AUTHENTICATION

9.3.1 Authentication Using Passwords How Crackers Break In

UNIX Password Security

Improving Password Security

One-Time Passwords

Challenge-Response Authentication

9.3.2 Authentication Using a Physical Object 9.3.3 Authentication Using Biometrics 9.3.4 Countermeasures

9.4 ATTACKS FROM INSIDE THE SYSTEM 9.4.1 Trojan Horses

9.4.2 Login Spoofing

9.4.3 Logic Bombs

9.4.4 Trap Doors

9.4.5 Buffer Overflow

9.4.6 Generic Security Attacks

9.4.7 Famous Security Flaws

Famous Security Flaws in UNIX

Famous Security Flaws in TENEX

Famous Security Flaws in OS/360

9.4.8 Design Principles for Security

9.5 ATTACKS FROM OUTSIDE THE SYSTEM 9.5.1 Virus Damage Scenarios

9.5.2 How Viruses Work

Companion Viruses

Executable Program Viruses

Memory Resident Viruses

Boot Sector Viruses

Device Driver Viruses

Macro Viruses

Source Code Viruses

9.5.3 How Viruses Spread

9.5.4 Antivirus and Anti-Antivirus Techniques Virus Scanners

Integrity Checkers

Behavioral Checkers

Virus Avoidance

Recovery from a Virus Attack

9.5.5 The Internet Worm

9.5.6 Mobile Code

Sandboxing

Interpretation

Code signing

9.5.7 Java Security

9.6 PROTECTION MECHANISMS

9.6.1 Protection Domains

9.6.2 Access Control Lists

9.6.3 Capabilities

9.7 TRUSTED SYSTEMS

9.7.1 Trusted Computing Base

9.7.2 Formal Models of Secure Systems 9.7.3 Multilevel Security

The Bell-La Padula Model

The Biba Model

9.7.4 Orange Book Security

9.7.5 Covert Channels

9.8 RESEARCH ON SECURITY

9.9 SUMMARY

10 CASE STUDY 1: UNIX AND LINUX

10.1 HISTORY OF UNIX

10.1.1 UNICS

10.1.2 PDP-11 UNIX

10.1.3 Portable UNIX

10.1.4 Berkeley UNIX

10.1.5 Standard UNIX

10.1.6 MINIX

10.1.7 Linux

10.2 OVERVIEW OF UNIX

10.2.1 UNIX Goals

10.2.2 Interfaces to UNIX

10.2.3 The UNIX Shell

10.2.4 UNIX Utility Programs

10.2.5 Kernel Structure

10.3 PROCESSES IN UNIX

10.3.1 Fundamental Concepts

10.3.2 Process Management System Calls in UNIX Thread Management System Calls

10.3.3 Implementation of Processes in UNIX

Threads in UNIX

Threads in Linux

Scheduling in UNIX

Scheduling in Linux

10.3.4 Booting UNIX

10.4 MEMORY MANAGEMENT IN UNIX

10.4.1 Fundamental Concepts

10.4.2 Memory Management System Calls in UNIX 10.4.3 Implementation of Memory Management in UNIX Swapping

Paging in UNIX

The Page Replacement Algorithm

Memory Management in Linux

10.5 INPUT/OUTPUT IN UNIX

10.5.1 Fundamental Concepts

Networking

10.5.2 Input/Output System Calls in UNIX

10.5.3 Implementation of Input/Output in UNIX 10.5.4 Streams

10.6 THE UNIX FILE SYSTEM

10.6.1 Fundamental Concepts

10.6.2 File System Calls in UNIX

10.6.3 Implementation of the UNIX File System The Berkeley Fast File System

The Linux File System

10.6.4 NFS: The Network File System

NFS Architecture

NFS Protocols

NFS Implementation

10.7 SECURITY IN UNIX

10.7.1 Fundamental Concepts

10.7.2 Security System Calls in UNIX

10.7.3 Implementation of Security in UNIX 10.8 SUMMARY

11 CASE STUDY 2: WINDOWS 2000

11.1 HISTORY OF WINDOWS 2000

11.1.1 MS-DOS

11.1.2 Windows 95/98/Me

11.1.3 Windows NT

11.1.4 Windows 2000

11.2 PROGRAMMING WINDOWS 2000

11.2.1 The Win32 Application Programming Interface 11.2.2 The Registry

11.3 SYSTEM STRUCTURE

11.3.1 Operating System Structure

The Hardware Abstraction Layer

The Kernel Layer

The Executive

The Device Drivers

11.3.2 Implementation of Objects

The Object Name Space

11.3.3 Environment Subsystems

11.4 PROCESSES AND THREADS IN WINDOWS 2000 11.4.1 Fundamental Concepts

11.4.2 Job, Process, Thread and Fiber Management API Calls Interprocess Communication

11.4.3 Implementation of Processes and Threads

Scheduling

11.4.4 MS-DOS Emulation

11.4.5 Booting Windows 2000

11.5 MEMORY MANAGEMENT

11.5.1 Fundamental Concepts

11.5.2 Memory Management System Calls

11.5.3 Implementation of Memory Management

Page Fault Handling

The Page Replacement Algorithm

Physical Memory Management

11.6 INPUT/OUTPUT IN WINDOWS 2000

11.6.1 Fundamental Concepts

11.6.2 Input/Output API Calls

11.6.3 Implementation of I/O

11.6.4 Device Drivers

11.7 THE WINDOWS 2000 FILE SYSTEM

11.7.1 Fundamental Concepts

11.7.2 File System API Calls in Windows 2000

11.7.3 Implementation of the Windows 2000 File System File System Structure

File Name Lookup

File Compression

File Encryption

11.8 SECURITY IN WINDOWS 2000

11.8.1 Fundamental Concepts

11.8.2 Security API Calls

11.8.3 Implementation of Security

11.9 CACHING IN WINDOWS 2000

11.10 SUMMARY

12 OPERATING SYSTEM DESIGN

12.1 THE NATURE OF THE DESIGN PROBLEM 12.1.1 Goals

12.1.2 Why is it Hard to Design an Operating System?

12.2 INTERFACE DESIGN

12.2.1 Guiding Principles

Principle 1: Simplicity

Principle 2: Completeness

Principle 3: Efficiency

12.2.2 Paradigms

User Interface Paradigms

Execution Paradigms

Data Paradigms

12.2.3 The System Call Interface

12.3 IMPLEMENTATION

12.3.1 System Structure

Layered Systems

Exokernels

Client-Server Systems

Extensible Systems

Kernel Threads

12.3.2 Mechanism versus Policy

12.3.3 Orthogonality

12.3.4 Naming

12.3.5 Binding Time

12.3.6 Static versus Dynamic Structures

12.3.7 Top-Down versus Bottom-Up Implementation 12.3.8 Useful Techniques

Hiding the Hardware

Indirection

Reusability

Reentrancy

Brute Force

Check for Errors First

12.4 PERFORMANCE

12.4.1 Why Are Operating Systems Slow?

12.4.2 What Should Be Optimized?

12.4.3 Space-Time Trade-offs

12.4.4 Caching

12.4.5 Hints

12.4.6 Exploiting Locality

12.4.7 Optimize the Common Case

12.5 PROJECT MANAGEMENT

12.5.1 The Mythical Man Month

12.5.2 Team Structure

12.5.3 The Role of Experience

12.5.4 No Silver Bullet

12.6 TRENDS IN OPERATING SYSTEM DESIGN

12.6.1 Large Address Space Operating Systems 12.6.2 Networking

12.6.3 Parallel and Distributed Systems

12.6.4 Multimedia

12.6.5 Battery-Powered Computers

12.6.6 Embedded Systems

12.7 SUMMARY

13 READING LIST AND BIBLIOGRAPHY 13.1 SUGGESTIONS FOR FURTHER READING 13.1.1 Introduction and General Works

13.1.2 Processes and Threads

13.1.3 Deadlocks

13.1.4 Memory Management

13.1.5 Input/Output

13.1.6 File Systems

13.1.7 Multimedia Operating Systems

13.1.8 Multiple Processor Systems

13.1.9 Security

13.1.10 UNIX and Linux

13.1.11 Windows 2000

13.1.12 Design Principles

13.2 ALPHABETICAL BIBLIOGRAPHY ABOUT THE AUTHOR

[Next]

**PREFACE**

The world has changed a great deal since the first edition of this book appeared in 1992. Computer networks and distributed systems of all kinds have become very common. Small children now roam the Internet, where previously only computer professionals went. As a consequence, this book has changed a great deal, too.

The most obvious change is that the first edition was about half on single-processor operating systems and half on distributed systems. I chose that format in 1991 because few universities then had courses on distributed systems and whatever students learned about distributed systems had to be put into the operating systems course, for which this book was intended. Now most universities have a separate course on distributed systems, so it is not necessary to try to combine the two subjects into one course and one book. This book is intended for a first course on operating systems, and as such focuses mostly on traditional single-processor systems.

I have coauthored two other books on operating systems. This leads to two possible course sequences.

Practically-oriented sequence:

1. Operating Systems Design and Implementation by Tanenbaum and Woodhull 2. Distributed Systems by Tanenbaum and Van Steen

Traditional sequence:

1. Modern Operating Systems by Tanenbaum

2. Distributed Systems by Tanenbaum and Van Steen

The former sequence uses MINIX and the students are expected to experiment with MINIX in an accompanying laboratory supplementing the first course. The latter sequence does not use MINIX. Instead, some small simulators are available that can be used for student exercises during a first course using this book. These simulators can be found starting on the author's Web page: *www.cs.vu.nl/~ast/* by clicking on

Software and supplementary material for my books.

In addition to the major change of switching the emphasis to single-processor operating systems in this book, other major changes include the addition of entire chapters on computer security, multimedia operating systems, and Windows 2000, all important and timely topics. In addition, a new and unique chapter on operating system design has been added.

Another new feature is that many chapters now have a section on research about the topic of the chapter. This is intended to introduce the reader to modern work in processes, memory management, and so on. These sections have numerous references to the current research literature for the interested reader. In addition, Chapter 13 has many introductory and tutorial references.

Finally, numerous topics have been added to this book or heavily revised. These topics include: graphical user interfaces, multiprocessor operating systems, power management for laptops, trusted systems, viruses, network terminals, CD-ROM file systems, mutexes, RAID, soft timers, stable storage, fair-share scheduling, and new paging algorithms. Many new problems have been added and old ones updated. The total number of problems now exceeds 450. A solutions manual is available to professors using this book in a course. They can obtain a copy from their local Prentice Hall representative. In addition, over 250 new references to the current literature have been added to bring the book up to date.

Despite the removal of more than 400 pages of old material, the book has increased in size due to the large amount of new material added. While the book is still suitable for a one-semester or two-quarter course, it is probably too long for a one-quarter or one trimester course at most universities. For this reason, the book has been designed in a modular way. Any course on operating systems should cover chapters 1 through 6. This is basic material that every student show know.

If additional time is available, additional chapters can be covered. Each of them assumes the reader has finished chapters 1 through 6, but Chaps. 7 through 12 are each self contained, so any desired subset can be used and in any order, depending on the interests of the instructor. In the author's opinion, Chaps. 7 through 12 are much more interesting than the earlier ones. Instructors should tell their students that they have to eat their broccoli before they can have the double chocolate fudge cake dessert.

I would like to thank the following people for their help in reviewing parts of the manuscript: Rida Bazzi, Riccardo Bettati, Felipe Cabrera, Richard Chapman, John Connely, John Dickinson, John Elliott, Deborah Frincke, Chandana Gamage, Robbert Geist, David Golds, Jim Griffioen, Gary Harkin, Frans Kaashoek, Mukkai Krishnamoorthy, Monica Lam, Jussi Leiwo, Herb Mayer, Kirk McKusick, Evi Nemeth, Bill Potvin, Prasant Shenoy, Thomas Skinner, Xian-He Sun, William Terry, Robbert Van Renesse, and Maarten van Steen. Jamie Hanrahan, Mark Russinovich, and Dave Solomon were enormously knowledgeable about Windows 2000 and very helpful. Special thanks go to Al Woodhull for valuable reviews and thinking of many new end-of-chapter problems.

My students were also helpful with comments and feedback, especially Staas de Jong, Jan de Vos, Niels Drost, David Fokkema, Auke Folkerts, Peter Groenewegen, Wilco Ibes, Stefan Jansen, Jeroen Ketema, Joeri Mulder, Irwin Oppenheim, Stef Post, Umar Rehman, Daniel Rijkhof, Maarten Sander, Maurits van der Schee, Rik van der Stoel, Mark van Driel, Dennis van Veen, and Thomas Zeeman.

Barbara and Marvin are still wonderful, as usual, each in a unique way. Finally, last but not least, I would like to thank Suzanne for her love and patience, not to mention all the *druiven* and *kersen,* which have replaced the *sinasappelsap* in recent times.

Andrew S. Tanenbaum

[Previous] [Next]

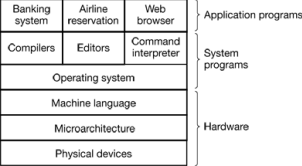
**1**

**INTRODUCTION**

A modern computer system consists of one or more processors, some main memory, disks, printers, a keyboard, a display, network interfaces, and other input/output devices. All in all, a complex system. Writing programs that keep track of all these components and use them correctly, let alone optimally, is an extremely difficult job. For this reason, computers are equipped with a layer of software called the **operating system,** whose job is to manage all these devices and provide user programs with a simpler interface to the hardware. These systems are the subject of this book.

The placement of the operating system is shown in Fig. 1-1. At the bottom is the hardware, which, in many cases, is itself composed of two or more levels (or layers). The lowest level contains physical devices, consisting of integrated circuit chips, wires, power supplies, cathode ray tubes, and similar physical devices. How these are constructed and how they work are the provinces of the electrical engineer.

Next comes the **microarchitecture level,** in which the physical devices are grouped together to form functional units. Typically this level contains some registers internal to the CPU (Central Processing Unit) and a data path containing an arithmetic logic unit. In each clock cycle, one or two operands are fetched from the registers and combined in the arithmetic logic unit (for example, by addition or Boolean AND). The result is stored in one or more registers. On some machines, the operation of the data path is controlled by software, called the **microprogram.** On other machines, it is controlled directly by hardware circuits.



**Figure 1-1.** A computer system consists of hardware, system programs, and application programs.

The purpose of the data path is to execute some set of instructions. Some of these can be carried out in one data path cycle; others may require multiple data path cycles. These instructions may use registers or other hardware facilities. Together, the hardware and instructions visible to an assembly language programmer form the **ISA (Instruction Set Architecture)** level. This level is often called **machine language.**

The machine language typically has between 50 and 300 instructions, mostly for moving data around the machine, doing arithmetic, and comparing values. In this level, the input/output devices are controlled by loading values into special **device registers.** For example, a disk can be commanded to read by loading the values of the disk address, main memory address, byte count, and direction (read or write) into its registers. In practice, many more parameters are needed, and the status returned by the drive after an operation is highly complex. Furthermore, for many I/O (Input/Output) devices, timing plays an important role in the programming.

To hide this complexity, an operating system is provided. It consists of a layer of software that (partially) hides the hardware and gives the programmer a more convenient set of instructions to work with. For example, read block from file is conceptually simpler than having to worry about the details of moving disk heads, waiting for them to settle down, and so on.

On top of the operating system is the rest of the system software. Here we find the command interpreter (shell), window systems, compilers, editors, and similar application-independent programs. It is important to realize that these programs are definitely not part of the operating system, even though they are typically supplied by

the computer manufacturer. This is a crucial, but subtle, point. The operating system is (usually) that portion of the software that runs in **kernel mode** or **supervisor mode.** It is protected from user tampering by the hardware (ignoring for the moment some older or low-end microprocessors that do not have hardware protection at all). Compilers and editors run in **user mode.** If a user does not like a particular compiler, he[\*] is free to write his own if he so chooses: he is not free to write his own clock interrupt handler, which is part of the operating system and is normally protected by hardware against attempts by users to modify it.

This distinction, however, is sometimes blurred in embedded systems (which may not have kernel mode) or interpreted systems (such as Java-based operating systems that use interpretation, not hardware, to separate the components). Still, for traditional computers, the operating system is what runs in kernel mode.

That said, in many systems there are programs that run in user mode but which help the operating system or perform privileged functions. For example, there is often a program that allows users to change their passwords. This program is not part of the operating system and does not run in kernel mode, but it clearly carries out a sensitive function and has to be protected in a special way.

In some systems, this idea is carried to an extreme form, and pieces of what is traditionally considered to be the operating system (such as the file system) run in user space. In such systems, it is difficult to draw a clear boundary. Everything running in kernel mode is clearly part of the operating system, but some programs running outside it are arguably also part of it, or at least closely associated with it.

Finally, above the system programs come the application programs. These programs are purchased or written by the users to solve their particular problems, such as word processing, spreadsheets, engineering calculations, or storing information in a database.

**1.1 WHAT IS AN OPERATING SYSTEM?**

Most computer users have had some experience with an operating system, but it is difficult to pin down precisely what an operating system is. Part of the problem is that operating systems perform two basically unrelated functions, extending the machine and managing resources, and depending on who is doing the talking, you hear mostly about one function or the other. Let us now look at both.

**1.1.1 The Operating System as an Extended Machine**

As mentioned earlier, the **architecture** (instruction set, memory organization, I/O, and bus structure) of most computers at the machine language level is primitive and awkward to program, especially for input/output. To make this point more concrete, let us briefly look at how floppy disk I/O is done using the NEC PD765 compatible controller chips used on most Intel-based personal computers. (Throughout this book we will use the terms “floppy disk” and “diskette” interchangeably.) The PD765 has 16 commands, each specified by loading between 1 and 9 bytes into a device register. These commands are for reading and writing data, moving the disk arm, and formatting tracks, as well as initializing, sensing, resetting, and recalibrating the controller and the drives.

The most basic commands are read and write, each of which requires 13 parameters, packed into 9 bytes. These parameters specify such items as the address of the disk block to be read, the number of sectors per track, the recording mode used on the physical medium, the intersector gap spacing, and what to do with a deleted-data address-mark. If you do not understand this mumbo jumbo, do not worry; that is precisely the point—it is rather esoteric. When the operation is completed, the controller chip returns 23 status and error fields packed into 7 bytes. As if this were not enough, the floppy disk programmer must also be constantly aware of whether the motor is on or off. If the motor is off, it must be turned on (with a long startup delay) before data can be read or written. The motor cannot be left on too long, however, or the floppy disk will wear out. The programmer is thus forced to deal with the trade-off between long startup delays versus wearing out floppy disks (and losing the data on them).

Without going into the *real* details, it should be clear that the average programmer probably does not want to get too intimately involved with the programming of floppy disks (or hard disks, which are just as complex and quite different). Instead, what the programmer wants is a simple, high-level abstraction to deal with. In the case of disks, a typical abstraction would be that the disk contains a collection of named files. Each file can be opened for reading or writing, then read or written, and finally closed. Details such as whether or not recording should use modified frequency modulation and what the current state of the motor is should not appear in the abstraction presented to the user.

The program that hides the truth about the hardware from the programmer and presents

a nice, simple view of named files that can be read and written is, of course, the operating system. Just as the operating system shields the programmer from the disk hardware and presents a simple file-oriented interface, it also conceals a lot of unpleasant business concerning interrupts, timers, memory management, and other low level features. In each case, the abstraction offered by the operating system is simpler and easier to use than that offered by the underlying hardware.

In this view, the function of the operating system is to present the user with the equivalent of an **extended machine** or **virtual machine** that is easier to program than the underlying hardware. How the operating system achieves this goal is a long story, which we will study in detail throughout this book. To summarize it in a nutshell, the operating system provides a variety of services that programs can obtain using special instructions called system calls. We will examine some of the more common system calls later in this chapter.

**1.1.2 The Operating System as a Resource Manager**

The concept of the operating system as primarily providing its users with a convenient interface is a top-down view. An alternative, bottom-up, view holds that the operating system is there to manage all the pieces of a complex system. Modern computers consist of processors, memories, timers, disks, mice, network interfaces, printers, and a wide variety of other devices. In the alternative view, the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs competing for them.

Imagine what would happen if three programs running on some computer all tried to print their output simultaneously on the same printer. The first few lines of printout might be from program 1, the next few from program 2, then some from program 3, and so forth. The result would be chaos. The operating system can bring order to the potential chaos by buffering all the output destined for the printer on the disk. When one program is finished, the operating system can then copy its output from the disk file where it has been stored to the printer, while at the same time the other program can continue generating more output, oblivious to the fact that the output is not really going to the printer (yet).

When a computer (or network) has multiple users, the need for managing and protecting the memory, I/O devices, and other resources is even greater, since the users might otherwise interfere with one another. In addition, users often need to share not only

hardware, but information (files, databases, etc.) as well. In short, this view of the operating system holds that its primary task is to keep track of who is using which resource, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users.

Resource management includes multiplexing (sharing) resources in two ways: in time and in space. When a resource is time multiplexed, different programs or users take turns using it. First one of them gets to use the resource, then another, and so on. For example, with only one CPU and multiple programs that want to run on it, the operating system first allocates the CPU to one program, then after it has run long enough, another one gets to use the CPU, then another, and then eventually the first one again. Determining how the resource is time multiplexed — who goes next and for how long — is the task of the operating system. Another example of time multiplexing is sharing the printer. When multiple print jobs are queued up for printing on a single printer, a decision has to be made about which one is to be printed next.

The other kind of multiplexing is space multiplexing, instead of the customers taking turns, each one gets part of the resource. For example, main memory is normally divided up among several running programs, so each one can be resident at the same time (for example, in order to take turns using the CPU). Assuming there is enough memory to hold multiple programs, it is more efficient to hold several programs in memory at once rather than give one of them all of it, especially if it only needs a small fraction of the total. Of course, this raises issues of fairness, protection, and so on, and it is up to the operating system to solve them. Another resource that is space multiplexed is the (hard) disk. In many systems a single disk can hold files from many users at the same time. Allocating disk space and keeping track of who is using which disk blocks is a typical operating system resource management task.

**1.2 HISTORY OF OPERATING SYSTEMS**

Operating systems have been evolving through the years. In the following sections we will briefly look at a few of the highlights. Since operating systems have historically been closely tied to the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like. This mapping of operating system generations to computer generations is crude, but it does provide some structure where there would otherwise be none.

The first true digital computer was designed by the English mathematician Charles Babbage (1792-1871). Although Babbage spent most of his life and fortune trying to build his “analytical engine.” he never got it working properly because it was purely mechanical, and the technology of his day could not produce the required wheels, gears, and cogs to the high precision that he needed. Needless to say, the analytical engine did not have an operating system.

As an interesting historical aside, Babbage realized that he would need software for his analytical engine, so he hired a young woman named Ada Lovelace, who was the daughter of the famed British poet Lord Byron, as the world’s first programmer. The programming language Ada is named after her.

**1.2.1 The First Generation (1945-55) Vacuum Tubes and Plugboards**

After Babbage’s unsuccessful efforts, little progress was made in constructing digital computers until World War II. Around the mid-1940s, Howard Aiken at Harvard, John von Neumann at the Institute for Advanced Study in Princeton, J. Presper Eckert and William Mauchley at the University of Pennsylvania, and Konrad Zuse in Germany, among others, all succeeded in building calculating engines. The first ones used mechanical relays but were very slow, with cycle times measured in seconds. Relays were later replaced by vacuum tubes. These machines were enormous, filling up entire rooms with tens of thousands of vacuum tubes, but they were still millions of times slower than even the cheapest personal computers available today.

In these early days, a single group of people designed, built, programmed, operated, and maintained each machine. All programming was done in absolute machine language, often by wiring up plugboards to control the machine’s basic functions. Programming languages were unknown (even assembly language was unknown). Operating systems were unheard of. The usual made of operation was for the programmer to sign up for a block of time on the signup sheet on the wall, then come down to the machine room, insert his or her plugboard into the computer, and spend the next few hours hoping that none of the 20,000 or so vacuum tubes would burn out during the run. Virtually all the problems were straightforward numerical calculations, such as grinding out tables of sines, cosines, and logarithms.

By the early 1950s, the routine had improved somewhat with the introduction of

punched cards. It was now possible to write programs on cards and read them in instead of using plugboards; otherwise, the procedure was the same,

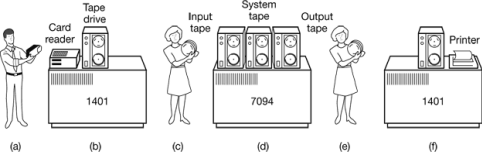
**1.2.2 The Second Generation (1955-65) Transistors and Batch Systems**

The introduction of the transistor in the mid-1950s changed the picture radically. Computers became reliable enough that they could be manufactured and sold to paying customers with the expectation that they would continue to function long enough to get some useful work done. For the first time, there was a clear separation between designers, builders, operators, programmers, and maintenance personnel.

These machines, now called **mainframes,** were locked away in specially air conditioned computer rooms, with staffs of professional operators to run them. Only big corporations or major government agencies or universities could afford the multimillion dollar price tag. To run a **job** (i.e., a program or set of programs), a programmer would first write the program on paper (in FORTRAN or assembler), then punch it on cards. He would then bring the card deck down to the input room and hand it to one of the operators and go drink coffee until the output was ready.

When the computer finished whatever job it was currently running, an operator would go over to the printer and tear off the output and carry it over to the output room, so that the programmer could collect it later. Then he would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in. Much computer time was wasted while operators were walking around the machine room.

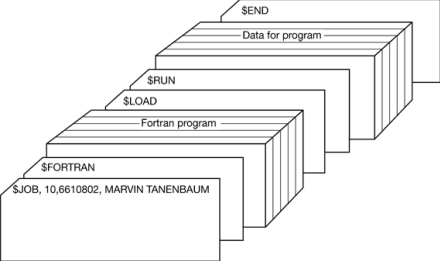
Given the high cost of the equipment, it is not surprising that people quickly looked for ways to reduce the wasted time. The solution generally adopted was the **batch system.** The idea behind it was to collect a tray full of jobs in the input room and then read them onto a magnetic tape using a small (relatively) inexpensive computer, such as the IBM 1401, which was very good at reading cards, copying tapes, and printing output, but not at all good at numerical calculations. Other, much more expensive machines, such as the IBM 7094, were used for the real computing. This situation is shown in Fig. 1-2.

**Figure 1-2.** An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

After about an hour of collecting a butch of jobs, the tape was rewound and brought into the machine room, where it was mounted on a tape drive. The operator then loaded a special program (the ancestor of today’s operating system), which read the first job from tape and ran it. The output was written onto a second tape, instead of being printed. After each job finished, the operating system automatically read the next job from the tape and began running it. When the whole batch was done, the operator removed the input and output tapes, replaced the input tape with the next batch, and brought the output tape to a 1401 for printing **offline** (i.e., not connected to the main computer).

The structure of a typical input job is shown in Fig. 1-3. It started out with a $JOB card, specifying the maximum run time in minutes, the account number to be charged, and the programmer’s name. Then came a $FORTRAN card, telling the operating system to load the FORTRAN compiler from the system tape. It was followed by the program to be compiled, and then a $LOAD card, directing the operating system to load the object program just compiled. (Compiled programs were often written on scratch tapes and had to be loaded explicitly.) Next came the $RUN card, telling the operating system to run the program with the data following it. Finally, the $END card marked the end of the job. These primitive control cards were the forerunners of modern job control languages and command interpreters.

Large second-generation computers were used mostly for scientific and engineering calculations, such as solving the partial differential equations that often occur in physics and engineering. They were largely programmed in FORTRAN and assembly language. Typical operating systems were FMS (the Fortran Monitor System) and IBSYS, IBM’s operating system for the 7094.

**Figure 1-3.** Structure of a typical FMS job.

**1.2.3 The Third Generation (1965-1980) ICs and**

**Multiprogramming**

By the early 1960s, most computer manufacturers had two distinct, and totally incompatible, product lines. On the one hand there were the word-oriented, large-scale scientific computers, such as the 7094, which were used for numerical calculations in science and engineering. On the other hand, there were the character-oriented, commercial computers, such as the 1401, which were widely used for tape sorting and printing by banks and insurance companies.

Developing and maintaining two completely different product lines was an expensive proposition for the manufacturers. In addition, many new computer customers initially needed a small machine but later outgrew it and wanted a bigger machine that would run all their old programs, but faster.

IBM attempted to solve both of these problems at a single stroke by introducing the System/360. The 360 was a series of software-compatible machines ranging from 1401- sized to much more powerful than the 7094. The machines differed only in price and performance (maximum memory, processor speed, number of I/O devices permitted,

and so forth). Since all the machines had the same architecture and instruction set, programs written for one machine could run on all the others, at least in theory. Furthermore, the 360 was designed to handle both scientific (i.e., numerical) and commercial computing. Thus a single family of machines could satisfy the needs of all customers. In subsequent years, IBM has come out with compatible successors to the 360 line, using more modern technology, known as the 370, 4300, 3080, and 3090 series.

The 360 was the first major computer line to use (small-scale) Integrated Circuits (ICs), thus providing a major price/performance advantage over the second-generation machines, which were built up from individual transistors. It was an immediate success, and the idea of a family of compatible computers was soon adopted by all the other major manufacturers. The descendants of these machines are still in use at computer centers today. Nowadays they are often used for managing huge databases (e.g., for airline reservation systems) or as servers for World Wide Web sites that must process thousands of requests per second.

The greatest strength of the “one family” idea was simultaneously its greatest weakness. The intention was that all software, including the operating system, OS/360 had to work on all models. It had to run on small systems, which often just replaced 1401s for copying cards to tape, and on very large systems, which often replaced 7094s for doing weather forecasting and other heavy computing. It had to be good on systems with few peripherals and on systems with many peripherals. It had to work in commercial environments and in scientific environments. Above all, it had to be efficient for all of these different uses.

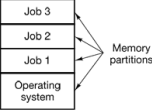
There was no way that IBM (or anybody else) could write a piece of software to meet all those conflicting requirements. The result was an enormous and extraordinarily complex operating system, probably two to three orders of magnitude larger than FMS. It consisted of millions of lines of assembly language written by thousands of programmers, and contained thousands upon thousands of bugs, which necessitated a continuous stream of new releases in an attempt to correct them. Each new release fixed some bugs and introduced new ones, so the number of bugs probably remained constant in time.

One of the designers of OS/360. Fred Brooks, subsequently wrote a witty and incisive book (Brooks, 1996) describing his experiences with OS/360. While it would be impossible to summarize the book here, suffice it to say that the cover shows a herd of

prehistoric beasts stuck in a tar pit. The cover of Silberschatz et al. (2000) makes a similar point about operating systems being dinosaurs.

Despite its enormous size and problems, OS/360 and the similar third-generation operating systems produced by other computer manufacturers actually satisfied most of their customers reasonably well. They also popularized several key techniques absent in second-generation operating systems. Probably the most important of these was **multiprogramming.** On the 7094, when the current job paused to wait for a tape or other I/O operation to complete, the CPU simply sat idle until the I/O finished. With heavily CPU-bound scientific calculations, I/O is infrequent, so this wasted time is not significant. With commercial data processing, the I/O wait time can often be 80 or 90 percent of the total time, so something had to be done to avoid having the (expensive) CPU be idle so much.

The solution that evolved was to partition memory into several pieces with a different job in each partition, as shown in Fig. 1-4. While one job was waiting for I/O to complete, another job could be using the CPU. If enough jobs could be held main memory at once, the CPU could be kept busy nearly 100 percent of the time. Having multiple jobs safely in memory at once requires special hardware to protect each job against snooping and mischief by the other ones, but the 360 and other third-generation systems were equipped with this hardware.



**Figure 1-4.** A multiprogramming system with three jobs in memory.

Another major feature present in third-generation operating systems was the ability to read jobs from cards onto the disk as soon as they were brought to the computer room. Then, whenever a running job finished, the operating system could load a new job from the disk into the now-empty partition and run it. This technique is called **spooling** (from Simultaneous Peripheral Operation On Line) and was also used for output. With spooling, the 1401s were no longer needed, and much carrying of tapes disappeared.

Although third-generation operating systems were well suited for big scientific

calculations and massive commercial data processing runs, they were still basically batch systems. Many programmers pined for the first-generation days when they had the machine all to themselves for a few hours, so they could debug their programs quickly. With third-generation systems, the time between submitting a job and getting back the output was often several hours, so a single misplaced comma could cause a compilation to fail, and the programmer to waste half a day.

This desire for quick response time paved the way for **timesharing,** a variant of multiprogramming, in which each user has an online terminal. In a timesharing system, if 20 users are logged in and 17 of them are thinking or talking or drinking coffee, the CPU can be allocated in turn to the three jobs that want service. Since people debugging programs usually issue short commands (e.g., compile a five-page procedure[\*] ) rather than long ones (e.g., sort a million-record file), the computer can provide fast, interactive service to a number of users and perhaps also work on big batch jobs in the background when the CPU is otherwise idle. The first serious timesharing system, **CTSS (Compatible Time Sharing System),** was developed at **M.I.T.** on a specially modified 7094 (Corbat et al., 1962). However, timesharing did not really become popular until the necessary protection hardware became widespread during the third generation

After the success of the CTSS system, MIT, Bell Labs, and General Electric (then a major computer manufacturer) decided to embark on the development of a “computer utility,” a machine that would support hundreds of simultaneous timesharing users. Their model was the electricity distribution system — when you need electric power, you just stick a plug in the wall, and within reason, as much power as you need will be there. The designers of this system, known as **MULTICS** (**MULTiplexed Information and Computing Service),** envisioned one huge machine providing computing power for everyone in the Boston area. The idea that machines far more powerful than their GE-645 mainframe would be sold for a thousand dollars by the millions only 30 years later was pure science fiction. Sort of like the idea of supersonic trans-Atlantic undersea trains now.

MULTICS was a mixed success. It was designed to support hundreds of users on a machine only slightly more powerful than an Intel 386-based PC, although it had much more I/O capacity. This is not quite as crazy as it sounds, since people knew how to write small, efficient programs in those days, a skill that has subsequently been lost. There were many reasons that MULTICS did not take over the world, not the least of which is that it was written in PL/I, and the PL/I compiler was years late and barely

worked at all when it finally arrived. In addition, MULTICS was enormously ambitious for its time, much like Charles Babbage’s analytical engine in the nineteenth century.

To make a long story short, MULTICS introduced many seminal ideas into the computer literature, but turning it into a serious product and a major commercial success was a lot harder than anyone had expected. Bell Labs dropped out of the project, and General Electric quit the computer business altogether. However, M.I.T. persisted and eventually got MULTICS working. It was ultimately sold as a commercial product by the company that bought GE’s computer business (Honeywell) and installed by about 80 major companies and universities worldwide. While their numbers were small, MULTICS users were fiercely loyal. General Motors, Ford, and the U.S. National Security Agency, for example, only shut down their MULTICS systems in the late 1990s, 30 years after MULTICS was released.

For the moment, the concept of a computer utility has fizzled out but it may well come back in the form of massive centralized Internet servers to which relatively dumb user machines are attached, with most of the work happening on the big servers. The motivation here is likely to be that most people do not want to administrate an increasingly complex and finicky computer system and would prefer to have that work done by a team of professionals working for the company running the server. E commerce is already evolving in this direction, with various companies running e-malls on multiprocessor servers to which simple client machines connect, very much in the spirit of the MULTICS design.

Despite its lack of commercial success, MULTICS had a huge influence on subsequent operating systems. lt is described in (Corbat et al,, 1972; Corbat and Vyssotsky, 1965; Daley and Dennis, 1968; Organick, 1972; and Saltzer, 1974). It also has a still-active Web sit*www.multicians.org* , with a great deal of information about the system, its designers, and its users.

Another major development during the third generation was the phenomenal growth of minicomputers, starting with the DEC PDP-1 in 1961. The PDP-1 had only 4K of 18-bit words, but at $120,000 per machine (less than 5 percent of the price of a 7094), it sold like hotcakes. For certain kinds of nonnumerical work, it was almost as fast as the 7094 and gave birth to a whole new industry. It was quickly followed by a series of other PDPs (unlike IBM’s family, all incompatible) culminating in the PDP-11.

One of the computer scientists at Bell Labs who had worked on the MULTICS project.

Ken Thompson, subsequently found a small PDP-7 minicomputer that no one was using and set out to write a stripped-down, one-user version of MULTICS. This work later developed into the **UNIX** operating system, which became popular in the academic world, with government agencies, and with many companies.

The history of UNIX has been told elsewhere (e.g., Salus, 1994). Part of that story will be given in Chap. 10. For now, suffice it to say, that because the source code was widely available, various organizations developed their own (incompatible) versions, which led to chaos. Two major versions developed, **System V** , from AT&T, and **BSD,** (Berkeley Software Distribution) from the University of California at Berkeley. These had minor variants as well. To make it possible to write programs that could run on any UNIX system. IEEE developed a standard for UNIX, called **POSIX,** that most versions of UNIX now support. POSIX defines a minimal system call interface that conformant UNIX systems must support. In fact, some other operating systems now also support the POSIX interface.

As an aside, it is worth mentioning that in 1987, the author released a small clone of UNIX, called **MINIX,** for educational purposes. Functionally, MINIX is very similar to UNIX, including POSIX support. A book describing its internal operation and listing the source code in an appendix is also available (Tanenbaum and Woodhull, 1997). MINIX is available for free (including all the source code) over the Internet at URL *www.cs.vu.nl/-ast/minix.html.*

The desire for a free production (as opposed to educational) version of MINIX led a Finnish student, Linus Torvalds, to write **Linux.** This system was developed on MINIX and originally supported various MINIX features (e.g., the MINIX file system). It has since been extended in many ways but still retains a large amount of underlying structure common to MINIX, and to UNIX (upon which the former was based). Most of what will be said about UNIX in this book thus applies to System V, BSD, MINIX, Linux, and other versions and clones of UNIX as well.

**1.2.4 The Fourth Generation (1980-Present) Personal Computers**

With the development of LSI (Large Scale Integration) circuits, chips containing thousands of transistors on a square centimeter of silicon, the age of the personal computer dawned. In terms of architecture, personal computers (initially called **microcomputers)** were not all that different from minicomputers of the PDP-11 class,

but in terms of price they certainly were different. Where the minicomputer made it possible for a department in a company or university to have its own computer, the microprocessor chip made it possible for a single individual to have his or her own personal computer.

In 1974, when Intel came out with the 8080, the first general-purpose 8-bit CPU, it wanted an operating system for the 8080, in part to be able to test it. Intel asked one of its consultants, Gary Kildall, to write one. Kildall and a friend first built a controller for the newly-released Shugart Associates 8-inch floppy disk and hooked the floppy disk up to the 8080, thus producing the first microcomputer with a disk. Kildall then wrote a disk-based operating system called **CP/M (Control Program for Microcomputers)** for it. Since Intel did not think that disk-based microcomputers had much of a future, when Kildall asked for the rights to CP/M, Intel granted his request. Kildall then formed a company, Digital Research, to further develop and sell CP/M.

In 1977, Digital Research rewrote CP/M to make it suitable for running on the many microcomputers using the 8080, Zilog Z80, and other CPU chips. Many application programs were written to run on CP/M, allowing it to completely dominate the world of microcomputing for about 5 years.

In the early 1980s, IBM designed the IBM PC and looked around for software to run on it. People from IBM contacted Bill Gates to license his BASIC interpreter. They also asked him if he knew of an operating system to run on the PC, Gates suggested that IBM contact Digital Research, then the world’s dominant operating systems company. Making what was surely the worst business decision in recorded history, Kildall refused to meet with IBM, sending a subordinate instead. To make matters worse, his lawyer even refused to sign IBM’s nondisclosure agreement covering the not-yet-announced PC. Consequently, IBM went back to Gates asking if he could provide them with an operating system.

When IBM came back, Gates realized that a local computer manufacturer, Seattle Computer Products, had a suitable operating system. **DOS (Disk Operating System).** He approached them and asked to buy it (allegedly for $50,000). which they readily accepted. Gates then offered IBM a DOS/BASIC package which IBM accepted. IBM wanted certain modifications, so Gates hired the person who wrote DOS, Tim Paterson, as an employee of Gates’ fledgling company, Microsoft, to make them. The revised system was renamed **MS-DOS (MicroSoft Disk Operating System)** and quickly came to dominate the IBM PC market. A key factor here was Gates’ (in retrospect, extremely

wise) decision to sell MS-DOS to computer companies for bundling with their hardware, compared to Kildall’s attempt to sell CP/M to end users one at a time (at least initially).

By the time the IBM PC/AT came out in 1983 with the Intel 80286 CPU, MS-DOS was firmly entrenched and CP/M was on its last legs. MS-DOS was later widely used on the 80386 and 80486. Although the initial version of MS-DOS was fairly primitive, subsequent versions included more advanced features, including many taken from UNIX. (Microsoft was well aware of UNIX, even selling a microcomputer version of it called XENIX during the company’s early years.)

CP/M, MS-DOS, and other operating systems for early microcomputers were all based on users typing in commands from the keyboard. That eventually changed due to research done by Doug Engelbart at Stanford Research Institute in the 1960s. Engelbart invented the **GUI (Graphical User Interface),** pronounced “gooey,” complete with windows, icons, menus, and mouse. These ideas were adopted by researchers at Xerox PARC and incorporated into machines they built.

One day, Steve Jobs, who co-invented the Apple computer in his garage, visited PARC, saw a GUI, and instantly realized its potential value, something Xerox management famously did not (Smith and Alexander, 1988). Jobs then embarked on building an Apple with a GUI. This project led to the Lisa, which was too expensive and failed commercially. Jobs’ second attempt, the Apple Macintosh, was a huge success, not only because it was much cheaper than the Lisa, but also because it was **user friendly,** meaning that it was intended for users who not only knew nothing about computers but furthermore had absolutely no intention whatsoever of learning.

When Microsoft decided to build a successor to MS-DOS, it was strongly influenced by the success of the Macintosh. It produced a GUI-based system called Windows, which originally ran on top of MS-DOS (i.e.. it was more like a shell than a true operating system). For about 10 years, from 1985 to 1993, Windows was just a graphical environment on top of MS-DOS. However, starting in 1995 a freestanding version of Windows, Windows 95, was released that incorporated many operating system features into it, using the underlying MS-DOS system only for booting and running old MS-DOS programs, in 1998, a slightly modified version of this system, called Windows 98 was released. Nevertheless, both Windows 95 and Windows 98 still contain a large amount of 16-bit Intel assembly language.

Another Microsoft operating system is Windows NT (NT stands for New Technology), which is compatible with Windows 95 at a certain level, but a complete rewrite from scratch internally. It is a full 32-bit system. The lead designer for Windows NT was David Cutler, who was also one of the designers of the VAX VMS operating system, so some ideas from VMS are present in NT. Microsoft expected that the first version of NT would kill off MS-DOS and all other versions of Windows since it was a vastly superior system, but it fizzled. Only with Windows NT 4.0 did it finally catch on in a big way, especially on corporate networks. Version 5 of Windows NT was renamed Windows 2000 in early 1999. It was intended to be the successor to both Windows 98 and Windows NT 4.0. That did not quite work out either, so Microsoft came out with yet another version of Windows 98 called **Windows Me (Millennium edition).**

The other major contender in the personal computer world is UNIX (and its various derivatives). UNIX is strongest on workstations and other high-end computers, such as network servers. It is especially popular on machines powered by high-performance RISC chips. On Pentium-based computers, Linux is becoming a popular alternative to Windows for students and increasingly many corporate users. (As an aside, throughout this book we will use the term “Pentium” to mean the Pentium I, II, III, and 4.)

Although many UNIX users, especially experienced programmers, prefer a command based interface to a GUI, nearly all UNIX systems support a windowing system called the **X Windows** system produced at M.I.T. This system handles the basic window management, allowing users to create, delete, move, and resize windows using a mouse. Often a complete GUI, such as **Motif,** is available to run on top of the X Windows system giving UNIX a look and feel something like the Macintosh or Microsoft Windows, for those UNIX users who want such a thing.

An interesting development that began taking place during the mid-1980s is the growth of networks of personal computers running **network operating systems** and **distributed operating** systems (Tanenbaum and Van Steen, 2002). In a network operating system, the users are aware of the existence of multiple computers and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own local user (or users).

Network operating systems are not fundamentally different from single-processor operating systems. They obviously need a network interface controller and some low level software to drive it, as well as programs to achieve remote login and remote file access, but these additions do not change the essential structure of the operating system.

A distributed operating system, in contrast, is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple processors. The users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

True distributed operating systems require more than just adding a little code to a uniprocessor operating system, because distributed and centralized systems differ in critical ways. Distributed systems, for example, often allow applications to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimize the amount of parallelism.

Communication delays within the network often mean that these (and other) algorithms must run with incomplete, outdated, or even incorrect information. This situation is radically different from a single processor system in which the operating system has complete information about the system state.

**1.2.5 Ontogeny Recapitulates Phytogeny**

After Charles Darwin’s book *The Origin of the Species* was published, the German zoologist Ernst Haeckel stated that “Ontogeny Recapitulates Phylogeny.” By this he meant that the development of an embryo (ontogeny) repeats (i.e., recapitu1ates) the evolution of the species (phy1ogeny). In other words, after fertilization, a human egg goes through stages of being a fish, a pig, and so on before turning into a human baby. Modern biologists regard this as a gross simplification, but it still has a kernel of truth in it.

Something analogous has happened in the computer industry. Each new species (mainframe, minicomputer, personal computer, embedded computer, smart card, etc.) seems to go through the development that its ancestors did. The first mainframes were programmed entirely in assembly language. Even complex programs, like compilers and operating systems, were written in assembler. By the time minicomputers appeared on the scene, FORTRAN, COBOL, and other high-level languages were common on mainframes, but the new minicomputers were nevertheless programmed in assembler (for lack of memory). When microcomputers (early personal computers) were invented, they, too, were programmed in assembler, even though by then minicomputers were also programmed in high-level languages. Palmtop computers also started with assembly

code but quickly moved on to high-level languages (mostly because the development work was done on bigger machines). The same is true for smart cards.

Now let us look at operating systems. The first mainframes initially had no protection hardware and no support for multiprogramming, so they ran simple operating systems that handled one manually-loaded program at a time. Later they acquired the hardware and operating system support to handle multiple programs at once, and then full timesharing capabilities.

When minicomputers first appeared, they also had no protection hardware and ran one manually-loaded program at a time, even though multiprogramming was well established in the mainframe world by then. Gradually, they acquired protection hardware and the ability to run two or more programs at once. The first microcomputers were also capable of running only one program at a time, but later acquired the ability to multiprogram. Palmtops and smart cards went the same route.

Disks first appeared on large mainframes, then on minicomputers, microcomputers, and so on down the line. Even now, smart cards do not have hard disks, but with the advent of flash ROM, they will soon have the equivalent of it. When disks first appeared, primitive file systems sprung up. On the CDC 6600, easily the most powerful mainframe in the world during much of the 1960s, the file system consisted of users having the ability to create a file and then declare it to be permanent, meaning it stayed on the disk even after the creating program exited. To access such a file later, a program had to attach it with a special command and give its password (supplied when the file was made permanent). In effect, there was a single directory shared by all users. It was up to the users to avoid file name conflicts. Early minicomputer file systems had a single directory shared by all users and so did early microcomputer file systems.

Virtual memory (the ability to run programs larger than the physical memory) had a similar development. It first appeared in mainframes, minicomputers, microcomputers and gradually worked its way down to smaller and smaller systems. Networking had a similar history.

In all cases, the software development was dictated by the technology. The first microcomputers, for example, had something like 4 KB of memory and no protection hardware. High-level languages and multiprogramming were simply too much for such a tiny system to handle. As the microcomputers evolved into modern personal computers, they acquired the necessary hardware and then the necessary software to

handle more advanced features. It is likely that this development will continue for years to come. Other fields may also have this wheel of reincarnation, but in the computer industry it seems to spin faster.

**1.3 THE OPERATING SYSTEM ZOO**

All of this history and development has left us with a wide variety of operating systems, not all of which are widely known. In this section we will briefly touch upon seven of them. We will come back to some of these different kinds of systems later in the book.

**1.3.1 Mainframe Operating Systems**

At the high end are the operating systems for the mainframes, those room-sized computers still found in major corporate data centers. These computers distinguish themselves from personal computers in terms of their I/O capacity. A mainframe with 1000 disks and thousands of gigabytes of data is not unusual: a personal computer with these specifications would be odd indeed. Mainframes are also making something of a comeback as high-end Web servers, servers for large-scale electronic commerce sites, and servers for business-to-business transactions.

The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O. They typically offer three kinds of services: batch, transaction processing, and timesharing. A batch system is one that processes routine jobs without any interactive user present. Claims processing in an insurance company or sales reporting for a chain of stores is typically done in batch mode. Transaction processing systems handle large numbers of small requests, for example, check processing at a bank or airline reservations. Each unit of work is small, but the system must handle hundreds or thousands per second. Timesharing systems allow multiple remote users to run jobs on the computer at once, such as querying a big database. These functions are closely related: mainframe operating systems often perform all of them. An example mainframe operating system is OS/390, a descendant of OS/360.

**1.3.2 Server Operating Systems**

One level down are the server operating systems. They run on servers, which are either

very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service, or Web service. Internet providers run many server machines to support their customers and Web sites use servers to store the Web pages and handle the incoming requests. Typical server operating systems are UNIX and Windows 2000. Linux is also gaining ground for servers.

**1.3.3 Multiprocessor Operating Systems**

An increasingly common way to get major-league computing power is to connect multiple CPUs into a single system. Depending on precisely how they are connected and what is shared, these systems are called parallel computers, multicomputers, or multiprocessors. They need special operating systems, but often these are variations on the server operating systems, with special features for communication and connectivity.

**1.3.4 Personal Computer Operating Systems**

The next category is the personal computer operating system. Their job is to provide a good interface to a single user. They are widely used for word processing, spreadsheets, and Internet access. Common examples are Windows 98, Windows 2000, the Macintosh operating system, and Linux. Personal computer operating systems are so widely known that probably little introduction is needed. In fact, many people are not even aware that other kinds exist.

**1.3.5 Real-Time Operating Systems**

Another type of operating system is the real-time system. These systems are characterized by having time as a key parameter. For example, in industrial process control systems, real-time computers have to collect data about the production process and use it to control machines in the factory. Often there are hard deadlines that must be met. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time, if a welding robot welds too early or too late, the car will be ruined. If the action absolutely *must* occur at a certain moment (or within a certain range), we have a **hard real-time system.**

Another kind of real-time system is a **soft real-time** system, in which missing an occasional deadline is acceptable. Digital audio or multimedia systems fall in this category. VxWorks and QNX are well-known real-time operating systems.

**1.3.6 Embedded Operating Systems**

Continuing on down to smaller and smaller systems, we come to palmtop computers and embedded systems. A palmtop computer or **PDA (Personal Digital Assistant)** is a small computer that fits in a shirt pocket and performs a small number of functions such as an electronic address book and memo pad. Embedded systems run on the computers that control devices that are not generally thought of as computers, such as TV sets, microwave ovens, and mobile telephones. These often have some characteristics of real time systems but also have size, memory, and power restrictions that make them special. Examples of such operating systems are PalmOS and Windows CE (Consumer Electronics).

**1.3.7 Smart Card Operating Systems**

The smallest operating systems run on smart cards, which are credit card-sized devices containing a CPU chip. They have very severe processing power and memory constraints. Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions on the same smart card. Often these are proprietary systems.

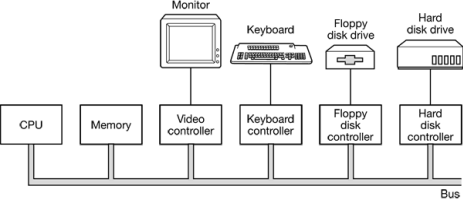
Some smart cards are Java oriented. What this means is that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets (small programs) are downloaded to the card and are interpreted by the JVM interpreter. Some of these cards can handle multiple Java applets at the same time, leading to multiprogramming and the need to schedule them. Resource management and protection also become an issue when two or more applets are present at the same time. These issues must be handled by the (usually extremely primitive) operating system present on the card.

**1.4 COMPUTER HARDWARE REVIEW**

An operating system is intimately tied to the hardware of the computer it runs on. It

extends the computer’s instruction set and manages its resources. To work it must know a great deal about the hardware, at least, about how the hardware appears to the programmer.

Conceptually, a simple personal computer can be abstracted to a model resembling that of Fig. 1-5. The CPU, memory, and I/O devices are all connected by a system bus and communicate with one another over it. Modern personal computers have a more complicated structure, involving multiple buses, which we will look at later. For the time being, this model will be sufficient. In the following sections, we will briefly review these components and examine some of the hardware issues that are of concern to operating system designers.

**Figure 1-5.** Some of the components of a simple personal computer.

**1.4.1 Processors**

The “brain” of the computer is the CPU. It fetches instructions from memory and executes them. The basic cycle of every CPU is to fetch the first instruction from memory, decode it to determine its type and operands, execute it, and then fetch, decode, and execute subsequent instructions. In this way, programs are carried out.

Each CPU has a specific set of instructions that it can execute. Thus a Pentium cannot execute SPARC programs and a SPARC cannot execute Pentium programs. Because accessing memory to get an instruction or data word takes much longer than executing an instruction, all CPUs contain some registers inside to hold key variables and temporary results. Thus the instruction set generally contains instructions to load a word

from memory into a register, and store a word from a register into memory. Other instructions combine two operands from registers, memory, or both into a result, such as adding two words and storing the result in a register or in memory.

In addition to the general registers used to hold variables and temporary results, most computers have several special registers that are visible to the programmer. One of these is the **program counter,** which contains the memory address of the next instruction to be fetched. After that instruction has been fetched, the program counter is updated to point to its successor.

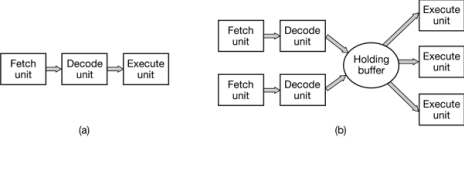
Another register is the **stack pointer,** which points to the top of the current stack in memory. The stack contains one frame for each procedure that has been entered but not yet exited. A procedure’s stack frame holds those input parameters, local variables, and temporary variables that are not kept in registers.

Yet another register is the **PSW (Program Status Word).** This register contains the condition code bits, which are set by comparison instructions, the CPU priority, the mode (user or kernel), and various other control bits. User programs may normally read the entire PSW but typically may write only some of its fields. The PSW plays an important role in system calls and I/O.

The operating system must be aware of all the registers. When time multiplexing the CPU, the operating system will often stop the running program to (re)start another one. Every time it stops a running program, the operating system must save all the registers so they can be restored when the program runs later.

To improve performance, CPU designers have long abandoned the simple model of fetching, decoding, and executing one instruction at a time. Many modern CPUs have facilities for executing more than one instruction at the same time. For example, a CPU might have separate fetch, decode, and execute units, so that while it was executing instruction *n* , it could also be decoding instruction *n* + 1 and fetching instruction *n* + 2.

Such an organization is called a **pipeline** and is illustrated in Fig. 1-6(a) for a pipeline with three stages. Longer pipelines are common. In most pipeline designs, once an instruction has been fetched into the pipeline, it must he executed, even if the preceding instruction was a conditional branch that was taken. Pipelines cause compiler writers and operating system writers great headaches because they expose the complexities of the underlying machine to them.

**Figure 1-6.** (a) A three-stage pipeline. (b) A superscalar CPU.

Even more advanced than a pipeline design is a **superscalar** CPU, shown in Fig. 1-6(b). In this design, multiple execution units are present, for example, one for integer arithmetic, one for floating-point arithmetic, and one for Boolean operations. Two or more instructions are fetched at once, decoded, and dumped into a holding buffer until they can be executed. As soon as an execution unit is free, it looks in the holding buffer to see if there is an instruction it can handle, and if so, it removes the instruction from the buffer and executes it. An implication of this design is that program instructions are often executed out of order. For the most part, it is up to the hardware to make sure the result produced is the same one a sequential implementation would have produced, but an annoying amount of the complexity is foisted onto the operating system, as we shall see.

Most CPUs, except very simple ones used in embedded systems, have two modes, kernel mode and user mode, as mentioned earlier. Usually a bit in the PSW controls the mode. When running in kernel mode, the CPU can execute every instruction in its instruction set and use every feature of the hardware. The operating system runs in kernel mode, giving it access to the complete hardware.

In contrast, user programs run in user mode, which permits only a subset of the instructions to be executed and a subset of the features to be accessed. Generally, all instructions involving I/O and memory protection are disallowed in user mode. Setting the PSW mode bit to kernel mode-is also forbidden, of course.

To obtain services from the operating system, a user program must make a **system call,** which traps into the kernel and invokes the operating system. The TRAP instruction

switches from user mode to kernel mode and starts the operating system. When the work has been completed, control is returned to the user program at the instruction following the system call. We will explain the details of the system call process later in this chapter. As a note on typography, we will use the lower case Helvetica font to indicate system calls in running text, like this: read .

It is worth noting that computers have traps other than the instruction for executing a system call. Most of the other traps are caused by the hardware to warn of an exceptional situation such as an attempt to divide by 0 or a floating-point underflow. In all cases the operating system gets control and must decide what to do. Sometimes the program must be terminated with an error. Other times the error can be ignored (an underflowed number can be set to 0). Finally, when the program has announced in advance that it wants to handle certain kinds of conditions, control can be passed back to the program to let it deal with the problem.

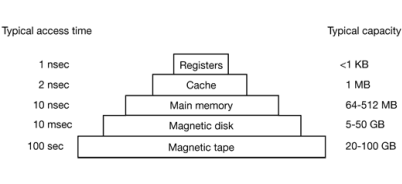
**1.4.2 Memory**

The second major component in any computer is the memory. Ideally, a memory should be extremely fast (faster than executing an instruction so the CPU is not held up by the memory), abundantly large, and dirt cheap. No current technology satisfies all of these goals, so a different approach is taken. The memory system is constructed as a hierarchy of layers, as shown in Fig. 1-7.

The top layer consists of the registers internal to the CPU. They are made of the same material as the CPU and are thus just as fast as the CPU. Consequently, there is no delay in accessing them. The storage capacity available in them is typically 32 x 32-bits on a 32-bit CPU and 64 x 64-bits on a 64-bit CPU. Less than 1 KB in both cases. Programs must manage the registers (i.e., decide what to keep in them) themselves, in software.

Next comes the cache memory, which is mostly controlled by the hardware. Main memory is divided up into cache lines, typically 64 bytes, with addresses 0 to 63 in cache fine 0, addresses 64 to 127 in cache line 1, and so on. The most heavily used cache lines are kept in a high-speed cache located inside or very close to the CPU. When the program needs to read a memory word, the cache hardware checks to see if the line needed is in the cache. If it is, called a **cache hit, t** he request is satisfied from the cache and no memory request is sent over the bus to the main memory. Cache hits normally take about two clock cycles. Cache misses have to go to memory, with a substantial time penalty. Cache memory is limited in size due to its high cost. Some

machines have two or even three levels of cache, each one slower and bigger than the one before it.

**Figure 1-7.** A typical memory hierarchy. The numbers are very rough approximations.

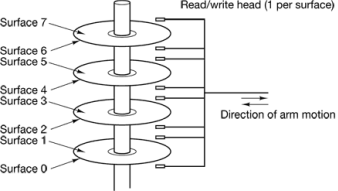
Main memory comes next. This is the workhorse of the memory system. Main memory is often called **RAM (Random Access Memory).** Old timers sometimes call it **core memory,** because computers in the 1950s and 1960s used tiny magnetizable ferrite cores for main memory. Currently, memories are tens to hundreds of megabytes and growing rapidly. All CPU requests that cannot be satisfied out of the cache go to main memory.

Next in the hierarchy is magnetic disk (hard disk). Disk storage is two orders of magnitude cheaper than RAM per bit and often two orders of magnitude larger as well. The only problem is that the time to randomly access data on it is close to three orders of magnitude slower. This low speed is due to the fact that a disk is a mechanical device, as shown in Fig. 1-8.

A disk consists of one or more metal platters that rotate at 5400, 7200, or 10,800 rpm. A mechanical arm pivots over the platters from the corner, similar to the pickup arm on an old 33 rpm phonograph for playing vinyl records. Information is written onto the disk in a series of concentric circles. At any given arm position, each of the heads can read an annular region called a **track.** Together, all the tracks for a given arm position form a **cylinder.**

Each track is divided into some number of sectors, typically 512 bytes per sector. On modern disks, the outer cylinders contain more sectors than the inner ones. Moving the

arm from one cylinder to the next one takes about 1 msec. Moving it to a random cylinder typically takes 5 msec to 10 msec, depending on the drive. Once the arm is on the correct track, the drive must wait for the needed sector to rotate under the head, an additional delay of 5 msec to 10 msec, depending on the drive’s rpm. Once the sector is under the head, reading or writing occurs at a rate of 5 MB/sec on low-end disks to 160 MB/sec on faster ones.



**Figure 1-8.** Structure of a disk drive.

The final layer in the memory hierarchy is magnetic tape. This medium is often used as a backup for disk storage and for holding very large data sets. To access a tape, it must first be put into a tape reader, either by a person or a robot (automated tape handling is common at installations with huge databases). Then the tape may have to be spooled forwarded to get to the requested block. All in all, this could take minutes. The big plus of tape is that it is exceedingly cheap per bit and removable, which is important for backup tapes that must be stored off-site in order to survive fires, floods, earthquakes, etc.

The memory hierarchy we have discussed is typical, but some installations do not have all the layers or have a few different ones (such as optical disk). Still, in all of them, as one goes down the hierarchy, the random access time increases dramatically, the capacity increases equally dramatically, and the cost per bit drops enormously. Consequently, it is likely that memory hierarchies will be around for years to come.

In addition to the kinds of memory discussed above, many computers have a small amount of nonvolatile random access memory. Unlike RAM, nonvolatile memory does not lose its contents when the power is switched off. **ROM (Read Only Memory)** is

programmed at the factory and cannot be changed afterward. It is fast and inexpensive. On some computers, the bootstrap loader used to start the computer is contained in ROM. Also, some I/O cards come with ROM for handling low-level device control.

**EEPROM (Electrically Erasable ROM)** and **flash RAM** are also nonvolatile, but in contrast to ROM can be erased and rewritten. However, writing them takes orders of magnitude more time than writing RAM, so they are used in the same way ROM is, only with the additional feature that it is now possible to correct bugs in programs they hold by rewriting them in the field.

Yet another kind of memory is CMOS, which is volatile. Many computers use CMOS memory to hold the current time and date. The CMOS memory and the clock circuit that increments the time in it are powered by a small battery, so the time is correctly updated, even when the computer is unplugged. The CMOS memory can also hold the configuration parameters, such as which disk to boot from. CMOS is used because it draws so little power that the original factory-installed battery often lasts for several years. However, when it begins to fail, the computer can appear to have Alzheimer’s disease, forgetting things that it has known for years, like which hard disk to boot from.

Let us now focus on main memory for a little while. It is often desirable to hold multiple programs in memory at once. If one program is blocked waiting for a disk read to complete, another program can use the CPU, giving a better CPU utilization. However, with two or more programs in main memory at once, two problems must be solved:

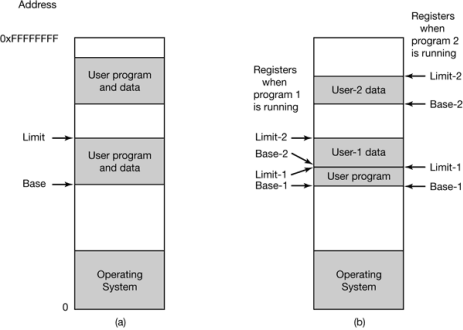
1. How to protect the programs from one another and the kernel from them all. 2. How to handle relocation.

Many solutions are possible. However, all of them involve equipping the CPU with special hardware.

The first problem is obvious, but the second one is a bit more subtle. When a program is compiled and linked, the compiler and liker do not know where in physical memory it will be loaded when it is executed. For this reason, they usually assume it will start at address 0, and just put the first instruction there. Suppose that the first instruction fetches a word from memory address 10,000. Now suppose that the entire program and data are loaded starting at address 50,000. When the first instruction executes, it will fail because it will reference the word at 10,000, instead of the word at 60,000. To solve this

problem, we need to either relocate the program at load time, finding all the addresses and modifying them (which is doable, but expensive), or have relocation done on-the-fly during execution.

The simplest solution is shown in Fig. 1-9(a). In this figure we see a computer equipped with two special registers, the base register and the limit register. (Please note that in this book, numbers beginning with 0x are in hexadecimal the C language convention. Similarly, number beginning with a leading zero are in octal.) When a program is run, the base register is set to point to the start of its program text and the limit register tells how large the combined program text and data are. When an instruction is to be fetched, the hardware checks to see if the program counter is less than the limit register, and if it is, adds it to the base register and sends the sum to memory. Similarly when the program wants to fetch a data word (e.g., from address 10,000), the hardware automatically adds the contents of the base register (e.g., 50,000) to that address and sends the sum (60,000) to the memory. The base register makes it impossible for a program to reference any part of memory below itself. Furthermore, the limit register makes it impossible to reference any part of memory above itself. Thus this scheme solves both the protection and the relocation problem at the cost of two new registers and a slight increase in cycle time (to perform the limit check and addition).

**Figure 1-9.** (a) Use of one base-limit pair. The program can access memory between the base and the limit. (b) Use of two base-limit pairs. The program code is between Base-1 and Limit-1 whereas the data are between Base-2 and Limit-2.

The check and mapping result in converting an address generated by the program, called a **virtual address,** into an address used by the memory, called a **physical address.** The device that performs the check and mapping is called the **MMU (Memory Management Unit).** It is located on the CPU chip or close to it, but is logically between the CPU and the memory.

A more sophisticated MMU is illustrated in Fig. 1-9(b). Here we have an MMU with two pairs of base and limit registers, one for the program text and one for the data. The program counter and all other references to the program text use pair 1 and data references use pair 2. As a consequence, it is now possible to have multiple users share the same program with only one copy of it in memory, something not possible with the first scheme. When program 1 is running, the four registers are set as indicated by the arrows to the left of Fig. 1-9(b). When program 2 is running, they are set as indicated by the arrows to the right of the figure. Much more sophisticated MMUs exist. We will

study some of them later in this book. The thing to note here is that managing the MMU must be an operating system function, since users cannot be trusted to do it correctly.

Two aspects of the memory system have a major effect on performance. First, caches hide the relatively slow speed of memory. When a program has been running for a while, the cache is full of that programs cache lines, giving good performance. However, when the operating system switches from one program to another, the cache remains full of the first program’s cache lines. The ones needed by the new program must be loaded one at a time from physical memory. This operation can be a major performance hit if it happens too often.

Second, when switching from one program to another, the MMU registers have to be changed. In Fig. 1-9(b), only four registers have to be reset, which is not a problem, but in real MMUs, many more registers have to be reloaded, either explicitly or dynamically, as needed. Either way, it takes time. The moral of the story is that switching from one program to another, called a **context switch,** is an expensive business.

**1.4.3 I/O Devices**

Memory is not the only resource that the operating system must manage. I/O devices also interact heavily with the operating system. As we saw in Fig. 1-5, I/O devices generally consist of two parts: a controller and the device itself. The controller is a chip or a set of chips on a plug-in board that physically controls the device. It accepts commands from the operating system, for example, to read data from the device, and carries them out.

In many cases, the actual control of the device is very complicated and detailed, so it is the job of the controller to present a simpler interface to the operating system. For example, a disk controller might accept a command to read sector 11,206 from disk 2. The controller then has to convert this linear sector number to a cylinder, sector, and head. This conversion may be complicated by the fact that outer cylinders have more sectors than inner ones and that some bad sectors have been remapped onto other ones. Then the controller has to determine which cylinder the disk arm is on and give it a sequence of pulses to move in or out the requisite number of cylinders. It has to wait until the proper sector has rotated under the head and then start reading and storing the bits as they come off the drive, removing the preamble and computing the checksum. Finally, it has to assemble the incoming bits into words and store them in memory. To

do all this work, controllers often contain small embedded computers that are programmed to do their work.

The other piece is the actual device itself. Devices have fairly simple interfaces, both because they cannot do much and to make them standard. The latter is needed so that any IDE disk controller can handle any IDE disk, for example. **IDE** stands for **Integrated Drive Electronics** and is the standard type of disk on Pentiums and some other computers. Since the actual device interface is hidden behind the controller, all that the operating system sees is the interface to the controller which may be quite different from the interface to the device.

Because each type of controller is different, different software is needed to control each one. The software that talks to a controller, giving it commands and accepting responses, is called a device driver. Each controller manufacturer has to supply a driver for each operating system it supports. Thus a scanner may come with drivers for Windows 98, Windows 2000, and UNIX, for example.

To be used, the driver has to be put into the operating system so it can run in kernel mode. Theoretically, drivers can run outside the kernel, but few current systems support this possibility because it requires the ability to allow a user-space driver to be able to access the device in a controlled way, a feature rarely supported. There are three ways the driver can be put into the kernel. The first way is to relink the kernel with the new driver and then reboot the system. Many UNIX systems work like this. The second way is to make an entry in an operating system file telling it that it needs the driver and then reboot the system. At boot time, the operating system goes and finds the drivers it needs and loads them. Windows works this way. The third way is for the operating system to be able to accept new drivers while running and install them on-the-fly without the need to reboot. This way used to be rare but is becoming much more common now. Hot pluggable devices, such as USB and IEEE 1394 devices (discussed below) always need dynamically loaded drivers.

Every controller has a small number of registers that are used to communicate with it. For example, a minimal disk controller might have registers for specifying the disk address, memory address, sector count, and direction (read or write). To activate the controller, the driver gets a command from the operating system, then translates it into the appropriate values to write into the device registers.

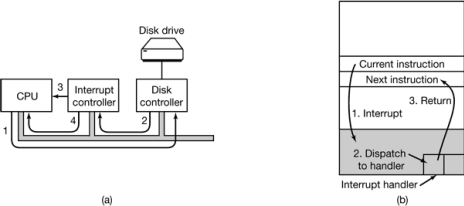
On some computers, the device registers are mapped into the operating system’s address

space, so they can be read and written like ordinary memory words. On such computers, no special I/O instructions are needed and user programs can be kept away from the hardware by not putting these memory addresses within their reach (e.g., by using base and limit registers). On other computers, the device registers are put in a special I/O port space, with each register having a port address. On these machines, special IN and OUT instructions are available in kernel mode to allow drivers to read and write the registers. The former scheme eliminates the need for special I/O instructions but uses up some of the address space. The latter uses no address space but requires special instructions. Both systems are widely used.

Input and output can be done in three different ways. In the simplest method, a user program issues a system call, which the kernel then translates into a procedure call to the appropriate driver. The driver then starts the I/O and sits in a tight loop continuously polling the device to see if it is done (usually there is some bit that indicates that the device is still busy). When the I/O has completed, the driver puts the data where they are needed (if any), and returns. The operating system then returns control to the caller. This method is called **busy waiting** and has the disadvantage of tying up the CPU polling the device until it is finished.

The second method is for the driver to start the device and ask it to give an interrupt when it is finished. At that point the driver returns. The operating system then blocks the caller if need be and looks for other work to do. When the controller detects the end of the transfer, it generates an **interrupt** to signal completion.

Interrupts are very important in operating systems, so let us examine the idea more closely. In Fig. 1-10(a) we see a three-step process for I/O. In step 1, the driver tells the controller what to do by writing into its device registers. The controller then starts the device. When the controller has finished reading or writing the number of bytes it has been told to transfer, it signals the interrupt controller chip using certain bus lines in step 2. If the interrupt controller is prepared to accept the interrupt (which it may not be if it is busy with a higher priority one), it asserts a pin on the CPU chip informing it, in step 3. In step 4, the interrupt controller puts the number of the device on the bus so the CPU can read it and know which device has just finished (many devices may be running at the same time).

**Figure 1-10.** (a) The steps in starting an I/O device and getting an interrupt. (b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program.

Once the CPU has decided to take the interrupt, the program counter and PSW are typically then pushed onto the current stack and the CPU switched into kernel mode. The device number may be used as an index into part of memory to find the address of the interrupt handler for this device. This part of memory is called the **interrupt vector.** Once the interrupt handler (part of the driver for the interrupting device) has started, it removes the stacked program counter and PSW and saves them, then queries the device to learn its status. When the handler is all finished, it returns to the previously-running user program to the first instruction that was not yet executed. These steps are shown in Fig. 1-10(b).

The third method for doing I/O makes use of a special **DMA (Direct Memory Access)** chip that can control the flow of bits between memory and some controller without constant CPU intervention. The CPU sets up the DMA chip telling it how many bytes to transfer, the device and memory addresses involved, and the direction, and lets it go. When the DMA chip is done, it causes an interrupt, which is handled as described above. DMA and I/O hardware in general will be discussed in more detail in Chap. 5.

Interrupts can often happen at highly inconvenient moments, for example, while another interrupt handler is running. For this reason, the CPU has a way to disable interrupts and then reenable them later. While interrupts are disabled, any devices that finish continue to assert their interrupt signals, but the CPU is not interrupted until interrupts are enabled again. If multiple devices finish while interrupts are disabled, the interrupt

controller decides which one to let through first, usually based on static priorities assigned to each device. The highest priority device wins.

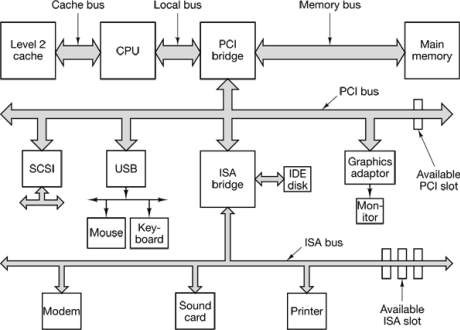
**1.4.4 Buses**

The organization of Fig. 1-5 was used on minicomputers for years and also on the original IBM PC. However, as processors and memories got faster, the ability of a single bus (and certainly the IBM PC bus) to handle all the traffic was strained to the breaking point. Something had to give. As a result, additional buses were added, both for faster I/O devices and for CPU to memory traffic. As a consequence of this evolution, a large Pentium system currently looks something like Fig. 1-11.

This system has eight buses (cache, local, memory, PCI, SCSI, USB, IDE, and ISA), each with a different transfer rate and function. The operating system must be aware of all of them for configuration and management. The two main buses are the original IBM PC **ISA (Industry Standard Architecture)** bus and its successor, the **PCI (Peripheral Component Interconnect)** bus. The ISA bus, which was originally the IBM PC/AT bus, runs at 8.33 MHz and can transfer 2 bytes at once, for a maximum speed of 16.67 MB/sec. it is included for backward compatibility with old and slow I/O cards. The PCI bus was invented by Intel as a successor to the ISA bus. It can run at 66 MHz and transfer 8 bytes at a time, for a data rate of 528 MB/sec. Most high-speed I/O devices use the PCI bus now. Even some non-Intel computers use the PCI bus due to the large number of I/O cards available for it.

In this configuration, the CPU talks to the PCI bridge chip over the local bus, and the PCI bridge chip talks to the memory over a dedicated memory bus, often running at 100 MHz. Pentium systems have a level-1 cache on chip and a much larger level-2 cache off chip, connected to the CPU by the cache bus.

In addition, this system contains three specialized buses: IDE, USB, and SCSI. The IDE bus is for attaching peripheral devices such as disks and CD-ROMs to the system. The IDE bus is an outgrowth of the disk controller interface on the PC/AT and is now standard on nearly all Pentium-based systems for the hard disk and often the CD-ROM.

**Figure 1-11.** The structure of a large Pentium system

The **USB (Universal Serial Bus)** was invented to attach all the slow I/O devices, such as the keyboard and mouse, to the computer. It uses a small four-wire connector, two of which supply electrical power to the USB devices. USB is a centralized bus in which a root device polls the I/O devices every 1 msec to see if they have any traffic. It can handle an aggregate load of 1.5 MB/sec. All the USB devices share a single USB device driver, making it unnecessary to install a new driver for each new USB device. Consequently, USB devices can be added to the computer without the need to reboot.

The **SCSI (Small Computer System Interface)** bus is a high-performance bus intended for fast disks, scanners, and other devices needing considerable bandwidth. It can run at up to 160 MB/sec. It has been present on Macintosh systems since they were invented and is also popular on UNIX and some Intel-based systems.

Yet another bus (not shown in Fig. 1-11) is **IEEE 1394.** Sometimes it is called FireWire, although strictly speaking, FireWire is the name Apple uses for its implementation of 1394. Like USB, IEEE 1394 is bit serial but is designed for packet

transfers at speeds up to 50 MB/sec, making it useful for connecting digital camcorders and similar multimedia devices to a computer. Unlike USB, IEEE 1394 does not have a central controller. SCSI and IEEE 1394 face competition from a faster version of USB being developed.

To work in an environment such as that of Fig. 1-11, the operating system has to know what is out there and configure it. This requirement led Intel and Microsoft to design a PC system called **plug and play,** based on a similar concept first implemented in the Apple Macintosh. Before plug and play, each I/O card had a fixed interrupt request level and fixed addresses for its I/O registers. For example, the keyboard was interrupt 1 and used I/O addresses 0x60 to 0x64, the floppy disk controller was interrupt 6 and used I/O addresses 0x3F0 to 0x3F7, and the printer was interrupt 7 and used I/O addresses 0x378 to 0x37A, and so on.

So far, so good. The trouble came when the user bought a sound card and a modem card and both happened to use, say, interrupt 4. They would conflict and would not work together. The solution was to include DIP switches or jumpers on every I/O card and instruct the user to please set them to select an interrupt level and I/O device addresses that did not conflict with any others in the user’s system. Teenagers who devoted their lives to the intricacies of the PC hardware could sometimes do this without making errors. Unfortunately, nobody else could, leading to chaos.

What plug and play does is have the system automatically collect information about the I/O devices, centrally assign interrupt levels and I/O addresses, and then tell each card what its numbers are. Very briefly, that works as follows on the Pentium. Every Pentium contains a parentboard (formerly called a motherboard before political correctness hit the computer industry). On the parentboard is a program called the system **BIOS (Basic Input Output System).** The BIOS contains low-level I/O software, including procedures to read the keyboard, write to the screen, and do disk I/O, among other things. Nowadays, it is held in a flash RAM, which is nonvolatile but which can be updated by the operating system when bugs are found in the BIOS.

When the computer is booted, the BIOS is started. It first checks to see how much RAM is installed and whether the keyboard and other basic devices are installed and responding correctly. It starts out by scanning the ISA and PCI buses to detect all the devices attached to them. Some of these devices are typically **legacy** (i.e., designed before plug and play was invented) and have fixed interrupt levels and I/O addresses (possibly set by switches or jumpers on the I/O card, but not modifiable by the operating

system). These devices are recorded. The plug and play devices are also recorded. If the devices present are different from when the system was last booted, the new devices are configured.

The BIOS then determines the boot device by trying a list of devices stored in the CMOS memory. The user can change this list by entering a BIOS configuration program just after booting. Typically, an attempt is made to boot from the floppy disk. If that fails the CD-ROM is tried. If neither a floppy nor a CD-ROM is present the system is booted from the hard disk. The first sector from the boot device is read into memory and executed. This sector contains a program that normally examines the partition table at the end of the boot sector to determine which partition is active. Then a secondary boot loader is read in from that partition. This loader reads in the operating system from the active partition and starts it.

The operating system then queries the BIOS to get the configuration information. For each device, it checks to see if it has the device driver. If not, it asks the user to insert a floppy disk or CD-ROM containing the driver (supplied by the device’s manufacturer). Once it has all the device drivers, the operating system loads them into the kernel. Then it initializes its tables, creates whatever background processes are needed, and starts up a login program or GUI on each terminal. At least, this is the way it is supposed to work. In real life, plug and play is frequently so unreliable that many people call it plug and pray.

**1.5 OPERATING SYSTEM CONCEPTS**

All operating systems have certain basic concepts such as processes, memory, and files that are central to understanding them. In the following sections, we will look at some of these basic concepts ever so briefly, as an introduction. We will come back to each of them in great detail later in this hook. To illustrate these concepts we will use examples from time to time, generally drawn from UNIX. Similar examples typically exist in other systems as well, however.

**1.5.1 Processes**

A key concept in all operating systems is the **process.** A process is basically a program in execution. Associated with each process is its **address space,** a list of memory

locations from some minimum (usually 0) to some maximum, which the process can read and write. The address space contains the executable program, the program’s data, and its stack. Also associated with each process is some set of registers, including the program counter, stack pointer, and other hardware registers, and all the other information needed to run the program.

We will come back to the process concept in much more detail in Chap. 2. but for the time being, the easiest way to get a good intuitive feel for a process is to think about timesharing systems. Periodically, the operating system decides to stop running one process and start running another, for example, because the first one has had more than its share of CPU time in the past second.

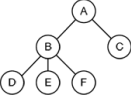
When a process is suspended temporarily like this, it must later be restarted in exactly the same state it had when it was stopped. This means that all information about the process must be explicitly saved somewhere during the suspension. For example, the process may have several files open for reading at once. Associated with each of these files is a pointer giving the current position (i.e., the number of the byte or record to be read next). When a process is temporarily suspended, all these pointers must be saved so that a read call executed after the process is restarted will read the proper data. In many operating systems, all the information about each process, other than the contents of its own address space, is stored in an operating system table called the **process table,** which is an array (or linked list) of structures, one for each process currently in existence.

Thus, a (suspended) process consists of its address space, usually called the **core image** (in honor of the magnetic core memories used in days of yore), and its process table entry, which contains its registers, among other things.

The key process management system calls are those dealing with the creation and termination of processes. Consider a typical example. A process called the **command interpreter** or **shell** reads commands from a terminal. The user has just typed a command requesting that a program be compiled. The shell must now create a new process that will run the compiler. When that process has finished the compilation, it executes a system call to terminate itself.

If a process can create one or more other processes (referred to as **child processes** ) and these processes in turn can create child processes, we quickly arrive at the process tree structure of Fig. 1-12. Related processes that are cooperating to get some job done often

need to communicate with one another and synchronize their activities. This communication is called **interprocess communication,** and will be addressed in detail in Chap. 2.



**Figure 1-12.** A process tree. Process *A* created two child processes: *B* and *C.* Process *B* created three child processes: *D* , *E* . and *F* .

Other process system calls are available to request more memory (or release unused memory), wait for a child process to terminate, and overlay its program with a different one.

Occasionally, there is a need to convey information to a running process that is not sitting around waiting for this information. For example, a process that is communicating with another process on a different computer does so by sending messages to the remote process over a computer network. To guard against the possibility that a message or its reply is lost, the sender may request that its own operating system notify it after a specified number of seconds, so that it can retransmit the message if no acknowledgement has been received yet. After setting this timer, the program may continue doing other work.

When the specified number of seconds has elapsed, the operating system sends an **alarm signal** to the process. The signal causes the process to temporarily suspend whatever it was doing, save its registers on the stack, and start running a special signal handling procedure, for example, to retransmit a presumably lost message. When the signal handler is done, the running process is restarted in the state it was in just before the signal. Signals are the software analog of hardware interrupts and can be generated by a variety of causes in addition to timers expiring. Many traps detected by hardware, such as executing an illegal instruction or using an invalid address, are also converted into signals to the guilty process.

Each person authorized to use a system is assigned a **UID** (**User IDentification** ) by the system administrator. Every process started has the UID of the person who started it. A

child process has the same UID as its parent. Users can be members of groups, each of which has a **GID** (**Group IDentification** ).

One UID, called the **superuser** (in UNIX), has special power and may violate many of the protection rules. In large installations, only the system administrator knows the password needed to become superuser, but many of the ordinary users (especially students) devote considerable effort to trying to find flaws in the system that allow them to become superuser without the password.

We will study processes, interprocess communication, and related issues in Chap. 2. **1.5.2 Deadlocks**

When two or more processes are interacting, they can sometimes get themselves into a stalemate situation they cannot get out of. Such a situation is called a deadlock.

Deadlocks can best be introduced with a real-world example everyone is familiar with, deadlock in traffic. Consider the situation of Fig. 1-13(a). Here four buses are approaching an intersection. Behind each one are more buses (not shown). With a little bit of bad luck, the first four could all arrive at the intersection simultaneously, leading to the situation of Fig. 1-13(b), in which they are deadlocked because none of them can go forward. Each one is blocking one of the others. They cannot go backward due to other buses behind them. There is no easy way out.

**Figure 1-13.** (a) A potential deadlock. (b) An actual deadlock.

Processes in a computer can experience an analogous situation in which they cannot

make any progress. For example, imagine a computer with a tape drive and CD recorder. Now imagine that two processes each need to produce a CD-ROM from data on a tape. Process 1 requests and is granted the tape drive. Next process 2 requests and is granted the CD-recorder. Then process 1 requests the CD-recorder and is suspended until process 2 returns it. Finally, process 2 requests the tape drive and is also suspended because process 1 already has it. Here we have a deadlock from which there is no escape. We will study deadlocks and what can be done about them in detail in Chap. 3.

**1.5.3 Memory Management**

Every computer has some main memory that it uses to hold executing programs. In a very simple operating system, only one program at a time is in memory. To run a second program, the first one has to be removed and the second one placed in memory.

More sophisticated operating systems allow multiple programs to be in memory at the same time. To keep them from interfering with one another (and with the operating system), some kind of protection mechanism is needed. While this mechanism has to be in the hardware, it is controlled by the operating system.

The above viewpoint is concerned with managing and protecting the computer’s main memory. A different, but equally important memory-related issue, is managing the address space of the processes. Normally, each process has some set of addresses it can use, typically running from 0 up to some maximum. In the simplest case, the maximum amount of address space a process has is less than the main memory. In this way, a process can fill up its address space and there will be enough room in main memory to hold it all.

However, on many computers addresses are 32 or 64 bits, giving an address space of 232 or 264 bytes, respectively. What happens if a process has more address space than the computer has main memory and the process wants to use it all? In the first computers, such a process was just out of luck. Nowadays, a technique called virtual memory exists, in which the operating system keeps part of the address space in main memory and part on disk and shuttles pieces back and forth between them as needed. This important operating system function, and other memory management-related functions will be covered in Chap. 4.

**1.5.4 Input/Output**

All computers have physical devices for acquiring input and producing output. After all, what good would a computer be if the users could not tell it what to do and could not get the results after it did the work requested. Many kinds of input and output devices exist, including keyboards, monitors, printers, and so on. It is up to the operating system to manage these devices.

Consequently, every operating system has an I/O subsystem for managing its I/O devices. Some of the I/O software is device independent, that is, applies to many or all I/O devices equally well. Other parts of it, such as device drivers, are specific to particular I/O devices. In Chap. 5 we will have a look at I/O software.

**1.5.5 Files**

Another key concept supported by virtually all operating systems is the file system. As noted before, a major function of the operating system is to hide the peculiarities of the disks and other I/O devices and present the programmer with a nice, clean abstract model of device-independent files. System calls are obviously needed to create files, remove files, read files, and write files. Before a file can be read, it must be located on the disk and opened, and after it has been read it should be closed, so calls are provided to do these things.

To provide a place to keep files, most operating systems have the concept of a **directory** as a way of grouping files together. A student, for example, might have one directory for each course he is taking (for the programs needed for that course), another directory for his electronic mail, and still another directory for his World Wide Web home page. System calls are then needed to create and remove directories. Calls are also provided to put an existing file in a directory, and to remove a file from a directory. Directory entries may be either files or other directories. This model also gives rise to a hierarchy—the file system—as shown in Fig. 1-14.

The process and file hierarchies both are organized as trees, but the similarity stops there. Process hierarchies usually are not very deep (more than three levels is unusual), whereas file hierarchies are commonly four, five, or even more levels deep. Process hierarchies are typically short-lived, generally a few minutes at most, whereas the directory hierarchy may exist for years. Ownership and protection also differ for processes and files. Typically, only a parent process may control or even access a child process, but mechanisms nearly always exist to allow files and directories to be read by

a wider group than just the owner.

**Figure 1-14.** A file system for a university department.

Every file within the directory hierarchy can be specified by giving its **path name** from the top of the directory hierarchy, the **root directory.** Such absolute path names consist of the list of directories that must be traversed from the root directory to get to the file, with slashes separating the components. In Fig. 1-14, the path for file *CS101* is

*/Faculty/Prof.Brown/Courses/CS101.* The leading slash indicates that the path is absolute, that is, starting at the root directory. As an aside, in MS-DOS and Windows, the backslash (\) character is used as the separator instead of the slash (/) character, so the file path given above would be written as *\Faculty\Prof.Brown\Courses\CS101.* Throughout this book we will generally use the UNIX convention for paths.

At every instant, each process has a current **working directory,** in which path names not beginning with a slash are looked for. As an example, in Fig. 1-14, if */Faculty/Prof.Brown* were the working directory, then use of the path name *Courses/CS101* would yield the same file as the absolute path name given above.

Processes can change their working directory by issuing a system call specifying the new working directory.

Before a file can be read or written, it must be opened, at which time the permissions are checked. If the access is permitted, the system returns a small integer called a **file descriptor** to use in subsequent operations. If the access is prohibited, an error code is returned.

Another important concept in UNIX is the mounted file system. Nearly all personal computers have one or more floppy disk drives into which floppy disks can be inserted and removed. To provide an elegant way to deal with removable media (including CD ROMs), UNIX allows the file system on a floppy disk to be attached to the main tree. Consider the situation of Fig. 1-15(a). Before the mount call, the **root file system** , on the hard disk, and a second file system, on a floppy disk, are separate and unrelated.



**Figure 1-15.** (a) Before mounting, the files on drive 0 are not accessible. (b) After mounting, they are part of the file hierarchy.

However, the file system on the floppy cannot be used, because there is no way to specify path names on it, UNIX does not allow path names to be prefixed by a drive name or number; that would be precisely the kind of device dependence that operating systems ought to eliminate. Instead, the mount system call allows the file system on the floppy to be attached to the root file system wherever the program wants it to be. In Fig. 1-15(b) the file system on the floppy has been mounted on directory *b,* thus allowing access to files */b/x* and */b/y.* If directory *b* had contained any files they would not be

accessible while the floppy was mounted, since /*b* would refer to the root directory of the floppy. (Not being able to access these files is not as serious as it at first seems: file systems are nearly always mounted on empty directories.) If a system contains multiple hard disks, they can all be mounted into a single tree as well.

Another important concept in UNIX is the **special file.** Special files are provided in order to make I/O devices look like files. That way, they can be read and written using the same system calls as are used for reading and writing files. Two kinds of special files exist: **block special files** and **character special files.** Block special files are used to model devices that consist of a collection of randomly addressable blocks, such as disks. By opening a block special file and reading, say, block 4, a program can directly access the fourth block on the device, without regard to the structure of the file system contained on it. Similarly, character special files are used to model printers, modems, and other devices that accept or output a character stream. By convention, the special files are kept in the */dev* directory. For example, */dev/lp* might be the line printer.

The last feature we will discuss in this overview is one that relates to both processes and files: pipes. A **pipe** is a sort of pseudofile that can be used to connect two processes, as shown in Fig. 1-16. If processes *A* and *B* wish to talk using a pipe, they must set it up in advance. When process *A* wants to send data to process *B,* it writes on the pipe as though it were an output file. Process *B* can read the data by reading from the pipe as though it were an input file. Thus, communication between processes in UNIX looks very much like ordinary file reads and writes. Stronger yet, the only way a process can discover that the output file it is writing on is not really a file, but a pipe, is by making a special system call. File systems are very important. We will have much more to say about them in Chap. 6 and also in Chaps. 10 and 11.



**Figure 1-16.** Two processes connected by a pipe.

**1.5.6 Security**

Computers contain large amounts of information that users often want to keep confidential. This information may include electronic mail, business plans, tax returns, and much more. It is up to the operating system to manage the system security so that files, for example, are only accessible to authorized users.

As a simple example, just to get an idea of how security can work, consider UNIX. Files in UNIX are protected by assigning each one a 9-bit binary protection code. The protection code consists of three 3-bit fields, one for the owner, one for other members of the owner’s group (users are divided into groups by the system administrator), and

one for everyone else. Each field has a bit for read access, a bit for write access, and a bit for execute access. These 3 bits are known as the rwx bits. For example, the protection code *rwxr-x--x* means that the owner can read, write, or execute the file, other group members can read or execute (but not write) the file, and everyone else can execute (but not read or write) the file. For a directory, *x* indicates search permission. A dash means that the corresponding permission is absent.

In addition to file protection, there are many other security issues. Protecting the system from unwanted intruders, both human and nonhuman (e.g., viruses) is one of them. We will look at various security issues in Chap. 9.

**1.5.7 The Shell**

The operating system is the code that carries out the system calls. Editors, compilers, assemblers, linkers, and command interpreters definitely are not part of the operating system, even though they are important and useful. At the risk of confusing things somewhat, in this section we will look briefly at the UNIX command interpreter, called the **shell.** Although it is not part of the operating system, it makes heavy use of many operating system features and thus serves as a good example of how the system calls can be used. It is also the primary interface between a user sitting at his terminal and the operating system, unless the user is using a graphical user interface. Many shells exist, including *sh, csh, ksh,* and *bash.* All of them support the functionality described below, which derives from the original shell (*sh* )*.*

When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the **prompt,** a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user now types

date

for example, the shell creates a child process and runs the *date* program as the child. While the child process is running, the shell waits for it to terminate. When the child finishes, the shell types the prompt again and tries to read the next input line.

The user can specify that standard output be redirected to a file, for example, date >file

Similarly, standard input can be redirected, as in

sort <file1 >file2

which invokes the sort program with input taken from *file1* and output sent to *file2.*

The output of one program can be used as the input for another program by connecting them with a pipe. Thus

cat file1 file2 file3 | sort >/dev/lp

invokes the *cat* program to concatenate three files and send the output to *sort* to arrange all the lines in alphabetical order. The output of *sort* is redirected to the file */dev/lp* , typical1y the printer.

If a user puts an ampersand after a command, the shell does not wait for it to complete. Instead it just gives a prompt immediately. Consequently,

cat file1 file2 file3 | sort >/dev/lp &

starts up the sort as a background job, allowing the user to continue working normally while the sort is going on. The shell has a number of other interesting features, which we do not have space to discuss here. Most books on UNIX discuss the shell at some length (e.g., Kernighan and Pike, 1984; Kochan and Wood 1990; Medinets, 1999; Newham and Rosenblatt, 1998: and Robbins, 1999).

**1.5.8 Recycling of Concepts**

Computer science, like many fields, is largely technology driven. The reason the ancient Romans lacked cars is not that they liked walking so much. It is because they did not know how to build cars. Personal computers exist *not* because millions of people had some long pent-up desire to own a computer, but because it is now possible to manufacture them cheaply. We often forget how much technology affects our view of systems and it is worth reflecting on this point from time to time.

In particular, it frequently happens that a change in technology renders some idea obsolete and it quickly vanishes. However, another change in technology could revive it again. This is especially true when the change has to do with the relative performance of different parts of the system. For example, when CPUs became much faster than memories, caches became important to speed up the “slow” memory. If new memory technology some day makes memories much faster than CPUs, caches will vanish. And

if a new CPU technology makes them faster than memories again, caches will reappear. In biology, extinction is forever, but in computer science, it is sometimes only for a few years.

As a consequence of this impermanence, in this book we will from time to time look at “obsolete” concepts, that is, ideas that are not optimal with current technology. However, changes in the technology may bring back some of the so-called “obsolete concepts.” For this reason, it is important to understand why a concept is obsolete and what changes in the environment might bring it back again.

To make this point clearer, let us consider a few examples. Early computers had hardwired instruction sets. The instructions were executed directly by hardware and could not be changed. Then came microprogramming, in which an underlying interpreter carried out the instructions in software. Hardwired execution became obsolete. Then RISC computers were invented, and microprogramming (i.e., interpreted execution) became obsolete because direct execution was faster. Now we are seeing the resurgence of interpretation in the form of Java applets that are sent over the Internet and interpreted upon arrival. Execution speed is not always crucial because network delays are so great that they tend to dominate. But that could change, too, some day.

Early operating systems allocated files on the disk by just placing them in contiguous sectors, one after another. Although this scheme was easy to implement, it was not flexible because when a file grew, there was not enough room to store it any more. Thus the concept of contiguously allocated files was discarded as obsolete. Until CD-ROMs came around. There the problem of growing files did not exist. All of a sudden, the simplicity of contiguous file allocation was seen as a great idea and CD-ROM file systems are now based on it.

As our final idea, consider dynamic linking, The MULTICS system was designed to run day and night without ever stopping. To fix bugs in software, it was necessary to have a way to replace library procedures while they were being used. The concept of dynamic linking was invented for this purpose. After MULTICS died, the concept was forgotten for a while. However, it was rediscovered when modern operating systems needed a way to allow many programs to share the same library procedures without having their own private copies (because graphics libraries had grown so large). Most systems now support some form of dynamic linking once again. The list goes on, but these examples should make the point: an idea that is obsolete today may be the star of the party tomorrow.

Technology is not the only factor that drives systems and software. Economics plays a big role too. In the 1960s and 1970s, most terminals were mechanical printing terminals or 25 80 character-oriented CRTs rather than bitmap graphics terminals. This choice was not a question of technology. Bit-map graphics terminals were in use before 1960. It is just that they cost many tens of thousands of dollars each. Only when the price came down enormously could people (other than the military) think of dedicating one terminal to an individual user.

**1.6 SYSTEM CALLS**

The interface between the operating system and the user programs is defined by the set of system calls that the operating system provides. To really understand what operating systems do, we must examine this interface closely. The system calls available in the interface vary from operating system to operating system (although the underlying concepts tend to be similar).

We are thus forced to make a choice between (1) vague generalities (“operating systems have system calls for reading files”) and (2) some specific system (“UNIX has a read system call with three parameters: one to specify the file, one to tell where the data are to be put, and one to tell how many bytes to read”).

We have chosen the latter approach. It’s more work that way, but it gives more insight into what operating systems really do. Although this discussion specifically refers to POSIX (International Standard 9945-1), hence also to UNIX, System V, BSD, Linux, MINIX, etc., most other modern operating systems have system calls that perform the same functions, even if the details differ. Since the actual mechanics of issuing a system call are highly machine dependent and often must be expressed in assembly code, a procedure library is provided to make it possible to make system calls from C programs and often from other languages as well.

It is useful to keep the following in mind. Any single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a trap or system call instruction to transfer control to the operating system. The operating system then figures out what the calling process wants by inspecting the parameters. Then it carries out the system call and returns control to the instruction following the system call. In a sense,

making a system call is like making a special kind of procedure call, only system calls enter the kernel and procedure calls do not.

To make the system call mechanism clearer, let us take a quick look at the read system call. As mentioned above, it has three parameters: the first one specifying the file, the second one pointing to the buffer, and the third one giving the number of bytes to read. Like nearly all system calls, it is invoked from C programs by calling a library procedure with the same name as the system call: *read.* A call from a C program might look like this:

count = read(fd, buffer, nbytes);

The system call (and the library procedure) return the number of bytes actually read in *count.* This value is normally the same as *nbytes,* but may be smaller, if, for example, end-of-file is encountered while reading.

If the system call cannot be carried out, either due to an invalid parameter or a disk error, *count* is set to -1, and the error number is put in a global variable, *errno.* Programs should always check the results of a system call to see if an error occurred.

System calls are performed in a series of steps. To make this concept clearer, let us examine the read call discussed above. In preparation for calling the *read* library procedure, which actually makes the read system call, the calling program first pushes the parameters onto the stack, as shown in steps 1-3 in Fig. 1-17. C and C++ compilers push the parameters onto the stack in reverse order for historical reasons (having to do with making the first parameter to *printf,* the format string, appear on top of the stack). The first and third parameters are called by value, but the second parameter is passed by reference, meaning that the address of the buffer (indicated by &) is passed, not the contents of the buffer. Then comes the actual call to the library procedure (step 4). This instruction is the normal procedure call instruction used to call all procedures.

The library procedure, possibly written in assembly language, typically puts the system call number in a place where the operating system expects it, such as a register (step 5). Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel (step 6). The kernel code that starts examines the system call number and then dispatches to the correct system call handler, usually via a table of pointers to system call handlers indexed on system call number (step 7). At that point the system call handler runs (step 8). Once the system call handler