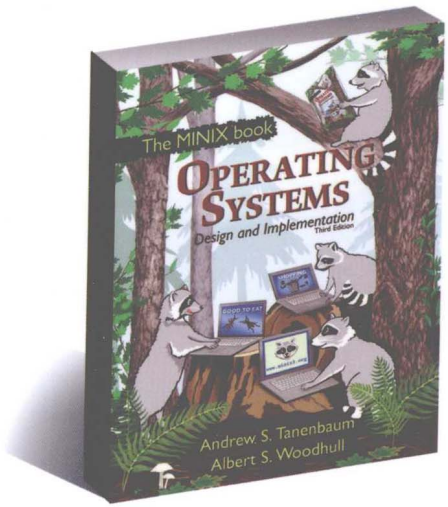
PEARSON
Prentice
Hall

大学计算机教育国外著名教材系列（影印版）

Operating Systems
Design and Implementation Third Edition

操作系统 设计与实现（第3版）

Andrew S. Tanenbaum
Albert S. Woodhull 著



PEARSON
Education
清华大学出版社

大学计算机教育国外著名教材系列（影印版）

Operating Systems

Design and Implementation

Third Edition

操 作 系 统

设计与实现

（第3版）

Andrew S. Tanenbaum

Albert S. Woodhull

清华大学出版社
北 京

北京市版权局著作权合同登记号　图字 01-2008-0461

图书在版编目(CIP)数据

操作系统：设计与实现 = Operating Systems: Design and Implementation: 第 3 版: 英文 /（美）特尼博姆（Tanenbaum, A.S.）等著. —影印本. —北京：清华大学出版社，2008.5

（大学计算机教育国外著名教材系列）

ISBN 978-7-302-17276-5

Ⅰ. 操… Ⅱ. 特… Ⅲ. 操作系统－高等学校－教材－英文 Ⅳ. TP316

中国版本图书馆 CIP 数据核字（2008）第 042081 号

责任印制：孟凡玉

出版发行：清华大学出版社　　**地　　址：**北京清华大学学研大厦 A 座

http://www.tup.com.cn　　**邮　　编：**100084

社　总　机：010-62770175　　**邮　　购：**010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质　量　反　馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者：北京鑫海金澳胶印有限公司

装 订 者：北京市密云县京文制本装订厂

发 行 者：全国新华书店

开　　本：185×230　　**印张：**39.75

版　　次：2008 年 5 月第 1 版　　**印　　次：**2008 年 5 月第 1 次印刷

印　　数：1~3000

定　　价：69.00 元（含光盘）

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：010-62770177 转 3103　　产品编号：026650-01

PREFACE

Most books on operating systems are strong on theory and weak on practice. This one aims to provide a better balance between the two. It covers all the fundamental principles in great detail, including processes, interprocess communication, semaphores, monitors, message passing, scheduling algorithms, input/output, deadlocks, device drivers, memory management, paging algorithms, file system design, security, and protection mechanisms. But it also discusses one particular system—MINIX 3—a UNIX-compatible operating system in detail, and even provides a source code listing for study. This arrangement allows the reader not only to learn the principles, but also to see how they are applied in a real operating system.

When the first edition of this book appeared in 1987, it caused something of a small revolution in the way operating systems courses were taught. Until then, most courses just covered theory. With the appearance of MINIX, many schools began to have laboratory courses in which students examined a real operating system to see how it worked inside. We consider this trend highly desirable and hope it continues.

It its first 10 years, MINIX underwent many changes. The original code was designed for a 256K 8088-based IBM PC with two diskette drives and no hard disk. It was also based on UNIX Version 7 As time went on, MINIX evolved in many ways: it supported 32-bit protected mode machines with large memories and hard disks. It also changed from being based on Version 7, to being based on the international POSIX standard (IEEE 1003.1 and ISO 9945-1). Finally, many

new features were added, perhaps too many in our view, but too few in the view of some other people, which led to the creation of Linux. In addition, MINIX was ported to many other platforms, including the Macintosh, Amiga, Atari, and SPARC. A second edition of the book, covering this system, was published in 1997 and was widely used at universities.

The popularity of MINIX has continued, as can be observed by examining the number of hits for MINIX found by Google.

This third edition of the book has many changes throughout. Nearly all of the material on principles has been revised, and considerable new material has been added. However, the main change is the discussion of the new version of the system, called MINIX 3. and the inclusion of the new code in this book. Although loosely based on MINIX 2, MINIX 3 is fundamentally different in many key ways.

The design of MINIX 3 was inspired by the observation that operating systems are becoming bloated, slow, and unreliable. They crash far more often than other electronic devices such as televisions, cell phones, and DVD players and have so many features and options that practically nobody can understand them fully or manage them well. And of course, computer viruses, worms, spyware, spam, and other forms of malware have become epidemic.

To a large extent, many of these problems are caused by a fundamental design flaw in current operating systems: their lack of modularity. The entire operatng system is typically millions of lines of C/C++ code compiled into a single massive executable program run in kernel mode. A bug in any one of those millions of lines of code can cause the system to malfunction. Getting all this code correct is impossible, especially when about 70% consists of device drivers, written by third parties, and outside the purview of the people maintaining the operating system.

With MINIX 3, we demonstrate that this monolithic design is not the only possibility. The MINIX 3 kernel is only about 4000 lines of executable code, not the millions found in Windows, Linux, Mac OS X, or FreeBSD. The rest of the system, including all the device drivers (except the clock driver), is a collection of small, modular, user-mode processes, each of which is tightly restricted in what it can do and with which other processes it may communicate.

While MINIX 3 is a work in progress, we believe that this model of building an operating system as a collection of highly-encapsulated user-mode processes holds promise for building more reliable systems in the future. MINIX 3 is especially focused on smaller PCs (such as those commonly found in Third-World countries and on embedded systems, which are always resource constrained). In any event, this design makes it much easier for students to learn how an operating system works than attempting to study a huge monolithic system.

The CD-ROM that is included in this book is a live CD. You can put it in your CD-ROM drive, reboot the computer, and MINIX 3 will give a login prompt within a few seconds. You can log in as *root* and give the system a try without first having to install it on your hard disk. Of course, it can also be installed on the hard disk. Detailed installation instructions are given in Appendix A.

As suggested above, MINIX 3 is rapidly evolving, with new versions being issued frequently. To download the current CD-ROM image file for burning, please go to the official Website: *www.minix3.org*. This site also contains a large amount of new software, documentation, and news about MINIX 3 development. For discussions about MINIX 3, or to ask questions, there is a USENET newsgroup: *comp.os.minix*. People without newsreaders can follow discussions on the Web at *http://groups.google.com/group/comp.os.minix*.

As an alternative to installing MINIX 3 on your hard disk, it is possible to run it on any one of several PC simulators now available. Some of these are listed on the main page of the Website.

Instructors who are using the book as the text for a university course can get the problem solutions from their local Prentice Hall representative. The book has its own Website. It can be found by going to *www.prenhall.com/tanenbaum* and selecting this title.

We have been extremely fortunate in having the help of many people during the course of this project. First and foremost, Ben Gras and Jorrit Herder have done most of the programming of the new version. They did a great job under tight time constraints, including responding to e-mail well after midnight on many occasions. They also read the manuscript and made many useful comments. Our deepest appreciation to both of them.

Kees Bot also helped greatly with previous versions, giving us a good base to work with. Kees wrote large chunks of code for versions up to 2.0.4, repaired bugs, and answered numerous questions. Philip Homburg wrote most of the networking code as well as helping out in numerous other useful ways, especially providing detailed feedback on the manuscript.

People too numerous to list contributed code to the very early versions, helping to get MINIX off the ground in the first place. There were so many of them and their contributions have been so varied that we cannot even begin to list them all here, so the best we can do is a generic thank you to all of them.

Several people read parts of the manuscript and made suggestions. We would like to give our special thanks to Gojko Babic, Michael Crowley, Joseph M. Kizza, Sam Kohn Alexander Manov, and Du Zhang for their help.

Finally, we would like to thank our families. Suzanne has been through this 16 times now. Barbara has been through it 15 times now. Marvin has been through it 14 times now. It's kind of getting to be routine, but the love and support is still much appreciated. (AST)

Al's Barbara has been through this twice now. Her support, patience, and good humor were essential. Gordon has been a patient listener. It is still a delight to have a son who understands and cares about the things that fascinate me. Finally, step-grandson Zain's first birthday coincides with the release of MINIX 3. Some day he will appreciate this. (ASW)

Andrew S. Tanenbaum
Albert S. Woodhull

CONTENTS

PREFACE xi

1 INTRODUCTION 1

1.1 WHAT IS AN OPERATING SYSTEM? 4
- 1.1.1 The Operating System as an Extended Machine 4
- 1.1.2 The Operating System as a Resource Manager 5

1.2 HISTORY OF OPERATING SYSTEMS 6
- 1.2.1 The First Generation (1945–55) Vacuum Tubes and Plugboards 7
- 1.2.2 The Second Generation (1955–65) Transistors and Batch Systems 7
- 1.2.3 The Third Generation (1965–1980) ICs and Multiprogramming 9
- 1.2.4 The Fourth Generation (1980–Present) Personal Computers 13
- 1.2.5 History of MINIX 3 14

1.3 OPERATING SYSTEM CONCEPTS 17
- 1.3.1 Processes 18
- 1.3.2 Files 20
- 1.3.3 The Shell 23

1.4 SYSTEM CALLS 24
- 1.4.1 System Calls for Process Management 25
- 1.4.2 System Calls for Signaling 29
- 1.4.3 System Calls for File Management 31
- 1.4.4 System Calls for Directory Management 36
- 1.4.5 System Calls for Protection 38
- 1.4.6 System Calls for Time Management 40

1.5 OPERATING SYSTEM STRUCTURE 40
1.5.1 Monolithic Systems 40
1.5.2 Layered Systems 43
1.5.3 Virtual Machines 44
1.5.4 Exokernels 47
1.5.5 Client-Server Model 47

1.6 OUTLINE OF THE REST OF THIS BOOK 49

1.7 SUMMARY 49

2 PROCESSES 53

2.1 INTRODUCTION TO PROCESSES 53
2.1.1 The Process Model 54
2.1.2 Process Creation 55
2.1.3 Process Termination 57
2.1.4 Process Hierarchies 58
2.1.5 Process States 58
2.1.6 Implementation of Processes 60
2.1.7 Threads 62

2.2 INTERPROCESS COMMUNICATION
2.2.1 Race Conditions 67
2.2.2 Critical Sections 68
2.2.3 Mutual Exclusion with Busy Waiting 69
2.2.4 Sleep and Wakeup 74
2.2.5 Semaphores 76
2.2.6 Mutexes 79
2.2.7 Monitors 79
2.2.8 Message Passing 83

2.3 CLASSICAL IPC PROBLEMS 86
2.3.1 The Dining Philosophers Problem 87
2.3.2 The Readers and Writers Problem 90

2.4 SCHEDULING 91
2.4.1 Introduction to Scheduling 92
2.4.2 Scheduling in Batch Systems 97
2.4.3 Scheduling in Interactive Systems 100
2.4.4 Scheduling in Real-Time Systems 107
2.4.5 Policy versus Mechanism 108
2.4.6 Thread Scheduling 108

2.5 OVERVIEW OF PROCESSES IN MINIX 3 110
2.5.1 The Internal Structure of MINIX 3 110
2.5.2 Process Management in MINIX 3 114
2.5.3 Interprocess Communication in MINIX 3 118
2.5.4 Process Scheduling in MINIX 3 120

2.6 IMPLEMENTATION OF PROCESSES IN MINIX 3 123
2.6.1 Organization of the MINIX 3 Source Code 123
2.6.2 Compiling and Runniing MINIX 3 126
2.6.3 The Common Header Files 128
2.6.4 The MINIX 3 Header Files 136
2.6.5 Process Data Structures and Header Files 144
2.6.6 Bootstrapping MINIX 3 154
2.6.7 System Initialization 158
2.6.8 Interrupt Handling in MINIX 3 165
2.6.9 Interprocess Communication in MINIX 3 176
2.6.10 Scheduling in MINIX 3 180
2.6.11 Hardware-Dependent Kernel Support 183
2.6.12 Utilities and the Kernel Library 188

2.7 THE SYSTEM TASK IN MINIX 3 190
2.7.1 Overview of the System Task 192
2.7.2 Implementation of the System Task 195
2.7.3 Implementation of the System Libarary 198

2.9 THE CLOCK TASK IN MINIX 3 202
2.8.1 Clock Hardware 202
2.8.2 Clock Software 204
2.8.3 Overview of the Clock Driver in MINIX 3 206
2.8.4 Implementation of the Clock Driver in MINIX 3 210

2.9 SUMMARY 212

3 INPUT/OUTPUT 219

3.1 PRINCIPLES OF I/O HARDWARE 220
3.1.1 I/O Devices 221
3.1.2 Device Controllers 221
3.1.3 Memory-Mapped I/O 223
3.1.4 Interrupts 224
3.1.5 Direct Memory Access 225

3.2 PRINCIPLES OF I/O SOFTWARE 227
3.2.1 Goals of the I/O Software 227
3.2.2 Interrupt Handlers 229
3.2.3 Device Drivers 220
3.2.4 Device-Independent I/O Software 231
3.2.5 User-Space I/O Software 234

3.3 DEADLOCKS 235
3.3.1 Resources 236
3.3.2 Principles of Deadlock 237
3.3.3 The Ostrich Algorithm 240
3.3.4 Detection and Recovery 242
3.3.5 Deadlock Prevention 243
3.3.6 Deadlock Avoidance 245

3.4 OVERVIEW OF I/O IN MINIX 3 250
3.4.1 Interrupt Handlers in MINIX 3 250
3.4.2 Device Drivers in MINIX 3 254
3.4.3 Device-Independent I/O Software in MINIX 3 257
3.4.4 User-level I/O Software in MINIX 3 258
3.4.5 Deadlock Handling in MINIX 3 258

3.5 BLOCK DEVICES IN MINIX 3 259
3.5.1 Overview of Block Device Drivers in MINIX 3 260
3.5.2 Common Block Device Driver Software 263
3.5.3 The Driver Library 267

3.6 RAM DISKS 269
3.6.1 RAM Disk Hardware and Software 269
3.6.2 Overview of the RAM Disk Driver in MINIX 3 271
3.6.3 Implementation of the RAM Disk Driver in MINIX 3 272

3.7 DISKS 276
3.7.1 Disk Hardware 276
3.7.2 RAID 278
3.7.3 Disk Software 279
3.7.4 Overview of the Hard Disk Driver in MINIX 3 285
3.7.5 Implementation of the Hard Disk Driver in MINIX 3 288
3.7.6 Floppy Disk Handling 298

3.8 TERMINALS 300
3.8.1 Terminal Hardware 301
3.8.2 Terminal Software 305
3.8.3 Overview of the Terminal Driver in MINIX 3 314
3.8.4 Implementation of the Device-Independent Terminal Driver 329

3.8.5 Implementation of the Keyboard Driver 348
3.8.6 Implementation of the Display Driver 355

3.9 SUMMARY 364

4 MEMORY MANAGEMENT 371

4.1 BASIC MEMORY MANAGEMENT 372
4.1.1 Monoprogramming without Swapping or Paging 372
4.1.2 Multiprogramming with Fixed Partitions 373
4.1.3 Relocation and Protection 375

4.2 SWAPPING 376
4.2.1 Memory Management with Bitmaps 378
4.2.2 Memory Management with Linked Lists 379

4.3 VIRTUAL MEMORY 381
4.3.1 Paging 382
4.3.2 Page Tables 386
4.3.3 TLBs—Translation Lookaside Buffers 390
4.3.4 Inverted Page Tables 393

4.4 PAGE REPLACEMENT ALGORITHMS 394
4.4.1 The Optimal Page Replacement Algorithm 395
4.4.2 The Not Recently Used Page Replacement Algorithm 396
4.4.3 The First-In, First-Out (FIFO) Page Replacement Algorithm 397
4.4.4 The Second Chance Page Replacement Algorithm 397
4.4.5 The Clock Page Replacement Algorithm 398
4.4.6 The Least Recently Used (LRU) Page Replacement Algorithm 399
4.4.7 Simulating LRU in Software 399

4.5 DESIGN ISSUES FOR PAGING SYSTEMS 402
4.5.1 The Working Set Model 402
4.5.2 Local versus Global Allocation Policies 404
4.5.3 Page Size 406
4.5.4 Virtual Memory Interface 408

4.6 SEGMENTATION 408
4.6.1 Implementation of Pure Segmentation 412
4.6.2 Segmentation with Paging: The Intel Pentium 413

4.7 OVERVIEW OF THE MINIX 3 PROCESS MANAGER 418
4.7.1 Memory Layout 420
4.7.2 Message Handling 423
4.7.3 Process Manager Data Structures and Algorithms 424
4.7.4 The FORK, EXIT, and WAIT System Calls 430
4.7.5 The EXEC System Call 431
4.7.6 The BRK System Call 435
4.7.7 Signal Handling 436
4.7.8 Other System Calls 444

4.8 IMPLEMENTATION OF THE MINIX 3 PROCESS MANAGER 445
4.8.1 The Header Files and Data Structures 445
4.8.2 The Main Program 448
4.8.3 Implementation of FORK, EXIT, and WAIT 453
4.8.4 Implementation of EXEC 456
4.8.5 Implementation of BRK 459
4.8.6 Implementation of Signal Handling 460
4.8.7 Implementation of Other System Calls 469
4.8.8 Memory Management Utilities 471

4.9 SUMMARY 473

5 FILE SYSTEMS 479

5.1 FILES 480
5.1.1 File Naming 480
5.1.2 File Structure 482
5.1.3 File Types 483
5.1.4 File Access 486
5.1.5 File Attributes 486
5.1.6 File Operations 488

5.2 DIRECTORIES 489
5.2.1 Simple Directories 489
5.2.2 Hierarchical Directory Systems 490
5.2.3 Path Names 491
5.2.4 Directory Operations 494

5.3 FILE SYSTEM IMPLEMENTATION 495
5.3.1 File System Layout 495
5.3.2 Implementing Files 497
5.3.3 Implementing Directories 500
5.3.4 Disk Space Management 507

5.3.5 File System Reliability 510
5.3.6 File System Performance 521
5.3.7 Log-Structured File Systems 522

5.4 SECURITY 524
5.4.1 The Security Environment 524
5.4.2 Generic Security Attacks 529
5.4.3 Design Principles for Security 530
5.4.4 User Authentication 531

5.5 PROTECTION MECHANISMS 535
5.5.1 Protection Domains 535
5.5.2 Access Control Lists 537
5.5.3 Capabilities 540
5.5.4 Covert Channels 543

5.6 OVERVIEW OF THE MINIX 3 FILE SYSTEM 546
5.6.1 Messages 547
5.6.2 File System Layout 547
5.6.3 Bitmaps 551
5.6.4 I-Nodes 553
5.6.5 The Block Cache 555
5.6.6 Directories and Paths 557
5.6.7 File Descriptors 559
5.6.8 File Locking 561
5.6.9 Pipes and Special Files 561
5.6.10 An Example: The READ System Call 563

5.7 IMPLEMENTATION OF THE MINIX 3 FILE SYSTEM 564
5.7.1 Header Files and Global Data Structures 564
5.7.2 Table Management 568
5.7.3 The Main Program 577
5.7.4 Operations on Individual Files 581
5.7.5 Directories and Paths 589
5.7.6 Other System Calls 594
5.7.7 The I/O Device Interface 595
5.7.8 Additional System Call Support 601
5.7.9 File System Utilities 603
5.7.10 Other MINIX 3 Components 604

SUMMARY 604

6 BIBLIOGRAPHY 609

1

INTRODUCTION

Without its software, a computer is basically a useless lump of metal. With its software, a computer can store, process, and retrieve information; play music and videos; send e-mail, search the Internet; and engage in many other valuable activities to earn its keep. Computer software can be divided roughly into two kinds: system programs, which manage the operation of the computer itself, and application programs, which perform the actual work the user wants. The most fundamental system program is the **operating system**, whose job is to control all the computer's resources and provide a base upon which the application programs can be written. Operating systems are the topic of this book. In particular, an operating system called MINIX 3 is used as a model, to illustrate design principles and the realities of implementing a design.

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. If every programmer had to be concerned with how disk drives work, and with all the dozens of things that could go wrong when reading a disk block, it is unlikely that many programs could be written at all.

Many years ago it became abundantly clear that some way had to be found to shield programmers from the complexity of the hardware. The way that has evolved gradually is to put a layer of software on top of the bare hardware, to manage all parts of the system, and present the user with an interface or **virtual**

machine that is easier to understand and program. This layer of software is the operating system.

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 is the province of the electrical engineer.

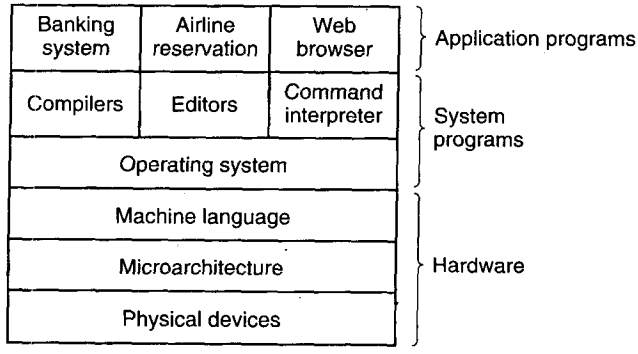


Figure 1-1. A computer system consists of hardware, system programs, and application programs.

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.

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)** 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 may be complex. Furthermore, for many I/O (Input/Output) devices, timing plays an important role in the programming.

A major function of the operating system is to hide all this complexity and give the programmer a more convenient set of instructions to work with. For example, read block from file is conceptually much 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 preinstalled by the computer manufacturer, or in a package with the operating system if it is installed after purchase. 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 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, including MINIX 3, 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. For example, in MINIX 3, the file system is simply a big C program running in user-mode.

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.

† "He" should be read as "he or she" throughout the book.

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 many 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.