

OPERATING SYSTEMS

DESIGN AND IMPLEMENTATION



Prentice-Hall International Editions

ANDREW S. TANENBAUM

OPERATING SYSTEMS:

Design

and

Implementation

Prentice-Hall Software Series
Brian W. Kernighan, Advisor

OPERATING SYSTEMS:

Design
and
Implementation

ANDREW S. TANENBAUM

*Vrije Universiteit
Amsterdam, The Netherlands*



Prentice-Hall International, Inc.

This edition may be sold only in those countries to which it is consigned by Prentice-Hall International. It is not to be re-exported and it is not for sale in the U.S.A., Mexico, or Canada.

© 1987 by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, NJ 07632

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9

ISBN 0-13-637331-3 025

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall, *Englewood Cliffs, New Jersey*

To Suzanne, Barbara, Marvin and the memory of Sweetie π

CONTENTS

PREFACE

xiv

1 INTRODUCTION

1

- 1.1 WHAT IS AN OPERATING SYSTEM? 3
 - 1.1.1 The Operating System as an Extended Machine 3
 - 1.1.2 The Operating System as a Resource Manager 4
- 1.2 HISTORY OF OPERATING SYSTEMS 5
 - 1.2.1 The First Generation (1945-1955): Vacuum Tubes and Plugboards 5
 - 1.2.2 The Second Generation (1955-1965): Transistors and Batch Systems 6
 - 1.2.3 The Third Generation (1965-1980): ICs and Multiprogramming 8
 - 1.2.4 The Fourth Generation (1980-1990): Personal Computers 11
 - 1.2.5 History of MINIX 13
- 1.3 OPERATING SYSTEM CONCEPTS 14
 - 1.3.1 Processes 15
 - 1.3.2 Files 16
 - 1.3.3 The Shell 20
- 1.4 SYSTEM CALLS 21
 - 1.4.1 System Calls for Process Management 23
 - 1.4.2 System Calls for Signaling 25
 - 1.4.3 System Calls for File Management 27
 - 1.4.4 System Calls for Directory Management 32
 - 1.4.5 System Calls for Protection 34
 - 1.4.6 System Calls for Time Management 36
- 1.5 OPERATING SYSTEM STRUCTURE 36
 - 1.5.1 Monolithic Systems 36
 - 1.5.2 Layered Systems 37
 - 1.5.3 Virtual Machines 39
 - 1.5.4 Client-Server Model 40

- 1.6 OUTLINE OF THE REST OF THIS BOOK 42
- 1.7 SUMMARY 42

2 PROCESSES

45

- 2.1 INTRODUCTION TO PROCESSES 45
 - 2.1.1 The Process Model 46
 - 2.1.2 Implementation of Processes 50
- 2.2 INTERPROCESS COMMUNICATION 51
 - 2.2.1 Race Conditions 51
 - 2.2.2 Critical Sections 53
 - 2.2.3 Mutual Exclusion with Busy Waiting 53
 - 2.2.4 Sleep and Wakeup 58
 - 2.2.5 Semaphores 60
 - 2.2.6 Event Counters 62
 - 2.2.7 Monitors 63
 - 2.2.8 Message Passing 67
 - 2.2.9 Equivalence of Primitives 72
- 2.3 CLASSICAL IPC PROBLEMS 75
 - 2.3.1 The Dining Philosophers Problem 75
 - 2.3.2 The Readers and Writers Problem 77
- 2.4 PROCESS SCHEDULING 80
 - 2.4.1 Round Robin Scheduling 81
 - 2.4.2 Priority Scheduling 82
 - 2.4.3 Multiple Queues 83
 - 2.4.4 Shortest Job First 84
 - 2.4.5 Policy-Driven Scheduling 85
 - 2.4.6 Two-level Scheduling 86
- 2.5 OVERVIEW OF PROCESSES IN MINIX 87
 - 2.5.1 The Internal Structure of MINIX 87
 - 2.5.2 Process Management in MINIX 88
 - 2.5.3 Interprocess Communication in MINIX 89
 - 2.5.4 Process Scheduling in MINIX 90
- 2.6 IMPLEMENTATION OF PROCESSES IN MINIX 90
 - 2.6.1 Organization of the MINIX Source Code 91
 - 2.6.2 The Common Header Files 92
 - 2.6.3 Process Data Structures and Header Files 95
 - 2.6.4 System Initialization 98
 - 2.6.5 Interrupt Handling in MINIX 99
 - 2.6.6 The Kernel's Assembly Code Utilities 101
 - 2.6.7 Interprocess Communication in MINIX 102
 - 2.6.8 Scheduling in MINIX 104
- 2.7 SUMMARY 105

3 INPUT/OUTPUT**110**

- 3.1 PRINCIPLES OF I/O HARDWARE 111
 - 3.1.1 I/O Devices 111
 - 3.1.2 Device Controllers 112
- 3.2 PRINCIPLES OF I/O SOFTWARE 116
 - 3.2.1 Goals of the I/O Software 116
 - 3.2.2 Interrupt Handlers 117
 - 3.2.3 Device Drivers 118
 - 3.2.4 Device-Independent I/O Software 119
 - 3.2.5 User-Space I/O Software 121
- 3.3 DEADLOCKS 122
 - 3.3.1 Resources 123
 - 3.3.2 Deadlock Modeling 124
 - 3.3.3 The Ostrich Algorithm 127
 - 3.3.4 Detection and Recovery 128
 - 3.3.5 Deadlock Prevention 128
 - 3.3.6 Deadlock Avoidance 130
- 3.4 OVERVIEW OF I/O IN MINIX 135
 - 3.4.1 Interrupt Handlers in MINIX 135
 - 3.4.2 Device Drivers in MINIX 135
 - 3.4.3 Device-Independent I/O Software in MINIX 138
 - 3.4.4 User-level I/O Software in MINIX 139
 - 3.4.5 Deadlock Handling in MINIX 139
- 3.5 RAM DISKS 139
 - 3.5.1 RAM Disk Hardware and Software 140
 - 3.5.2 Overview of the RAM Disk Driver in MINIX 141
 - 3.5.3 Implementation of the RAM Disk Driver in MINIX 142
- 3.6 DISKS 143
 - 3.6.1 Disk Hardware 143
 - 3.6.2 Disk Software 144
 - 3.6.3 Overview of the Floppy Disk Driver in MINIX 149
 - 3.6.4 Implementation of the Floppy Disk Driver in MINIX 151
- 3.7 CLOCKS 154
 - 3.7.1 Clock Hardware 154
 - 3.7.2 Clock Software 155
 - 3.7.3 Overview of the Clock Driver in MINIX 158
 - 3.7.4 Implementation of the Clock Driver in MINIX 159
- 3.8 TERMINALS 160
 - 3.8.1 Terminal Hardware 160
 - 3.8.2 Terminal Software 164
 - 3.8.3 Overview of the Terminal Driver in MINIX 171
 - 3.8.4 Implementation of the Terminal Driver in MINIX 177
- 3.9 THE SYSTEM TASK IN MINIX 182
- 3.10 SUMMARY 186

4	MEMORY MANAGEMENT	191
4.1	MEMORY MANAGEMENT WITHOUT SWAPPING OR PAGING	191
4.1.1	Monoprogramming without Swapping or Paging	192
4.1.2	Multiprogramming and Memory Usage	193
4.1.3	Multiprogramming with Fixed Partitions	196
4.2	SWAPPING	198
4.2.1	Multiprogramming with Variable Partitions	198
4.2.2	Memory Management with Bit Maps	200
4.2.3	Memory Management with Linked Lists	201
4.2.4	Memory Management with the Buddy System	203
4.2.5	Allocation of Swap Space	205
4.2.6	Analysis of Swapping Systems	205
4.3	VIRTUAL MEMORY	206
4.3.1	Paging	207
4.3.2	Segmentation	209
4.4	PAGE REPLACEMENT ALGORITHMS	213
4.4.1	Optimal Page Replacement	213
4.4.2	Not-Recently-Used Page Replacement	214
4.4.3	First-In First-Out Page Replacement	215
4.4.4	Least Recently Used Page Replacement	216
4.4.5	Simulating LRU in Software	217
4.5	DESIGN ISSUES FOR PAGING SYSTEMS	219
4.5.1	The Working Set Model	219
4.5.2	Local versus Global Allocation Policies	221
4.5.3	Page Size	222
4.5.4	Implementation Issues	224
4.6	OVERVIEW OF MEMORY MANAGEMENT IN MINIX	226
4.6.1	Memory Layout	227
4.6.2	Message Handling	229
4.6.3	Memory Manager Data Structures and Algorithms	230
4.6.4	The FORK, EXIT, and WAIT System Calls	233
4.6.5	The EXEC System Call	234
4.6.6	The BRK System Call	237
4.6.7	Signal Handling	237
4.6.8	Other System Calls	239
4.7	IMPLEMENTATION OF MEMORY MANAGEMENT IN MINIX	239
4.7.1	The Header Files	239
4.7.2	The Main Program	240
4.7.3	Implementation of FORK, EXIT, and WAIT	241
4.7.4	Implementation of EXEC	242
4.7.5	Implementation of BRK	243
4.7.6	Implementation of Signal Handling	244
4.7.7	Implementation of the Other System Calls	246
4.7.8	Memory Manager Utilities	246
4.8	SUMMARY	247

5 FILE SYSTEMS**251**

5.1 THE USER VIEW OF THE FILE SYSTEM	251
5.1.1 File Basics	252
5.1.2 Directories	254
5.2 FILE SYSTEM DESIGN	256
5.2.1 Disk Space Management	256
5.2.2 File Storage	258
5.2.3 Directory Structure	261
5.2.4 Shared Files	263
5.2.5 File System Reliability	266
5.2.6 File System Performance	270
5.3 FILE SERVERS	273
5.3.1 Interface Level	273
5.3.2 Atomic Update	274
5.3.3 Concurrency Control	276
5.3.4 Transactions	277
5.3.5 Replicated Files	279
5.4 SECURITY	279
5.4.1 The Security Environment	280
5.4.2 Famous Security Flaws	281
5.4.3 Generic Security Attacks	283
5.4.4 Design Principles for Security	284
5.4.5 User Authentication	285
5.5 PROTECTION MECHANISMS	289
5.5.1 Protection Domains	289
5.5.2 Access Control Lists	292
5.5.3 Capabilities	293
5.5.4 Protection Models	295
5.5.5 Covert Channels	297
5.6 OVERVIEW OF THE MINIX FILE SYSTEM	299
5.6.1 Messages	299
5.6.2 File System Layout	300
5.6.3 Bit Maps	302
5.6.4 I-nodes	304
5.6.5 The Block Cache	305
5.6.6 Directories and Paths	306
5.6.7 File Descriptors	308
5.6.8 Pipes and Special Files	309
5.6.9 An Example: The READ System Call	311
5.7 IMPLEMENTATION OF THE MINIX FILE SYSTEM	311
5.7.1 The Header Files	312
5.7.2 Table Management	313
5.7.3 The Main Program	318
5.7.4 Operations on Individual Files	319

5.7.5 Directories and Paths	325
5.7.6 Other System Calls	328
5.7.7 The I/O Device Interface	331
5.7.8 General Utilities	332
5.8 SUMMARY	332
6 READING LIST AND BIBLIOGRAPHY	337
6.1 SUGGESTIONS FOR FURTHER READING	337
6.1.1 Introduction and General Works	337
6.1.2 Processes	339
6.1.3 Input/Output	340
6.1.4 Memory Management	340
6.1.5 File Systems	341
6.2 ALPHABETICAL BIBLIOGRAPHY	342
APPENDICES	
A INTRODUCTION TO C	350
B INTRODUCTION TO THE IBM PC	363
C MINIX USERS' GUIDE	371
D MINIX IMPLEMENTERS' GUIDE	413
E MINIX SOURCE CODE LISTING	433
F MINIX CROSS REFERENCE LISTING	687
INDEX	711

give 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), 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 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 of the older 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 disk interrupt handler, which is part of the operating system and protected by hardware against attempts by users to modify it.

Finally, above the system programs come the application programs. These programs are written by the users to solve their particular problems, such as commercial data processing, engineering calculations, or game playing.

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, 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 controller chip, which is used on the IBM PC and many other 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

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

4.1.3. Multiprogramming with Fixed Partitions

By now it should be clear that it is often useful to have more than one process in memory at once. The question is then: "How should memory be organized to achieve this goal?" The easiest way is simply to divide memory up into n (possibly unequal) partitions. This partitioning can, for example, be done manually by the operator when the system is started up.

When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it. Since the partitions are fixed in this scheme, any space in a partition not used by a job is lost. In Fig. 4-4(a) we see how this system of fixed partitions and separate input queues looks.

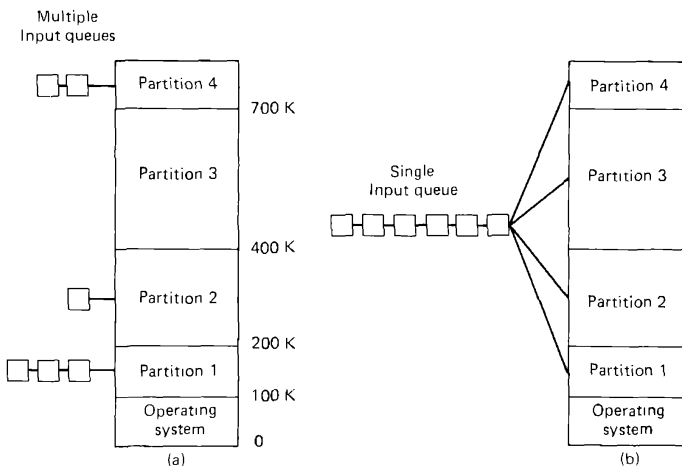


Fig. 4-4. (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

The disadvantage of sorting the incoming jobs into separate queues becomes apparent when the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 and 4 in Fig. 4-4(a). An alternative organization is to maintain a single queue as in Fig. 4-4(b). Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run. Since it is undesirable to waste a large partition on a small job, a different strategy is to search the whole input queue whenever a partition becomes free and pick the largest job that fits. Note that the latter algorithm discriminates against small jobs as being unworthy of having a whole partition, whereas usually it is desirable to give the smallest jobs the best service, not the worst.

This system, with fixed partitions set up by the operator in the morning and not changed thereafter, was used by OS/360 on large IBM mainframes for many

years. It was called MFT (Multiprogramming with a Fixed number of Tasks or OS/MFT). It is simple to understand and equally simple to implement: incoming jobs are queued until a suitable partition is available, at which time the job is loaded into that partition and run until it terminates.

Relocation and Protection

Multiprogramming introduces two essential problems that must be solved—relocation and protection. Look at Fig. 4-4. From the figure it is clear that different jobs will be run at different addresses. When a program is linked (i.e., the main program, user-written procedures, and library procedures are combined into a single address space), the linker must know at what address the program will begin in memory.

For example, suppose that the first instruction is a call to a procedure at relative address 100 within the binary file produced by the linker. If this program is loaded in partition 1, that instruction will jump to absolute address 100, which is inside the operating system. What is needed is a call to $100K + 100$. If the program is loaded into partition 2, it must be carried out as a call to $200K + 100$, and so on. This problem is known as the **relocation** problem.

One possible solution is to actually modify the instructions as the program is loaded into memory. Programs loaded into partition 1 have 100K added to each address, programs loaded into partition 2 have 200K added to addresses, and so forth. To perform relocation during loading like this, the linker must include in the binary program a list or bit map telling which program words are addresses to be relocated and which are opcodes, constants, or other items that must not be relocated. OS/MFT worked this way. Some microcomputers also work like this.

Relocation during loading does not solve the protection problem. A malicious program can always construct a new instruction and jump to it. Because programs in this system use absolute memory addresses rather than addresses relative to a register, there is no way to stop a program from building an instruction that reads or writes any word in memory. In multiuser systems, it is undesirable to let processes read and write memory belonging to other users.

The solution that IBM chose for protecting the 360 was to divide memory into blocks of 2K bytes and assign a 4-bit protection code to each block. The PSW contained a 4-bit key. The 360 hardware trapped any attempt by a running process to access memory whose protection code differed from the PSW key. Since only the operating system could change the protection codes and key, user processes were prevented from interfering with one another and with the operating system itself.

An alternative solution to both the relocation and protection problems is to equip the machine with two special hardware registers, called the **base** and **limit** registers. When a process is scheduled, the base register is loaded with the address of the start of its partition, and the limit register is loaded with the length of the partition. Every memory address generated automatically has the base

These are unabridged paperback
reprints of established titles widely used by
universities throughout the world.
Prentice-Hall International
publishes these lower-priced editions
for the benefit of students.



Prentice-Hall International Editions

This edition may be sold only in those countries
to which it is consigned by Prentice-Hall Interna-
tional. It is not to be re-exported, and is not for
sale in the U.S.A., Mexico, or Canada.



Prentice-Hall International

ISBN: 0-13-637331-3