Operating System Concepts

James L. Peterson
Abraham Silberschatz

Operating System Concepts

James L. Peterson
Abraham Silberschatz
University of Texas at Austin

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts · Menlo Park, California
London · Amsterdam · Don Mills, Ontario · Sydney

This book is in the Addison-Wesley series in Computer Science.

Consulting Editor
Michael A. Harrison

Library of Congress Cataloging in Publication Data

Peterson, James Lyle.
Operating system concepts.

Includes bibliographies.
1. Operating systems (Computers) I. Silberschatz, Abraham. II. Title.
QA76.6.P475 1982 001.64 82-22766
ISBN 0-201-06097-3

Reproduced by Addison-Wesley from camera-ready copy prepared by the authors.

ISBN 0-201-06097-3
ABCDEFGHIJ-AL-89876543

Preface

Operating systems are an essential part of a computer system. Similarly, a course on operating systems is an essential part of a computer science education. This book is intended as a text for an introductory course in operating systems at the junior, senior, or first-year graduate level. It provides a clear description of the *concepts* underlying operating systems.

This book is not centered around any particular operating system or hardware. Instead, it discusses fundamental concepts that are applicable to a variety of systems. Our emphasis is on solving the problems encountered in designing an operating system, regardless of the underlying hardware on which the system will run. We assume the reader is familiar with general assembly language programming and computer organization.

Content of this Book

The overall content of the book is as follows:

1 Introduction
2 Operating System Services
3 File Systems
4 CPU Scheduling
5 Memory Management
6 Virtual Memory
7 Disk and Drum Scheduling
8 Deadlocks
9 Concurrent Processes
10 Concurrent Programming
11 Protection
12 Design Principles
13 Distributed Systems
14 Historical Perspective

Chapters 1, 2, and 3 explain what operating systems *are* and what they *do*. These chapters explain how the concept of an operating system has developed, the common features of an operating system, what it does for the user, and what it does for the computer system operator. It is motivational, historical, and explanatory in nature. We do not deal with how things are performed internally in these chapters. Therefore, these chapters are suitable for the individuals or lower-level classes who want to learn what an operating system is, without getting into the details of the internal algorithms.

Chapters 4 to 8 deal with the classical internal algorithms and structures: *cpu scheduling, memory management,* and *device management.* They provide a firm practical understanding of the algorithms used, their properties, their advantages and disadvantages. The algorithms are presented in a natural order, so that new, more complicated systems can be built upon the understanding of simpler systems.

Chapter 9 introduces the unifying concept of the computer system as a collection of cooperating sequential processes. Chapters 10, 11, 12, and 13 present advanced topics and current trends, including high-level languages for writing concurrent programs, protection systems, design principles, and distributed systems. These topics are still being researched and may well need later revision. However, we include them in the book for two reasons. First, although research is still ongoing and final solutions to these problems are still being sought, there is general agreement that these topics are important and students should be exposed to them. Second, existing systems use these solutions, and anyone working with operating systems over the next five years will need to be aware of the developments in these directions.

Each chapter ends with references to further reading. Chapter 14 is essentially a set of references to further reading for the entire book, describing briefly some of the most influential operating systems.

Organization

Operating systems first began to appear in the late 1950's, and for twenty years underwent major changes in concepts and technology. As a result, the first-generation operating system textbooks that appeared during this period (Brinch Hansen [1973a], Madnick and Donovan [1974], Shaw [1974], Tsichritzis and Bernstein [1974]) tried to explain a subject that changed even as they were written.

Now, however, operating system theory and practice appears to have matured and stabilized. The fundamental operating system concepts are now well defined and well understood. While there will undoubtably be new algorithms, the basic approach to cpu scheduling,

memory management, the user interface, and so on, is not likely to change. Notice, for example, that there are few really new operating systems being written. Most large computers use operating systems that were designed in the 1960's. The newest operating systems are being developed for the multitude of microcomputer systems, but these are either CP/M, Unix, or imitations of these. It is now possible to write a book that presents well-understood, agreed-upon, classical operating system material.

This text is one of a second generation of operating system textbooks. Our text differs from other texts in the level of content and organization. The basic concepts have been carefully organized and presented, and the material flows naturally from these basic principles to more sophisticated ones.

The only controversial aspect of this text is its organization, specifically the definition of the formal process model as late as Chapter 9. Almost every other text places this material at the beginning as Chapter 2. In our experience, this arrangement does not work. The process model is a powerful and convenient unifying concept. However, when operating systems are first introduced, the student does not know the basic principles. To benefit from the process model, the student needs to understand how cpu scheduling and memory management can present an image of separate virtual processors, each with its own separate virtual memory space. Then, and only then, will the student really be able to understand why the process model is useful. Once the student has the proper background to be able to appreciate the process model of operating systems, the standard material concerning processes, process coordination, synchronization and communication is presented.

Concurrency itself, in the form of overlapped I/O, spooling, multiprogramming, and time-sharing, is introduced as early as Chapter 1. However, we feel that the *formal* process model is best reserved until the basic concepts (cpu scheduling and memory management) are well understood.

Acknowledgments

Eight years of CS 372 students at the University of Texas as Austin suffered through permutations of this material until we got it right. David Orshalick helped with the early table of contents. During the writing stage, we were invited to design and teach an operating system course for IBM, which helped clarify our organization.

As the text was written, Carol Engelhardt deciphered our handwriting and edited our text into Scribe format. Carol's efforts throughout this project were the only thing that got it done.

Jeff Ullman helped us to get draft copies on the Dover at Stanford. Arthur Keller and Gordon Novak helped get those drafts back to Texas. Susan Lilly was able to understand what we were trying to say in the drafts and edit them into readable text. Elaine Rich, Richard Cohen, and Brian Reid explained the subtleties of Scribe, helping us to define our documents and make them work. The manuscript was read in various forms by Michael Molloy, Gael Buckley, and the reviewers.

Finally, the entire book was translated (by Scribe) into troff and phototypeset. Art Rinn and Mark McCulloch were very helpful in explaining the Mergenthaler 202 so that we could actually get output.

While writing this text, it became clear that we would never be able to put in this volume everything we wanted. Thus there have already been mutterings of 'in the next edition ...' We would appreciate it if you, the reader, would notify us of any errors or omissions in the book. If you would like to suggest improvements or contribute exercises, we would be glad to hear from you. An errata sheet should be available to instructors in about a year.

Jim Peterson
Avi Silberschatz

Contents

Chapter 1 Introduction

1.1	What is an Operating System?	1
1.2	Early Systems	4
1.3	Simple Batch Systems	5
1.4	Sophisticated Batch	14
1.5	Time Sharing	20
1.6	Real-Time Systems	22
1.7	Multiprocessor Systems	23
1.8	Different Classes of Computers	24
1.9	Summary	26
	Exercises	27
	Bibliographic Notes	29

Chapter 2 Operating System Services

2.1	Types of Services	31
2.2	The User View	32
2.3	The Operating System View	41
2.4	Summary	46
	Bibliographic Notes	48

Chapter 3 File Systems

3.1	File Concept	49
3.2	Operations on Files	54
3.3	Directory Systems	59
3.4	File Protection	72
3.5	Allocation Methods	74
3.6	Implementation Issues	83
3.7	Summary	85
	Exercises	86
	Bibliographic Notes	89

Chapter 4 CPU Scheduling

4.1	Review of Multiprogramming Concepts	91
4.2	Scheduling Concepts	93
4.3	Scheduling Algorithms	103
4.4	Algorithm Evaluation	117
4.5	Multiple Processor Scheduling	123
4.6	Summary	124
	Exercises	125
	Bibliographic Notes	129

Chapter 5 Memory Management

5.1	Preliminaries	131
5.2	Bare Machine	133
5.3	Resident Monitor	133
5.4	Swapping	139
5.5	Fixed Partitions	144
5.6	Variable Partitions	151
5.7	Paging	158
5.8	Segmentation	170
5.9	Combined Systems	178
5.10	Summary	181
	Exercises	183
	Bibliographic Notes	187

Chapter 6 Virtual Memory

6.1	Overlays	189
6.2	Demand Paging	193
6.3	Performance of Demand Paging	198
6.4	Virtual Memory Concepts	200
6.5	Page Replacement Algorithms	205
6.6	Allocation Algorithms	215
6.7	Other Considerations	225
6.8	Summary	232
	Exercises	233
	Bibliographic Notes	241

Chapter 7 Disk and Drum Scheduling

7.1 Physical Characteristics 243
7.2 First-Come-First-Served Scheduling 247
7.3 Shortest-Seek-Time-First 248
7.4 SCAN 249
7.5 Selecting a Disk Scheduling Algorithm 251
7.6 Sector Queueing 252
7.7 Summary 254
Exercises 254
Bibliographic Notes 256

Chapter 8 Deadlocks

8.1 The Deadlock Problem 257
8.2 Deadlock Characterization 261
8.3 Deadlock Prevention 266
8.4 Deadlock Avoidance 269
8.5 Deadlock Detection 274
8.6 Recovery from Deadlock 277
8.7 Combined Approach to Deadlock Handling 280
8.8 Summary 281
Exercises 282
Bibliographic Notes 286

Chapter 9 Concurrent Processes

9.1 Precedence Graph 287
9.2 Specification 290
9.3 Review of Process Concept 297
9.4 Hierarchy of Processes 300
9.5 The Critical Section Problem 303
9.6 Semaphores 319
9.7 Classical Process Coordination Problems 323
9.8 Interprocess Communication 329
9.9 Summary 339
Exercises 340
Bibliographic Notes 346

Chapter 10 Concurrent Programming

10.1	Motivation	349
10.2	Modularization	350
10.3	Synchronization	355
10.4	Concurrent Languages	373
10.5	Summary	380
	Exercises	381
	Bibliographic Notes	384

Chapter 11 Protection

11.1	Goals of Protection	387
11.2	Mechanisms and Policies	388
11.3	Domain of Protection	389
11.4	Access Matrix	390
11.5	Implementation of Access Matrix	391
11.6	Dynamic Protection Structures	395
11.7	Revocation	400
11.8	Existing Systems	402
11.9	Language-Based Protection	407
11.10	Protection Problems	412
11.11	Security	414
11.12	Summary	416
	Exercises	416
	Bibliographic Notes	418

Chapter 12 Design Principles

12.1	Goals	421
12.2	Mechanisms and Policies	422
12.3	Layered Approach	422
12.4	Virtual Machines	426
12.5	Multiprocessors	428
12.6	Implementation	430
12.7	System Generation	431
12.8	Summary	433
	Exercises	433
	Bibliographic Notes	435

Chapter 13 Distributed Systems

13.1	Motivation	437
13.2	Topology	439
13.3	Communication	444
13.4	System Type	451
13.5	File Systems	454
13.6	Mode of Computation	456
13.7	Event Ordering	458
13.8	Synchronization	461
13.9	Deadlock Handling	465
13.10	Robustness	470
13.11	Summary	477
	Exercises	478
	Bibliographic Notes	479

Chapter 14 Historical Perspective

14.1	Atlas	483
14.2	XDS-940	484
14.3	THE	485
14.4	RC 4000	486
14.5	CTSS	487
14.6	Multics	488
14.7	OS/360	488
14.8	Unix	490
14.9	Other Systems	491

Bibliography 493

Index 527

1

Introduction

An *operating system* is a program which acts as an interface between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user may execute programs. The primary goal of an operating system is thus to make the computer system *convenient* to use. A secondary goal is to use the computer hardware in an *efficient* way.

To understand what operating systems are, it is necessary to understand how they have developed. In this chapter, we trace the development of operating systems from the first hands-on systems to current multiprogrammed and time-shared systems. As we move through the various stages, you will see how the components of operating systems evolved as the natural solution to problems in early computer systems. Understanding the reasons for operating system developments will give you an appreciation for what an operating system does and how it does it.

1.1 What is an Operating System?

An important part of almost every computer system is the operating system. A computer system can be roughly divided into 4 components (Figure 1.1):

- **hardware,**
- **operating system,**
- **applications programs,**
- **users.**

The hardware provides the basic computing resources. The applications programs define the utilization of these resources to solve the computing problems of the users. The operating system controls and

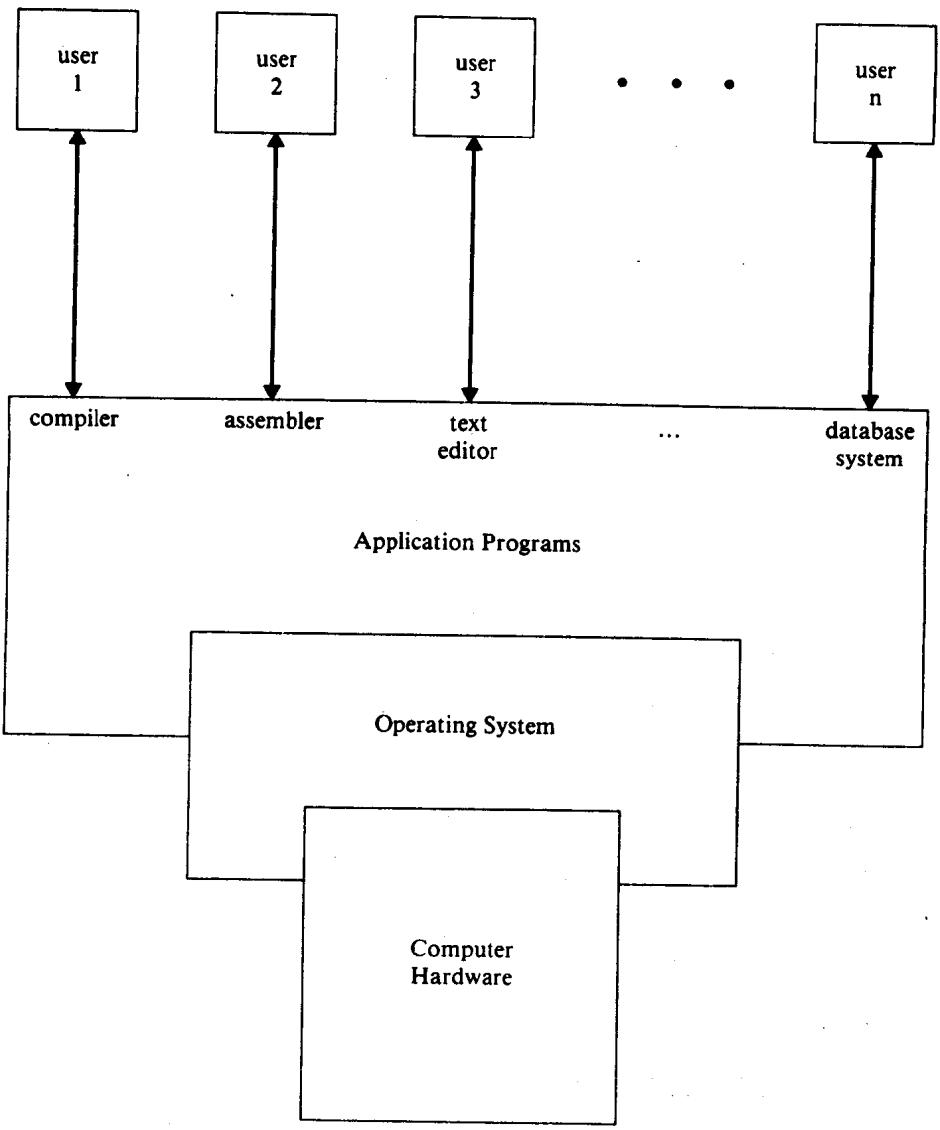

Figure 1.1 Abstract view of the components of a computer system

coordinates the use of the hardware among the various application programs for the users.

An operating system is similar to a *government.* The basic resources of a computer system are provided by its hardware, software, and data. The operating system provides the means for the proper use of these resources in the operation of the computer system. Like a government, the operating system performs no useful function by itself. It simply provides an *environment* within which other programs can do useful work.

We can view an operating system as a *resource allocator.* A computer system has many resources which may be required to solve a problem: cpu time, memory space, file storage, input/output (I/O) devices, and so on. The operating system acts as the manager of these resources and allocates them to specific programs and users as necessary for their tasks. Since there may be many, possibly conflicting, requests for resources, the operating system must decide which requests are allocated resources to operate the computer system fairly and efficiently.

A slightly different view of an operating system focuses on the need to control the various I/O devices and user programs. An operating system is a *control program.* (At least two operating systems incorporate this view into their names. CP/67 is a Control Program for the IBM 360/67; CP/M, the popular microcomputer operating system, is a Control Program for Microcomputers.) A control program controls the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

In general, however, there is no completely adequate definition of an operating system. Operating systems exist because they are a reasonable way to partition into smaller pieces the problem of creating a usable computing system. The fundamental goal of computer systems is to execute user programs and solve user problems. Towards this goal computer hardware is constructed. Since bare hardware alone is not very easy to use, applications programs are developed. These various different programs require certain common operations, such as controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

It is perhaps easier to define operating systems by what they *do,* rather than what they *are.* The primary goal of an operating system is *convenience of the user.* Operating systems exist because they are supposed to make it easier to compute with an operating system than without an operating system. This is particularly clear when you look at small operating systems for personal computers..

A secondary goal is *efficient* operation of the computer system. This goal is particularly important for large shared multi-user systems. These systems are typically very expensive, and so it is desirable to make them as efficient as possible. These two goals, convenience and efficiency, are sometimes contradictory. In the past, efficiency considerations often were considered to be more important. Thus, much of operating system theory concentrates on optimal use of computing resources.

To see what operating systems are and what operating systems do, let us consider how they have developed over the last thirty years. By tracing that evolution we can identify the common elements of an operating system and see how they developed.

Operating systems and computer architecture have had a great deal of influence on each other. To facilitate the use of the hardware, operating systems were developed. As operating systems were designed and used, it became obvious that changes in the design of the hardware could simplify the operating system. In this short historical review, notice how the introduction of new hardware features is the natural solution to many operating system problems.

1.2 Early Systems

Initially, there was only computer hardware. Early computers were (physically) very large machines run from a console. The programmer would write a program and then operate the program directly from the operator's console. First, the program would be manually loaded into memory, either from paper tape, the front panel switches or cards. Then the appropriate buttons would be pushed to load the starting address and to start the execution of the program. As the program ran, the programmer/operator could monitor its execution by the display lights on the console. If errors developed, the programmer could halt the program, examine memory and register contents, and debug the program directly from the console. Output was printed, or punched onto paper tape or cards for later printing.

An important aspect of this environment was its hands-on interactive nature. The programmer himself was the operator. Most systems used a sign-up or *reservation scheme* for allocating machine time. If you wanted to use the computer, you went to the sign-up sheet, looked for the next convenient free time on the machine, and signed up for it.

There were, however, certain problems with this approach. Suppose you had signed up for an hour of computer time to run a program that you were developing. You might run into a particularly nasty bug and be unable to finish in an hour. If someone had reserved the following block of time, you would have to stop, collect what you could, and

return at a later time to continue. On the other hand, if things went real well, you might finish in 35 minutes. Since you had thought you migh need the machine longer, you had signed up for an hour, and so th machine would sit idle for 25 minutes.

As time went on, additional software and hardware were developed. Card readers, line printers, and magnetic tape became commonplace. Assemblers, loaders, and linkers were designed to ease the programming task. Libraries of common functions were created. Common functions, once written in assembly language, could then be copied into a new program without having to be written again.

The routines which performed input and output were especially important. Each new I/O device had its own characteristics, requiring careful programming. One solution to this problem was to write, once, a subroutine which knew how to drive that device, and simply have everyone use this subroutine. Such a subroutine is called a *device driver*. A device driver knows how the buffers, flags, control bits, and status bits should be used for a particular device. Each different type of device has its own driver.

Later, compilers for Fortran, Cobol, and other languages appeared [Rosen 1967], making the programming task much easier, but the operation of the computer more complex. To prepare a Fortran program for execution, for example, the programmer would first need to load the Fortran compiler into the computer. The compiler was normally kept on magnetic tape, so the proper tape would need to be mounted on a tape drive. The program would be read through the card reader and written onto another tape. The Fortran compiler produced assembly language output, which then needed to be assembled. This required mounting another tape with the assembler. Finally, the binary object form of the program would be ready to execute. It could be loaded into memory and debugged from the console, as before.

Notice that there could be a significant amount of *setup time* involved in the running of a job. Each job consisted of many separate steps: loading the Fortran compiler tape, running the compiler, unloading the compiler tape, loading the assembler tape, running the assembler, unloading the assembler tape, loading the object program, and running the object program. If an error occurs at any step, you might have to start over at the beginning. Each job step might involve the loading and unloading of magnetic tapes, paper tapes, and cards.

1.3 Simple Batch Systems

The job setup time was a real problem. During the time that tapes were being mounted or the programmer was operating the console, the cpu