CENTRALIZED AND DISTRIBUTED OPERATING SYSTEMS

CENTRALIZED AND DISTRIBUTED OPERATING SYSTEMS

Gary J. Nutt

University of Colorado

PRENTICE HALL, *Englewood Cliffs, New Jersey 07632*

```
        Library of Congress Cataloging-in-Publication Data

Nutt, Gary J.
    Centralized and distributed operating systems / Gary J. Nutt.
       p.   cm.
    Includes bibliographical references (p.       ) and index.
    ISBN 0-13-122326-7
    1. Operating systems (Computers)    I. Title.
  QA76.76.O63N88  1992
  005.4'3--dc20                                          91-9197
                                                           CIP
```

Editorial/production supervision: Bayani Mendoza de Leon
Cover design: Wanda Lubelska
Manufacturing buyers: Linda Behrens/Dave Dickey

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-122326-7

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*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

PREFACE

Computing systems have experienced radical change during the last decade. Facilities have changed from predominantly time-sharing systems to networks of workstations and servers. While most of the principles of time-sharing and multiprogramming operating systems are still applicable to these new systems, there are new principles and design issues with which the contemporary operating system designer and software professional must be familiar to be effective in this new network environment.

This book describes principles that apply to both centralized operating systems for time-sharing and batch systems and to network and distributed operating systems. It then extends the study to encompass issues that are specific to operating systems that control networks of computers.

The reader is assumed to have used computers and operating systems in the past, possibly in an undergraduate course or in a professional position; however, we do not assume any previous operating system course as a prerequisite. Without the applied experience *using* systems, many of the ideas and issues addressed in the book will have little meaning. Experience with assembly language and computer architecture is also useful for understanding many of the concepts in the book.

The level of presentation is suitable for an a one-semester, introductory operating systems class in the first year of graduate study. We have used the material in this book in a first-year graduate course in operating systems at the University of Colorado. The course is offered in both the Computer Science and Electrical and Computer Engineering Departments and is suitable for first-year graduate students in both departments. The book can also be used as an advanced undergraduate textbook by omitting some of the advanced material, particularly in Chapters 10 through 12, and by spreading the material over a full year sequence.

The aforementioned graduate course has provided the stimulation for the book. While we have used other operating systems books in the course, we found that they either lacked sufficient technical depth for graduate-level work or did not address the breadth of topics that we found useful for a course of this type. This book represents our approach to the introductory graduate study of operating systems. It contains the basic content for the course, yet it still makes wide references to the operating systems literature. The first-year graduate student needs to be encouraged to use the material in the book, and to become familiar with the literature. To emphasize this view, we include the following problem with each assignment in the course:

> Read and report on an operating systems paper from *ACM Computing Surveys*, *Communications of the ACM*, *Journal of the ACM*, *ACM Transactions on Computer Systems*, *ACM Transactions on Programming Languages and Systems*, *IEEE Computer*, *IEEE Software*, *IEEE Transactions on Computers*, *IEEE Transactions on Software Engineering*, *AT&T Bell Labs Technical Journal*, *IBM Systems Journal*, ACM SIGOPS proceedings, ACM SIGMETRICS proceedings, or any paper in the bibliography. Your report should be about one page long; three-

quarters of the report should summarize the paper, and one-quarter of the report should be a critique. The report must be typewritten or typeset using **troff, nroff,** or any other word/document processor. Introduce your report with a formal citation of the paper in the following form:

> E. W. Dijkstra, "The Structure of THE Multiprogramming System," *Communications of the ACM*, Vol. 11, No. 5 (May, 1968), pp. 341-346.

The fundamental topics covered in this book are the essential elements of a centralized or a distributed operating system. Chapters 1 through 9 and 14 address issues related to sequential processes, scheduling, synchronization, deadlock, memory management, protection, device management, file systems, and fundamental performance evaluation. In addition, Chapters 10 through 12 covers other issues related exclusively to distributed operating systems: Networks, distributed storage, and distributed synchronization. Chapter 13 provides a broad discussion of operating system architectural designs, with specific discussions of traditional and contemporary centralized, networked, and distributed operating systems.

Many people have contributed to this book, either directly or indirectly. First, I must thank the students in Computer Science 5573 at the University of Colorado; the material was used in lectures for five years and as written drafts of the manuscript for two years. Tom Baring, David Goldstein, and Alan Youngblood were especially helpful in correcting a number of typographical errors in the late drafts of the manuscript. Bruce Sanders helped with typographical errors and with clarifying obscure discussions. Next, several reviews were obtained by Prentice Hall, including reviews by Fadhi Deek and Frank Gergelyi; they provided invaluable feedback concerning typographical errors, parts of the text that were not as clear as they might be, organization, and topic coverage.

The basic material that appears in this book comes from many sources, including the literature, colleagues, and several years of experience in industry. The early formation of the material was inspired by an operating system class offered by Alan Shaw at the University of Washington, and by research with Jerre Noe at the University of Washington and later with Skip Ellis at the University of Colorado and at Xerox PARC.

Gary Nutt
Boulder, Colorado

CONTENTS

1

INTRODUCTION

Operating systems are an important part of today's computer technology. A modern computer system typically uses a large fraction of its resources, especially primary memory, to support the operating system. The operating system is often viewed as an obstacle to the effective use of hardware, both by programming professionals and the uninitiated user. The software industry invests enormous sums of money and time into the development of operating systems. Why do we tolerate such mechanisms? What do they provide in return?

An operating system is the software that directly controls the hardware. It is intended to provide a useful and effective software environment for application programs and end users, that is, the operating system provides a usable interface between application software and the hardware. Such an interface allows a reasonable degree of independence from the hardware so that a **print** statement will write formatted information to a device without the application programmer having to know any of the details of the device's operation. In fact, the same **print** statement can be used with several different devices without changing the program in which it appears.

A computer is often shared among a set of different programs and/or a set of different users. The operating system ensures that the sharing is safe and equitable among the programs or users. It also provides boundaries between the programs or users so that errors in one program do not affect the operation of other programs.

This book is about principles of contemporary operating systems for network and distributed computer systems; much of the discussion also applies to traditional centralized operating systems. It describes the issues that arise in designing an operating system, as well as different approaches that have been used to analyze and resolve the issues.

One may choose to study operating systems for different reasons: It may be that the student wants to understand how to make better use of them. The computer science scholar may want to understand how they behave. The aspiring systems designer will study them to understand how to design better computer systems in the future.

Operating systems have been designed under a variety of different constraints and circumstances. Often, design decisions are reflected in the system's user interface as discontinuities, anomalies, or other logical inconsistencies. The user can make better use of an operating system if he† can understand the rationale behind some of these inconsistencies; it may be the case that a perceived inconsistency only points out a flaw in one's own model of how the system operates. The professional software engineer will understand how to write better software for operating systems environments if he understands some of the design issues, trade-offs, and design decisions with which the operating systems designer was faced when he designed the system.

Operating systems are traditionally one of the largest "programs" that any machine supports. They are also difficult to describe functionally, that is, it is difficult to infer the reaction of the program from its inputs. Software engineering techniques and methodologies were inspired by the need to understand how operating systems behaved in order to build operating systems that displayed the kind of behavior that was expected from them.

Some of the readers of this book will eventually design and build an operating system, or perhaps they have built one in the past. This book is intended to identify and describe the important issues in the design of centralized and distributed operating systems in order to understand the issues in organizing an operating system, to consider different techniques for resolving the issues, and to serve as a reference for the knowledge gained in the past.

The topics covered in this book are the essential elements of any operating system, including sequential processes, scheduling, process synchronization, device management, file systems, memory management, and protection and security. Chapters 10 through 12 are devoted to topics that apply primarily to distributed operating systems. The last two chapters discuss some of the conventional means by which operating systems are implemented and also introduce performance prediction methodologies.

1.1 VIEWS OF OPERATING SYSTEMS

There are two popular views of an operating system. First, one can think of an operating system as being a *resource manager* for a collection of resources. Second, the operating system can be viewed as one layer in a set of *layered abstract machines*.

1.1.1 The Resource Manager View

Computer hardware is made up of CPUs, memory, and peripheral devices. During the 1950s and 1960s, computer hardware was capital intensive. For a single programmer to use the machine it was analogous to allocating a large vehicle to transport a single person. To aggravate the situation, even when the programmer was allocated the machine, he often did not make effective use of it or was only able to use small parts of the machine at a time.

† Throughout this book, the author uses the masculine gender when referring to an individual person; all such masculine references are to a hypothetical person that may be either male or female.

To make more effective use of the hardware, the idea of sharing the machine among two or more programmers came into being. This could be done as long as one programmer was not using the same facilities as another programmer (at least at the same time). Thus, it was only necessary to introduce some mechanism that could ensure that two or more programmers were not attempting to use the same part of the machine at one time.

Any particular part of the machine can be thought of as a *resource* of the machine. For example, a block of primary memory is a resource; a tape or disk drive is a resource; and the CPU is a resource. Thus, for two or more programmers to share a single machine, some mechanism had to be derived to exclusively allocate control of parts of the machine to each programmer. A programmer should be able to request required resources from the mechanism, and then proceed when those parts had been allocated, safe in the knowledge that another programmer would not disturb the allocated resources.

An operating system fulfills this task. All the computer hardware resources belong to the operating system. Whenever a programmer — or other user of the computer system — wishes to use some of the resources, then he must request that the operating system allocate such resources. The programmer is expected to release control over the resources when he is through with them.

Thus it is natural to think of the operating system as a mechanism that manages the system resources among a group of clients.

1.1.2 The Abstract Machine View

The abstract machine view of operating systems has grown in popularity over the last several years. It is a more comprehensive model of operating systems than the resource manager model.

An architecture for a system has been defined to be a description of the interface to the system [4]. An architecture for a building is a description of how the building will be perceived by its occupants. Similarly, an architecture for hardware is a description of the how the components of the system interact with one another as viewed by the user of the system.

Since an architectural description of computer operation may be very complex, it is common to divide the architecture into manageable parts so that one can concentrate on one part of the architecture at a time. A *vertical division* of the architecture compartmentalizes parts of the architecture on the basis of functions (as perceived by the user of the system) and how they are implemented. A *horizontal division* of the architecture compartmentalizes the architecture into complete interfaces while ignoring implementations.

For example, if an office information system were to be designed using vertical division, then one part of the system might be electronic mail and its implementation, while another part of the system would be a calendar system and its implementation. The vertical division would discourage the use of common parts between the electronic mail subsystem and the calendar subsystem. Information might be stored entirely differently in the two subsystems, even though both would require the use of some type of file system. However, it would be easy to delegate the design of such a system, since the electronic mail system would be assigned to one group while the calendar subsystem was assigned to another. Development could proceed in parallel, with little worry about cooperation among the groups.

If the same office system were to be designed using horizontal division, then one part might be the user interface, a second part the file system, and so on. In this case, the calendar subsystem and the electronic mail subsystem would employ the same user interface and the same file system. This would reduce the number of redundant parts in the system, but would tend to serialize production unless some strong agreement were made between the group implementing the user interface and the groups implementing the specifics of the calendar or the electronic mail packages.

Abstract machines embrace the horizontal division of the architecture. The architecture is described as a family of n *abstract machines*, $\{A_i \mid 0 \leq i \leq n\text{-}1\}$, where A_i is built on top of the interface to A_{i-1} (see Figure 1.1). The interface to A_i is a *description* of the behavior of a mechanism that can be used by an implementer of A_{i+1}. The user of A_i need not know the details of the implementation of A_i, only its interface.

The bottom layer machine, A_0, is usually thought of as the "hardware," but of course the same layering technique can be applied to the design of the hardware itself. In this case, A_0 might be construed as the logic level of the hardware, while A_1 is a machine implemented by microcoding a control unit. Therefore, we ignore the actual definition of the base substrate and rely only on a sound interface to the hardware, A_H.

Assume that level A_H is implemented by the hardware of the machine, that is, the mechanism that implements A_{H+1} is software. Then A_H can be described in terms of machine instructions, registers, buses, controllers, interrupts, and memory.

Assembly language-level programmers work with A_H as a regular practice. At this level, many details must be kept in mind in order to successfully control the hardware. For example, the A_{H+1} machine programmer must always distinguish between floating point and integer addition (instead of simply using "+" to denote summation of two variables of the same type). In fact, variable typing itself is no more than a convention at the A_{H+1} level.

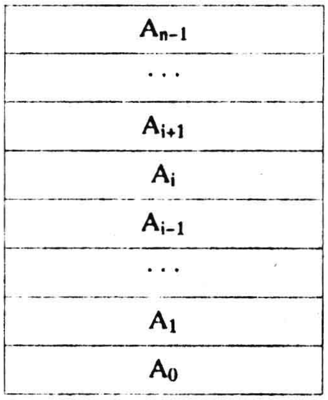

Figure 1.1 Layered Abstract Machine Division

What should be the purpose of the mechanism that implements A_{H+1}? It could, of course, provide service directly to the end users of the computer system. That is, A_H could be used as the basis for constructing all application programs. However, we have pointed out the need for resource management in systems that support multiple users or programs. It is reasonable to think of implementing resource management at level A_{H+1} and then implementing application programs at level A_{H+2}. Recalling the office information system example, it is easy to see that there is good reason to make room for other abstract machines between the resource manager and applications programs, for example, a file system. As a result, the abstract machine model for operating systems begins to take on the appearance shown in Figure 1.2.

Although the layered abstract machine model is widely used for designing computer systems in general and operating systems in particular, there is limited agreement about the details of the operating system interface, A_S, and even less agreement about the layers between A_H and the user interface, A_U. In Chapter 13, we will return to a discussion of abstract machine layering approaches that have been used between A_H and A_S.

1.1.3 Commercial Examples

Operating systems have evolved from a laboratory curiosity into an important part of commercial products. Computer manufacturers are generally unable to market hardware without including an operating system. Because operating systems provide an abstract machine interface to the application software, the *portability* of the application software depends on the nature of A_S — the operating system interface. (In some markets, the important interface is A_H instead of A_S, that is, portability is at the *object code level* as opposed to the operating system call level.)

Some manufacturers have taken the position that this is a strategic position to maintain: Once a customer has purchased application software that is not portable to another manufacturer's operating system interface, then the customer is captured. (Of course, this may also work against the manufacturer, since it prevents him from changing his own hardware and operating system to take advantage of technology changes.)

Because of this commercial influence, a few operating systems have become very visible over the years. We mention them here because of that importance and because they have either pushed the state of the art or popularized many different notions in the

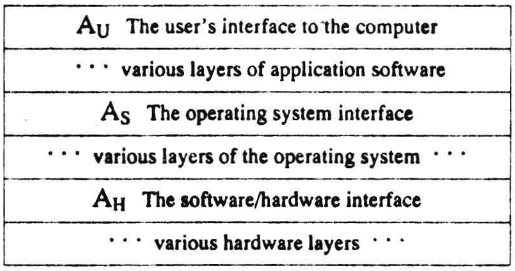
A_U The user's interface to the computer
· · · various layers of application software
A_S The operating system interface
· · · various layers of the operating system · · ·
A_H The software/hardware interface
· · · various hardware layers · · ·

Figure 1.2 The Operating System Abstract Machine

field. Because of their popularity, many readers of this book will have encountered some of these operating systems; thus we will use them to illustrate various concepts throughout the remainder of the book.

IBM OS/360

In the early 1960s, IBM introduced a revolutionary new machine called the *System/360*. The 360 was intended to accommodate customers who intended to upgrade from older IBM equipment and to address the "scientific" and "commercial" markets with a single system. The System/360 was marketed as a family of machines, each with the same functional A_H. This was accomplished through extensive use of microcode at level A_{H-1}.

System/360 was delivered with a new operating system called *OS/360*, which provided a consistent A_S across the family of computers. Thus, the family of machines provided application program portability at either the source or object code level; software compiled for a small machine could run directly on a larger machine in the line. This was the first time that such a family had been offered in a commercial marketplace.

OS/360 was also the largest piece of software that had been built until that time. It used a number of new techniques for controlling the hardware and for providing a sophisticated multiprogramming environment. Ironically, OS/360 also provided the primary motivation for the development of the software engineering discipline. Fredrick Brooks was the project leader for the development for OS/360; he wrote a series of revolutionary software engineering articles reviewing his experiences with the development of OS/360 [4].

AT&T UNIX

In 1974, Dennis Ritchie and Ken Thompson of AT&T Bell Laboratories published a paper describing a "small" time-sharing operating system for a DEC PDP 11/45 minicomputer [26]. UNIX had been influenced by Multics (an important research time-sharing operating system developed at Project MAC at M.I.T), but had taken many simplifying assumptions in its design and implementation.

UNIX became very popular among computer scientists since it was reasonably compact, it could be ported onto various A_H without great difficulty, and since it provided the most important aspects of a resource sharing operating system without an undue amount of other mechanism. UNIX was easy to extend so that one could build a customized programming environment. While UNIX was described in the technical literature in the early 1970s, AT&T was restricted from freely distributing the operating system due to certain regulatory agreements. These agreements did not apply to distributions of UNIX to universities; as a result, UNIX was widely distributed among universities and has subsequently become well-entrenched in that community.

In the late 1970s and early 1980s, variants of UNIX began to appear, largely due to the fact that the UNIX source code had been widely distributed within the academic community. Here, the most prominent version soon became the *Berkeley Software Distribution* or *BSD* from the University of California at Berkeley. In the 1990s three primary variants of the operating system exist: Research UNIX in the AT&T Bell Laboratories, AT&T UNIX now widely distributed by AT&T, and BSD UNIX.

Because of the relative ease with which UNIX can be ported among different A_H, and because of the relatively wide use of the UNIX A_S by application programs and application-support packages, various versions of UNIX are experiencing increasing commercial success. Most universities now support their educational program, at least in part, through the use of UNIX.

1.2 HISTORY OF OPERATING SYSTEMS

In the 1940s and 1950s, computers were often "personal computers" in the sense that the machine would only support a single user at a time. Operating systems were degenerate, since their primary purpose was resource management and since all resources were allocated to a single user at a time.

Once a user was allocated the machine, then he could use the peripheral devices to load a program into the memory and to execute the program. In general, a program would read data from some input device, process them, and then write a result to an output device.

I/O operations were an important part of the program functionality then, as they are today. I/O involves the management of a mechanical device under programmable control. Early systems had only a single programmable unit — the CPU, thus I/O operations required the full attention of the CPU to manage the operation of the device.

As hardware grew in sophistication, it soon became apparent that one could design hardware so that a special-purpose *channel* processor could be built to manage the mechanical input/output function, freeing up the processor to do other things. This allowed the CPU to be utilized for other processing while I/O was in progress. It also required that certain standard operations be performed on the channel in order to instruct it to perform specific I/O operations. Libraries of software to control channels soon evolved; these were the first examples of system software.

The existence of channels greatly increased the complexity of software. A program could only be made to be efficient by being constructed so that it maximized the overlap of operation between the CPU and the channel. This required that the program be written so that it started an I/O operation and then proceeded to do other processing until the channel had completed. Thus the program would have to be constructed so that it initiated the I/O operation well before the data were actually required, then it could perform independent tasks in parallel with the channel operation. Detecting the completion of the I/O operation was accomplished by periodically sampling the channel status.

1.2.1 Multiprogrammed Batch Systems

A *batch processing system* is an operating system that will service individual *jobs* from a queue. To use the batch processing system, a job is prepared (usually as a file or deck of punched cards) by specifying all the steps that are required to fulfill the processing needs and by supplying specific data that will be needed during processing. For example, a job might consist of three different parts: a control section, a program, and data (see Figure 1.3a). The control section in Figure 1.3a requests the operating system to compile the next record in the *batch stream*, the source code, then to link and load the relocatable object code with library routines, and finally to execute the resulting binary program on the data supplied in the last record of the batch stream.

In some cases, batch control directives are embedded in the batch stream rather than being in a separate control section (see Figure 1.3b); but in either case the operating

```
Control Section:
        Compile command                     Compile command
        Link & Load command
        Execute command
Program Section:
        <record separator>                  <record separator>
                   MAIN()                           MAIN()
                   READ(5,9001) A, I                READ(5,9001) A, I
                   ...                              ...
                   WRITE(6,9101) X                  WRITE(6,9101) X
        9001       FORMAT(...)              9001    FORMAT(...)
        9101       FORMAT(...)              9101    FORMAT(...)
                   <record separator>
Data Section:                               Link & Load command
                                            Execute command
        <record separator>                  <record separator>
        123.45     67                       123.4567
        234.56     78                       234.5678
        ...        ...
        567.89     12                       567.8 912

           (a)                              (b)
```

Figure 1.3 A Batch Job

system is responsible for executing some set of commands on the supplied information.

The batch system operator provides a series of jobs to the system, usually by reading decks of cards into a system queue (on magnetic tape) as shown in Figure 1.4. This process is called *job spooling*, since the operator enqueues jobs on the system queue much as one would wind thread onto a spool. The operating system removes jobs from the system queue whenever it is capable of processing another job.

Uniprogrammed batch systems are not much different from the simpler machines discussed in the previous section (except that the device that holds the job queue must be protected from user programs). Once a job has been removed from the system queue and allocated the processor and memory, it controls the entire machine (other than the system queue).

Multiprogrammed machines provide for the management of the simultaneous execution of jobs. The various components of the system are generally *serially reusable*, meaning that a unit of the resource may be allocated to a single job at a time, but as soon as one job has completed using a resource, then the resource can be reused by another job. The CPU, memory, and devices are serially reusable resources. To support multiprogramming, the operating system must manage the sharing of serially reusable resources among several jobs. That is, if two or more jobs are to share the resources, then they must share the memory of the machine (either by partitioning the memory and giving each exclusive control of a partition or by multiplexing the jobs in and out of the same section of memory).