Second Edition

OPERATING SYSTEMS PRINCIPLES

VAN NOSTRAND REINHOLD DATA PROCESSING SERIES

OPERATING SYSTEMS PRINCIPLES

SECOND EDITION

Stanley A. Kurzban
Thomas S. Heines
Anthony P. Sayers

VNR VAN NOSTRAND REINHOLD COMPANY
NEW YORK CINCINNATI TORONTO LONDON MELBOURNE

Library of Congress Catalog Card Number: 83-10274
ISBN: 0-442-25734-1

Manufactured in the United States of America

Published by Van Nostrand Reinhold Company Inc.
135 West 50th Street
New York, New York 10020

Van Nostrand Reinhold Company Limited
Molly Millars Lane
Wokingham, Berkshire RG11 2PY, England

Van Nostrand Reinhold
480 Latrobe Street
Melbourne, Victoria 3000, Australia

Macmillan of Canada
Division of Gage Publishing Limited
164 Commander Boulevard
Agincourt, Ontario M1S 3C7, Canada

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging in Publication Data

Kurzban, Stanley A.
Operating systems principles.

Bibliography: p.
Includes index.
1. Operating systems (Computers) I. Heines, Thomas S.
(Thomas Samuel), 1927– II. Sayers, Anthony P.
III. Title.
QA76.6.K874 1984 001.64'2 83-10274
ISBN 0-442-25734-1

Preface to First Edition

This book is an introduction to the concepts and technology of computer operating systems and is intended for use by students taking a first course in operating systems theory. Typical of the courses for which this book would be appropriate are the "undergraduate course of operating systems principles" described by the COSINE Committee on Education of the National Academy of Engineering's Commission on Education [COS71] and the Course 14 of the ACM's "Curriculum 68" [ACM68]. This book is also of value to those in management, procuring, project planning, and the like, who may become involved in the development, modification, or use of an operating system.

It is assumed that the reader has some familiarity with the elements of computer architecture, an assembly language, and one or more higher-level languages, and has, in addition, some experience in the use of an operating system.

The material in this book should enable readers to understand the function and design of any operating system they might encounter. They should be able to participate in the design, implementation, or modification of an operating system after learning the specifics of a given project.

The style of the book is tutorial; there are many examples and exercises. Abundant references direct the reader to more detailed studies of particular subjects.

For the purposes of this book, an operating system includes supervisory and file-management routines, utility programs, and processors for the system's control languages. Compilers are discussed only insofar as their characteristics are affected by their host operating system.

This book covers the concepts of all operating systems in current use. Since the authors have the greatest familiarity with the IBM 360/370 Operating System(s), more examples are taken from these systems than from any other; however, the concepts in the book are by no means limited to those systems.

Preface to the Second Edition

A measure of the aptness of this book's title is the extent to which its topics, "Principles," apply to operating systems as they evolve. We hope the reader agrees that the past seven years give evidence of the wisdom with which our title and topics were chosen.

While principles endure, emphasis may not. Many have observed that vast fortunes depend on operating systems' attributes, particularly their reliability and security. So it is that these topics receive increased attention in the present edition. This is especially true of the latter topic, one whose importance many failed to appreciate in 1975.

Another area growing in significance is distributed processing. Accordingly, the present edition devotes more attention to the movement of data and requests for service into and out of the system. On the other hand, various developments, including enhancements to the computers on which operating systems run, have modified the roles of queuing theory and compiler design, leading us to omit these topics that properly deserve separate books of their own rather than the cursory treatment we can afford them in the present context.

References to features and facilities that conserve space in main storage or optimize placement of files on secondary storage are retained despite the fact that reduced costs per unit and improved use of multiprogramming techniques have rendered some of these obsolete in the contexts of some systems.

The successors of operating systems developed for IBM's System/360 and System/370 computers now run on many other computers as well. To avoid confusion, we continue to refer to the older families of computers.

Stan Kurzban
Tom Heines

Contents

Preface / vii
Preface to the Second Edition / ix

1. Introduction / 1

1.1 Historical Perspective / 2
1.2 Goals of Operating Systems / 5
1.3 Components of Operating Systems / 10
1.4 Operating System Design Techniques / 20
1.5 Relationships with Hardware / 27

2. Types of Operating Systems / 33

2.1 Serial Batch Processing Systems / 33
2.2 Simple Multiprogramming Systems / 37
2.3 Complex Multiprogramming / 50
2.4 Multiprocessing Systems / 53
2.5 Real-Time Systems / 60

3. Operating System Services / 67

3.1 Hardware Interfaces / 67
3.2 Input/Output Services / 81
3.3 Error Recovery / 93
3.4 Language Processors / 99
3.5 Utilities / 109
3.6 Accounting for Resource Usage / 123
3.7 Access Control / 132

4. Job and Task Management / 140

4.1 General / 140
4.2 Management of Addressed Storage (Real) / 150
4.3 Management of Adressed Storage (Virtual) / 155

4.4 Management of Named Storage / 167
4.5 System Integrity / 174
4.6 Management of Tasks / 176
4.7 Management of Jobs and Steps / 185

5. Data Management / 199

5.1 Secondary Storage Management / 199
5.2 File Organization / 206
5.3 Access Methods / 216
5.4 Data Bases / 228
5.5 Data for People / 236
5.6 Programs as Data / 244

6. Symbol Binding / 250

6.1 Mechanisms / 251
6.2 Control Language / 269
6.3 The Operator's Interface / 285

7. The Development Process / 297

7.1 Preliminaries / 297
7.2 Tools / 311
7.3 Techniques / 317
7.4 Implementation Language / 341
7.5 Measurement, Modification, and Maintenance / 348

Answers / 361
Appendix A. Operating System Glossary / 364
Appendix B. Operating System Functions / 373
Appendix C. Sample Control Block / 380
Appendix D. Data Security Considerations / 382
Bibliography / 383
Index / 409

1
Introduction

The term "operating system" came into widespread use in the late 1950s. Sayers [Say71] defines an operating system as "a set of programs and routines which guide a computer in the performance of its tasks and assist the programs (and programmers) with certain supporting functions." This definition is accurate and useful, but by no means the only one. The American National Standard definition is: "Software which controls the execution of computer programs and which may provide scheduling, debugging, input/output control, accounting, compilation, storage assignment, data management, and related services." This definition, while similar to Sayers', seems too restrictive and too dependent on other terms which themselves are jargon and require definition.

We can expand Sayers' definition by naming the supporting functions provided or the programs assisted. Alternatively, we may make reference to current practices in the computer industry by defining operating systems as "that programming which is provided by the vendor of a computing system as an integral part of the product he markets." This, of course, yields an incomplete and imprecise definition which varies with vendors and computers.

Let us, then, accept Sayers' definition of an operating system with some qualifying remarks. Some authorities would restrict the term "operating system" to a set of programs which creates an apparent computing system as comprehensive as the original hardware system, but simpler and easier to use. For our purposes, however, those programs which are peripheral to the computing system, but vital to the effective use of the system and of potential value to all those who use the system, are included. Examples of such programs are utilities (for example, file-copying programs, sorts, and listing programs), spooling[1] programs, and routines for managing networks of computers. Language processing programs – assemblers, compilers, and interpreters – are considered here only insofar as they are influenced by the characteristics of particular operating systems.

Having defined our subject, we next consider how operating systems have evolved to their current level of sophistication. We then discuss the purposes

[1] Spooling, from *s*imultaneous *p*eripheral *o*perations *o*n-*l*ine, is the processing of data between a device designed for communication with people (a printer or card punch, for example) and an intermediate storage device (such as a disk or magnetic tape device).

served by operating systems, their constituent parts, some programming techniques used in their development, and the influence of computers upon the operating systems which support them. These topics will lay the foundation needed for more detailed study of operating systems.

1.1 HISTORICAL PERSPECTIVE

The development of operating systems has been marked by two different types of progress: evolution and retrenchment. Within generations of computers, evolution has dominated, with successive advances occurring in response to problems successively encountered. The problems often arose as a result of earlier advances, a phenomenon Marine [Mar70], in another context, attributes to "the engineering mentality," the way of thinking which insists upon solving every problem individually without regard for related problems or side effects. Such an approach, despite its seeming inadequacy, yields prompt solutions to problems, a requirement in the fast-advancing field of electronic computing. This type of development has fortunately been reinforced by retrenchment, more comprehensive advances, which consolidated earlier technological gains. These advances have most often coincided with the introduction of radically different computers, but also at times, for combinations of less than obvious reasons, occurred in the absence of hardware innovation. Radically new operating systems, not truly revolutionary, but significantly innovative, have permitted the consolidation of earlier advances and the development of integrated sets of new programs which provide totally new facilities.

Before there was an operating system, there were computers. With little or no accompanying programming, the computer was a very complex tool, difficult for even its designers to use efficiently. Systems with one-card loaders and primitive assemblers could be of use to those with the patience and analytical skill needed to break a problem down to a succession of additions, divisions, etc., provided someone well acquainted with the computer could write input/output (I/O) routines for them. But some systems developed prior to 1955 lacked even an assembler. They had to be programmed in octal or decimal, with the programmers supplying operation codes and branch locations by themselves. The addition of a single instruction near the beginning of a program meant the relocation (and repunching) of perhaps thousands of addresses, or patching – the addition of code at the end of a program and the replacement of an existing instruction by a branch to the new code. The new code had to begin with the replaced instruction, and end with a branch back to the location following the replacement. A program with many patches was not only inefficient, but also very difficult to debug.

The users of these primitive systems normally operated the computer personally. The absence of both aids for debugging and established operating procedures made this necessary. No third party could have known how to interrogate and modify storage in response to an unexpected condition or a bug. No one but the programmer could distinguish a loop from normal computation or determine from patterns of lights that a new reel of magnetic tape should be mounted.

Monitors

The first problem to be solved by a systems program, that is, one which performed no work specifically requested by any user, was, naturally enough, that of accomplishing transitions between users' programs. This was a serious problem. Many jobs required more time for preparation than for execution. *Monitors* such as the one developed in the late 1950s by programmers at General Motors and North American Aviation for the IBM 704 [Ros69a] accomplished job-to-job transition and invoked other programs which facilitated use of the computer – an assembler, a loader, a FORTRAN compiler, and a dump.

Monitors soon were developed to accept and produce reels of magnetic tape in lieu of card decks and printed listings, respectively. The tapes could be processed by a less expensive (peripheral) computer used only for that purpose. This relieved the principal (central) computer of the burden of dealing with slower devices. Jobs to be run under the monitors were restricted as to what they could do, making job set-up so simple that it was possible for operators to replace the programmers at the computer. Procedures could be varied, but the variations permitted were well-defined and within the operators' capabilities.

With the appropriation by the monitor of magnetic tape drives for specialized use, restrictions had to be placed upon programmers' use of I/O devices. Note that this is a typical example of a problem caused by a solution. This problem was a fortunate one, however, inasmuch as it led to the development of I/O subroutines. These greatly simplified programmers' views of I/O operations. Since these routines were used very frequently, they were stored in *libraries* on magnetic tape as object (executable or binary), rather than source (character string), code. This led in turn to the more extensive use of libraries of programs in this more efficient format.

Since such programs had to be used by almost all of the system's programmers, it became necessary to permit their loading at various locations in main storage to meet the constraints of many varied main programs. The relocatable *loader* could modify the routines as they were loaded from libraries, to reflect the locations selected for them.

Other innovations which appeared in the late 1950s included *directories* for program libraries to facilitate sequential searching, *overlay* mechanisms to permit the serial use of given storage locations by a sequence of routines, and a system *log* wherein accounting data could be recorded by the monitor.

Executives

The monitor of the late 1950s became an *executive* in the early 1960s. (This term had also been used earlier.) The executive was designed to facilitate efficient use of the channels and interruption mechanisms which were then becoming available. These permitted computational processing to occur simultaneously with I/O operations. The completion of an I/O operation caused an interruption of processing. Processing could be resumed after the successful completion of the I/O operation had been noted somewhere or something had been done about an unsuccessful operation. The complexity involved was best dealt with by a single group of programmers whose executive could then make other programmers' jobs easier.

Supervisors

As executives became *supervisors,* they performed more functions of general necessity, but troublesome complexity. They also became permanently resident in main storage, giving the term "system overhead" a spatial dimension as well as a temporal one. Most of the supervisor's routines were useful to most users of the system, but *all* users had to tolerate the presence of *all* the routines in main storage at *all* times.

With the problem of usability solved, the problem of integrity appeared; that is, since almost any reasonably intelligent person could now program a computer, many who might better not have, did. And when they did, they effected unintentional modifications of the supervisor, rendering it inoperable. So it was that protection mechanisms, such as registers for bounding addessing capability and locks for restricting access to sensitive data, were developed.

The IBM 7040/7090 family of computers was a focus for much of the evolutionary and nonevolutionary work of the early 1960s. IBM produced IBSYS [IBM6] with its IBJOB monitor and IOCS (Input/Output Control System) in 1963. In 1962-1965 Project MAC at MIT produced the Compatible Time-Sharing System (CTSS, language-compatible with FMS, the FORTRAN Monitor System) [Cri65], a system which *swapped* users' programs between main and secondary storage at intervals called *time slices.* Significant work was also done at the University of Michigan [Mic63], Yale [Yal63], and IBM's research laboratory at Yorktown Heights, New York, among other places. These systems consolidated earlier advances and opened new avenues of progress. All of the systems, but most notably IBSYS, went through significant periods of evolution, but nevertheless represented milestones in the development of operating systems at their inception.

Operating Systems

The IBM System/360 Operating System (OS/360) [Mea66], which supported the System/360 series of computers, ushered in the present era of operating systems.

While many facilities of existing operating systems (checkpoint/restart, time-sharing, and even simple multiprogramming) were lacking in OS/360 when it was first released in 1965, that system was truly a new species [Ros67]. It is well represented in the material which follows, and is therefore not described further at this point. Still more recent landmark systems are MULTICS (*mul*tiplexed *i*nformation and *c*omputing *s*ervice) [Cor65] and IBM's TSS/360 and OS/VS [IBM7]. Descriptions of these are also deferred.

The history of operating systems is not exclusively American. The ATLAS system [Kil67], developed in England by Ferranti, Limited, and Manchester University, seems to have been "years ahead of its time." The GEORGE systems [Cut70], developed by International Computers, Limited, (ICL), are also noteworthy and the THE System [Dij68b], developed by Dijkstra at the University of Eindhoven, the Netherlands, has contributed as much as any other to the theory of systems programming.

An excellent survey of the history of operating systems [Ros69a] is the source for much of the foregoing. The first two papers of [Rose67] cover the same ground and related subjects. See also [Kat73]. More detailed views of individual systems are best gleaned from the references cited.

1.2 GOALS OF OPERATING SYSTEMS

Operating systems are intended to facilitate efficient use of computers. They provide a convenient interface to hide from programmers the complexity of the bare computing systems. They manage the resources of computing systems so that the resources are optimally used. They permit the accounting for individuals' use of resources. They make it possible for programs to be impervious to minor malfunctions or the unavailability of one out of many similar resources, for example, one magnetic tape drive out of ten that might be attached to a computer. They protect users' programs and data from accidental or malicious destruction, theft, or unauthorized disclosure to other users. To do this effectively, they provide the same protection for themselves. And they can even give computing systems the appearance of being much larger than they really are, via so-called "virtual" resources.

Some of the terms we use, because they have special meanings in our context, require definitions. These are supplied in the following list of attributes of operating systems:

1. Usability – the property of being easy to use; appearing to have been designed for the user's convenience.
2. Generality – the property of being useful in many ways; the system does all and only what the set of all of its users want it to do.
3. Efficiency – the property of functioning quickly; the system makes optimum use of the resources at its disposal.

4. Visibility – the property of revealing to users all they must know to take maximum advantage of the system.
5. Flexibility – adaptability to a specific environment; the system's behavior can be suited to its tasks.
6. Opacity (commonly called "transparency" in defiance of that word's normal meaning) – the property of allowing users to remain unaware of all details they need not know, all that lies beneath the interface provided by the system.
7. Security – the property of protecting data from unauthorized access, whether malicious or accidental.
8. Integrity – the property of protecting itself and users from damage or any other ill effect of others' errors or malice.
9. Capacity – the property of lacking unnecessary limitations.
10. Reliability – the property of appearing to fail as rarely as possible.
11. Availability – the property of providing as much function as much of the time as possible.
12. Serviceability (or maintainability) – the property of being easily and quickly repaired.
13. Extensibility – the property of accepting additions and modifications with maximum ease.

The definitions of the attributes in our list contain few terms that restrict their applicability to operating systems. The designers of any product might do well to consult the list in doing their work.

All of these characteristics contribute to the broader goals of an operating system: to permit people to accomplish meaningful tasks more easily and less expensively than would otherwise be possible. The attributes we have listed may be seen as subgoals which contribute in more or less obvious ways to the principal objectives of ease of use and efficiency, in the more general sense of the latter term.

The most obvious accomplishment of an operating system is its presentation to the user of an interface much easier to use than that of the computing system itself. This is not so much true with respect to computational processing, where the assembler stands between the user and the computer. But it is true with respect to the peripheral, yet vital, processing incidental to computation.

Before data can be processed, it must be communicated. The operating system permits programmers to code simple READ and WRITE statements, although the computer itself requires much more precise direction. To process data efficiently, one must manage resources. The operating system needs only statements concerning requirements for types of resources. The selection of specific resources is handled by the system. In general, the operating system permits users to express their requirements in terms meaningful to them, instead of in the language specific to some conglomeration of circuits and registers.

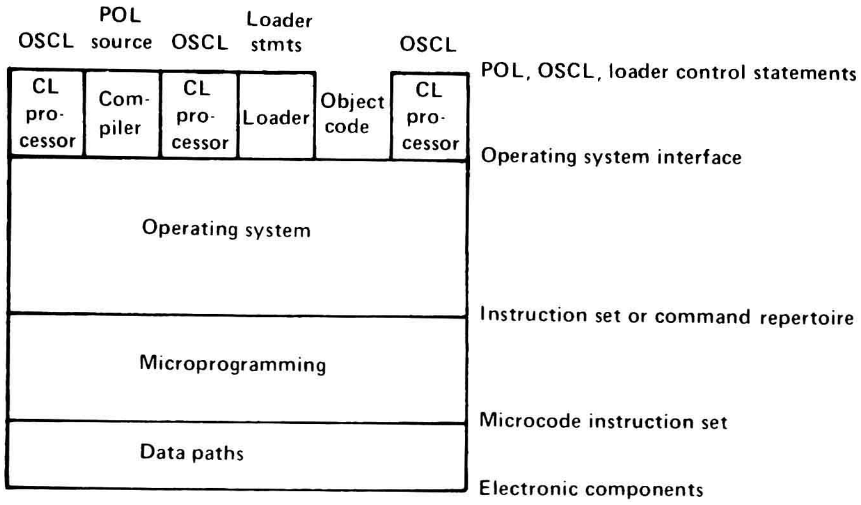

Figure 1.1. Levels of Interface for a Compile/Load/Go Job.

This simplification may be seen as an interface of a level higher than that of the computer itself. A higher-level language such as COBOL, FORTRAN, or PL/1 can be perceived as a still higher level of interface, one provided by a compiler. The compiler, in turn, is written to the interface of the operating system, the same interface at which the compiler's output, the object program, executes. This notion of levels of interface is graphically illustrated in Figure 1.1, where the operating system's control language (OSCL) is seen as another language. In the case of the OSCL, the language used for communicating with the operating system, the language processor is probably not properly called a compiler. But the analogy has substantial validity.

The facilities offered by an operating system are dictated by the generality of their usefulness. All useful programs must obtain data and produce data. I/O routines are therefore of general usefulness. For this reason, they are included in operating systems. Languages intended for use in scientific computation are likely to include a square-root function because it is of general use to scientists. But many operating systems are used extensively for nonscientific processing, and so do not have square-root facilities. Such a facility would not be of sufficient use to a sufficient portion of the using population to justify the cost of its implementation. Operating systems are not immune to economic principles: a product's value must exceed the cost of its production. This defines the bounds of generality.

An operating system is designed to serve not only its individual users, but also, more importantly, the totality of all its users as a group. One way it accomplishes the latter objective is by coordinating the users' utilization of the system's resources. The resources of a computing system are many: main storage, one or more central processing units (CPUs), I/O devices, channels, and secondary storage media. Whenever a CPU is waiting for work or main storage space is unused, a resource is being wasted. An operating system tries to process users' jobs in such combinations and sequences as to maximize the use of resources. By spooling and other means, the system makes use of unused resources to perform operations before they are absolutely required. When the operations are required, they may appear to occur instantaneously or, at least, much more quickly than would otherwise be the case. Thus less time is lost waiting for the required operation. This is an example of efficiency.

In dealing with the system's resources, the operating system performs a service of another type. It collects statistics concerning the use of the resources by particular users. This permits the manager of an installation to charge the users of the system on the basis of their use of it. This serves the community of users by discouraging users from using more than they need, thus making more available to all the users as a group. These data on use also permit tuning of the system, modification of parameters of the system, and even its configuration, to tailor the system to the requirements of its users. The provision of these data is visibility. The consequent tuning illustrates flexibility.

In providing a high level of interface, the operating system shields its users from more than just the computer's complexity. They are also protected from its variability. Since they may ask for *a* tape drive rather than *that* tape drive, they need not be aware that *that* tape drive is unavailable due to preventive maintenance. In fact, the tape drive they are given to use *may* be superior (for example, with respect to the density of the data on the recording medium) to the one they were expecting. It may even be a disk drive! The point is that by creating a new interface, the operating system can continue to function without apparent change, even if the interface to which it is written changes substantially. Just as a higher-level language may be machine-independent, so an operating system may be independent of many of the characteristics of the computing system beneath it. This is opacity. (It is in the sense that users "see" the tape drive "through" the system that some mistake opacity for "transparency.")

Operating systems exploit inventions that make it possible to restrict access to certain facilities and storage locations. An operating system may require that a user know a certain password or be authorized in some prescribed manner before he is granted certain privileges. These privileges might include the right to execute I/O instructions directly rather than through the system's I/O routines, the right to modify or destroy certain data files, or the right to read from certain main or secondary storage locations. Thus, the system can protect data and programs from unauthorized use or destruction. This applies not only to malicious misuse, where users try to circumvent the system's regulations for their