THE LOGICAL DESIGN OF OPERATING SYSTEMS

ALAN C. SHAW

THE LOGICAL DESIGN OF OPERATING SYSTEMS

ALAN C. SHAW

Computer Science Group
University of Washington

PRENTICE-HALL, INC.

ENGLEWOOD CLIFFS, N. J.

Library of Congress Cataloging in Publication Data
SHAW, ALAN C

The logical design of operating systems.

(Prentice-Hall series in automatic computation)
Bibliography: p.
1. Electronic digital computers—Programming.
2. Multiprogramming (Electronic computers) I. Title.
QA76.6.S52 001.6'42 74-537
ISBN 0-13-540112-7

10 9 8 7 6

Printed in the United States of America

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*

PREFACE

Computer operating systems are among the most complex "systems" devised by humans, and it is only recently that we have been able to understand and coherently organize this complexity. This book is a text on the *principles* of operating systems, with particular emphasis on multiprogramming. I have tried to present the concepts and techniques required for engineering and understanding these systems rather than discuss in detail how operating system *x* is implemented on machine *y*; however, many examples from real systems are given to illustrate the application of particular principles. The title "logical design" was selected to stress my concern with the logical organization and interactions of the elements of operating systems and with methods of "reasoning" about them.

The book is intended for computer science students and professionals with a basic knowledge of machine organization, assembly language, programming languages, and data structures. The prerequisite background can be obtained in an introductory one-term course in each of the above subjects, approximately equivalent to courses B2, I1, I2, and I3 of the curriculum proposed in *Curriculum 68* by the Association for Computing Machinery[1]. While the book was being written, I used it as the primary text for a one-term graduate course at Cornell University and at the University of Washington. The book is suitable for a one- or two-term course at either the graduate or advanced undergraduate level, and contains almost all the topics suggested in course I4 of *Curriculum 68*[1] and in the more recent COSINE report[2].

[1]ACM Curriculum Committee on Computer Science. *Curriculum 68*, recommendations for academic programs in computer science. *Comm. ACM*, **11**, 3 (March 1968), 151–197.

[2]COSINE Committee of the Commission on Education. An undergraduate course on operating systems principles (Denning, P. J., chairman). Commission on Education, National Academy of Engineering, Washington, D.C., 1971.

My global view is that the subject of operating systems is most conveniently divided into three related areas: process management, resource management, and file systems. Each of the nine chapters of the book is concerned with some aspects of one or more of these areas. Chapter 1 provides an overview of the organization of systems hardware and software, including a historical perspective and rationale. In Chapter 2, I use the simple setting of a job-at-a-time batch system to present some basic ideas on linking loaders and input-output methods. The model of interacting processes as a means for describing systems and as a framework for solving problems of process communication and synchronization (including some problems introduced in Chapter 2) is developed in Chapter 3. Chapter 4 is an introduction to multiprogramming systems; building upon the material developed in the preceding chapters, it discusses hardware and software requirements for multiprogramming, the "virtual" machines viewed by users and by systems programmers, and design methodologies. Techniques for the management of real and virtual memories are investigated in Chapter 5; the following chapter (Chapter 6) continues the study of the main memory resource, looking at the problems of single-copy sharing of information in real and virtual memory systems. Process and resource management ideas are consolidated in Chapter 7, where a comprehensive nucleus is described and used as a model for examining systems data structures, input-output processes, interrupt handling, and scheduling methods. Chapter 8 gives a detailed treatment of systems deadlock; methods for deadlock detection, recovery, and prevention are described for both serially-reusable (conventional) and consumable (message-like) resources. The last chapter (Chapter 9) discusses the basic elements of file systems, including a section on recovery from failure.

The book contains many exercises which the reader is strongly encouraged to do. In learning new ideas on computer systems, it is particularly important that students be given the opportunity to apply these ideas through programming projects. Nontrivial but tractable projects are not easy to design; for this reason, I have included an appendix containing a detailed specification of a large but manageable multiprogramming project which I have used successfully several times.

I have tried to reference all the material carefully so that the reader can pursue some area in further depth or obtain another point of view, and so that proper credit is given to the source of each technique or idea. I sincerely regret any errors or omissions in the latter. All references are collected at the end of the book and are cited in the text by the last name of the author followed by a date, e.g., Dijkstra, 1965b.

ACKNOWLEDGEMENTS

I am very grateful to a number of people for their help, encouragement, and intellectual influence during the preparation of this manuscript. W. F. Miller first introduced me to the pleasures and satisfaction of research and scholarship, and provided early support of my book writing in operating systems. I had the privilege of assisting N. Wirth in a systems programming class at Stanford University in 1965–66 and produced a set of notes based on his lectures;† these notes contained some of the principal ideas underlying the design and construction of operating systems and compilers. I am also indebted to Wirth for showing me that systems design and programming can be a scientific activity.

J. George and J. Horning read the manuscript and offered many constructive suggestions. The book developed from the "laboratories" of my operating systems classes at Cornell University and the University of Washington; I thank the students in these classes for their stimulation, energy, curiosity, good humor, and willingness to help define and organize a new field. G. Andrews, R. Holt, N. Weiderman, and T. Wilcox were especially helpful, not only in the above capacity but also as active research colleagues. In particular, parts of Chapter 7 use the results of the Ph.D. work of Weiderman and Chapter 8 is based on Holt's Ph.D. research.

My final acknowledgements go to the many researchers and practitioners who have contributed to the development of the field of operating systems. I have been most influenced by the published works of E. W. Dijkstra, and discuss his contributions throughout the book.

Alan C. Shaw

Seattle, Wash.

†Shaw, A. C., *Lecture Notes on a Course in Systems Programming.* Tech. Report No. 52, Computer Science Dept., Stanford University, Stanford, Calif. Dec. 1966. (Available from the Clearinghouse for Federal, Scientific, and Technical Information, U.S. Dept. of Commerce, National Bureau of Standards, Springfield, Virginia 22151 (Accession No. PB176762)).

CONTENTS

9 FILE SYSTEMS 244

1 THE ORGANIZATION OF COMPUTING SYSTEMS

The term *logical design* is used by computer designers to describe a systematic methodology, based on Boolean algebra, for designing switching networks. This book uses the term in a broader sense to denote a general method of *reasoning* about operating systems which allows their systematic design, and the study of their organization and behavior. Our emphasis is on general principles as opposed to *ad hoc* "tricks;" thus, coding techniques are not discussed in great detail, nor do we present a case study of a particular commercial system, even though many examples from the latter are given to illustrate particular points.

This chapter introduces the subject by examining the historical development of hardware and software components, by briefly outlining the organization and functions of computing systems, and by discussing systems programs from several different points of view. First, some basic terminology is defined.

1.1. SOME DEFINITIONS

Words such as "operating system," "time-sharing,"or "multiprogramming" do not have widely accepted precise definitions, except perhaps within the context of a theoretical study restricted to some small aspect of systems. Instead, these terms denote certain types of organization, functions, behavior, and/or methods of operation. With this in mind, we informally define several important terms commonly used to describe systems.

An *operating* (supervisory, monitor, executive) *system* (OS) is an *organized* collection of (systems) programs that acts as an interface between machine hardware and users, providing users with a set of facilities to simplify the design, coding, debugging, and maintenance of programs; and, at the same time, controlling the allocation of resources to assure efficient operation.

There are three categories of "pure" OS's, each of which may be characterized by the type of interaction permissible between a user and his job, and by the tolerances on system response time:

1. A *batch processing* OS is one in which user jobs are submitted in sequential batches on input devices and there is no interaction between a user and his job during processing. The user is completely isolated from his job and, as a result, equates system response with job turnaround time. The latter is generally satisfactory if it can be measured in small numbers of minutes or hours. Consequently, the OS can follow a relatively flexible scheduling policy.

2. A *time-sharing* OS is a system that provides computational services to many on-line users concurrently, allowing each user to interact with his computations. The effect of simultaneous access is achieved by sharing processor time and other resources among several users in a manner that guarantees some response to each user command within a few seconds. The computer is allocated to each user process for a small "time-slice," normally in the millisecond range; if the process is not completed at the end of its slice, it is interrupted and placed on a waiting queue, permitting another process its turn at the machine.

3. A *real-time* OS is one that services on-line external processes having *strict* timing constraints on response. Interrupt signals from external processes command the attention of the system; if they are not handled promptly (in microseconds, milliseconds, or seconds, depending on the process), the external process is seriously degraded or misrepresented. These systems are often designed for a particular application, for example, process control.

A particular OS might provide for any or all of batch processing, time-sharing or real-time jobs. For example, both real-time and time-sharing systems usually process batch jobs in the "background" when there is no on-line or external activity.

The most common method for implementing OS's is through multiprogramming. A *multiprogrammed* (multiprogramming) *OS* (MS) is one that maintains more than one user program in main storage simultaneously, sharing processor time, storage space, and other resources among the active user jobs. This resource sharing extends into the operating system; the programs comprising the OS are themselves multiprogrammed in most large systems.

Another way for an OS to process several jobs at once is by swapping. A *swapping OS* maintains several jobs on secondary storage and only one job in main storage at any time; the system switches to another job by moving the current job out of main store and loading a selected job from auxiliary storage. If the previous job is not completed, it will be swapped back in at a later time. This technique has been used mainly in small time-sharing systems.

Our last definition, multiprocessing, describes the hardware configuration of a system and is sometimes confused with multiprogramming. A *multiprocessing computer system* is a computer hardware complex with more than one independent processing unit. This includes central processors (CPU's), input-output (IO) processors, data channels, and special purpose processors, such as arithmetic units. Most often, the term refers to multiple central processing units.

This book is concerned primarily with multiprogrammed operating systems—the advantages and disadvantages of this organization as compared to others, and the techniques and requirements for time, space, and other resource sharing.

1.2. NOTATION FOR ALGORITHMS

The programming language Algol 60 (Naur, 1963), and recognizable variations thereof, will be our primary means for specifying algorithms. Algol, rather than English, flow charts, assembly language, or some other higher level language, was selected for the following reasons:

1. The syntax and semantics of Algol are clearly defined, with little ambiguity, in the public literature.

2. It has been used successfully for many years as an international publication language for algorithms.

3. Algol-like descriptions can be sufficiently "high-level" to eliminate many housekeeping details, if that is desired. Conversely, it can be used in a "low-level" manner that maps into machine language in a straightforward way.†

4. The author and many of his students and colleagues have found this to be a powerful notation for deriving, organizing, and analyzing algorithms. [For an introduction to Algol 60 and a copy of the original report, see Rosen (1967, pp. 48–117).]

†While we ascribe to the general philosophy of structured programming (see, for example, Dijkstra, 1969 and SIGPLAN, 1972), the reader will still find **go to** statements in some of our programs. **go to**'s have not been totally eliminated in favor of some other constructs because they are useful for describing machine level activities in a clear way and for explicitly exhibiting flow of control, yet they can also be used in a *disciplined* manner to yield well-structured programs.

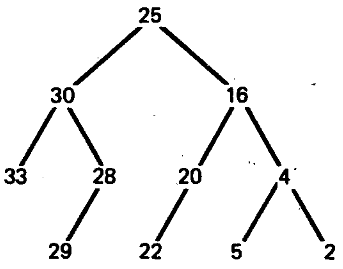
Example of an Algol procedure

Tree-like data structures are often used in operating systems, for example, to represent process hierarchies (Chapter 7) or file directories (Chapter 9).

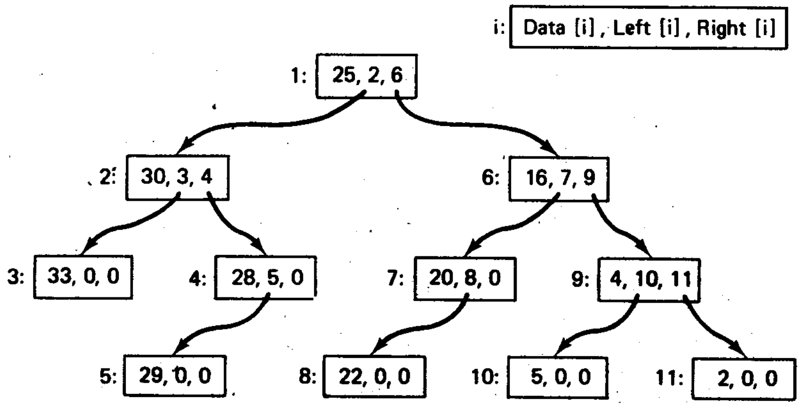
A *binary tree* consists of a finite set of nodes that is either empty or can be divided into a root node and two disjoint binary trees, the left and right subtrees (Knuth, 1968). Let each node n in a binary tree be represented by the triple $(Data[n], Left[n], Right[n])$, where $Data[n]$ is a positive integer, $Left[n]$ is a nonnegative pointer to the root of the left subtree, and $Right[n]$ is a nonnegative pointer to the right subtree. Reserve the node pointer $n = 0$ for the empty tree. Assume that for each node n:

$$(1)\qquad \begin{aligned} &Data[n] < Data[x] \quad \text{for all } x \in Leftsubtree(n) \quad \text{and} \\ &Data[n] > Data[x] \quad \text{for all } x \in Rightsubtree(n). \end{aligned}$$

Figure 1-1 contains an example of such a tree. [Symbol tables are some-



(a) A Binary Tree



(b) Internal Representation

Fig. 1-1 Binary tree organized for fast sorting, searching, and inserting.

times organized in this manner so they may be expanded and searched easily (Gries, 1971)]. Below is an Algol procedure *Treesearch*(*root*, *arg*, *m*) which will search the tree with root *root* for a node *n* such that *Data*[*n*] = *arg*; it will return **true** and set *m* to the matching node if successful, and **false** otherwise. The algorithm uses the recursive definition of a binary tree directly.

```
Boolean procedure Treesearch(root, arg, m) ;
value root, arg ; integer root, arg, m ;
comment Data [ ], Left [ ], Right [ ] are assumed global to this procedure.
Search the tree with root root for the node n such that Data[n] = arg ;
if root ≐ 0 then Treesearch := false else
begin
  integer d ;
  d := Data[root] ;
  if arg = d then begin m := root ; Treesearch := true end
  else
  if arg > d then Treesearch := Treesearch(Left[root], arg, m)
  else Treesearch := Treesearch(Right[root], arg, m)
end
```

EXERCISE

Write an Algol procedure *Addtotree*(*root*, *n*) which takes the binary tree with root *root* and node ordering defined by (1), and adds the isolated node *n* to it retaining the ordering of (1). Write another procedure which prints the data of the tree in ascending sequence; use any convenient primitive, such as *Write*(*x*), as an output call to print the variable *x*.

1.3 HISTORICAL PERSPECTIVE

This section briefly describes the historical evolution of computer hardware and software systems. A more detailed discussion and bibliography can be found in S. Rosen (1969) and R. Rosin (1969).

1.3.1 Early Systems

From about 1949, when the first stored program digital computer actually started executing instructions, until 1956, the basic organization and mode of operation of computers remained relatively constant (with some farsighted but mostly unsuccessful exceptions). Their classical von Neumann architecture was predicated on strictly sequential instruction execution including input-output operations. When loading and running programs, users would work at the console directly on-line to the machine, setting registers, stepping through instructions, examining storage locations, and generally interacting with their computation at the lowest machine level. (Time-sharing systems were a recognition of the advantages of operating in this fashion but at a higher level than the "raw" hardware). Programs were written in absolute machine language (decimal or octal notation) and were preceded by an absolute loader.

It is instructive to review the procedures for absolute loading since, even now, they represent the starting point for any software system on a raw machine. Any computer has the equivalent of a Load button; when pressed by an operator, it will cause the computer to read an input data record into some fixed set of contiguous main storage locations and then transfer control, i.e., set the instruction counter of the machine, to a fixed address in that set, usually the first.

Example

Let main storage of a primitive computer be designated $M[0]$, $M[1]$, $M[2]$, . . . , where each location $M[i]$ may contain one byte (8 bits) of information. Suppose that pressing the Load button will cause one 80-column card with 80 bytes of information to be read into $M[0], \ldots, M[79]$, followed by the setting of the instruction counter to zero; i.e.,

```
PressLoad:  Read(for $i := 0$ step 1 until 79 do $M[i]$) ;
            Transferto($M[0]$) ;
```

In order to read an absolute program, the first card, the one read by *PressLoad*, must contain machine instructions for reading succeeding cards (or at least the next card). Let each address, instruction, and datum in our primitive machine occupy 1 byte. Assume that the absolute program is punched on cards with the following format:

Card columns	Contents
1	loading address LA for 1st byte of program/data part of card.
2	the number of bytes, n, to be loaded; $n \leq 78$.
3 to $(n + 2)$	program/data part; the absolute code.

The last card contains $n = 0$, and the "loading address" is interpreted as the first instruction, the entry point, of the program; Fig. 1-2 shows the required cards in order. Finally, let storage locations $M[r], \ldots, M[r + 79]$ be a reserved read-in area where r is arbitrarily assigned as the starting location of the read-in area. Then a one-card absolute loader, appearing on the first card, performs the following actions:

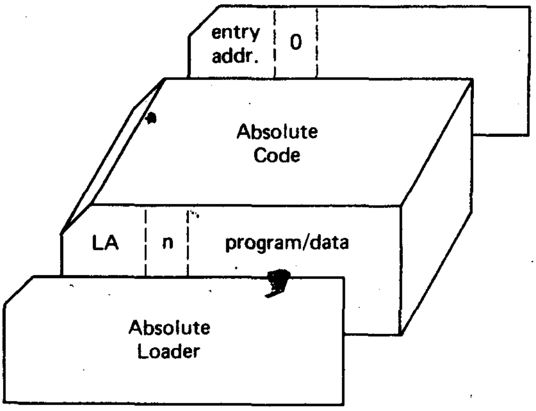


Fig. 1-2 Absolute loader and code cards.

```
Load:  $Read$(for $i$ := 0 step 1 until 79 do $M[r + i]$) ;
       $LA := M[r]$ ; $n := M[r + 1]$ ;
       if $n = 0$ then $Transferto(LA)$ ;
       for $i$ := 0 step 1 until $n - 1$ do
          $M[LA + i] := M[r + 2 + i]$ ;
       go to $Load$ ;
```

The loading process is a vivid example of bootstrapping—"pulling oneself up by one's own bootstraps."

In these early years, programming aids were either nonexistent or minimal —simple assemblers and interpreters at the most sophisticated installations, with little use of library routines. As the importance of symbolic programming was recognized and assembly systems came into more widespread use, a standard operating procedure evolved: A loader reads in an assembler; the assembler assembles into absolute code symbolic decks of user source programs and library routines; the assembled code is written on tape or cards, and a loader is again used to read these into main storage; the absolute program is then executed. Each step required manual assistance from an operator and consumed a great deal of time, especially in comparison with the computer time to process the cards at that step.

The "first generation" of operating systems was motivated by the above inefficiencies as well as by other considerations. These additional factors included the expense of on-line operation; the availability of other languages (the FORTRAN system being most prominent); the development of library programs and services especially related to input-output operations; and the awkwardness of translating into absolute code, which required that *all*