

Second Edition

OPERATING SYSTEMS

A Systematic View

WILLIAM S. DAVIS



MEDIATEK-1043
7,483,686

Sponsoring Editor: William B. Gruener
Production Editor: William J. Yskamp

Text Designer: Melinda Grosser
Illustrator: ARVAK
Cover Designer: Ann Scrimgeour Rose
Cover Illustrator: Ann Scrimgeour Rose
Art Coordinator: Joseph K. Vetere

Production Manager: Karen M. Guardino
Production Coordinator: Peter Petraitis

The text of this book was composed in Century Schoolbook by TriStar Graphics

Library of Congress Cataloging in Publication Data

Davis, William S., 1943-
Operating systems.
Includes index.

1. Operating systems (Computers) I. Title
QA76.6.D38 1983 001.64'25 82-3926
ISBN 0-201-11116-0 AACR2

Reprinted with corrections, April 1984

Copyright © 1983 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

ISBN 0-201-11116-0
DEFGHIJ-DO-8987664

is more than enough time to roll user A's program out to disk and roll-in user B's program. In fact, the computer is so fast that, in all likelihood, all five users will have had a shot at core before the first one is ready with a response—a computer can easily keep up with five human beings working at human speeds.

An even better approach involves relocation of user programs as they come back from disk or drum. Very few time-sharing users are interested in programming at the assembly language level, with most working in BASIC, FORTRAN, APL, Pascal, PL/I or some other compiler language. Few compiler languages use registers as accumulators or counters, thus eliminating one of the major problems of dynamic relocation. The basic idea is simple: When a user is ready to resume processing, the program is rolled into any available partition, and not necessarily into the partition it occupied before. Using this dynamic approach, the possibility of drawing an unlucky "busy" partition would disappear. Assuming that the average user would require the computer for one full second (a pretty long shot of time) between I/O operations, it would be theoretically possible to support up to 30 users in a *single* partition and still be able to respond to each and every user's normal 30-second cycle (we're probably overestimating our user's capabilities here) without any delays. Responses, of course, are not regular (this is a good queueing theory problem), but four or five regions should be more than adequate for our 50-user system.

Processor Management and Internal Priorities

Whenever two or more programs are concurrently sharing the resources of a single CPU, it is inevitable that conflicts over CPU access will occur. The processor can support only one program at any given instant of time. When more than one program is ready to use the CPU, a decision must be made: who goes first? Dealing with this internal priority question is a key element of a multiuser operating system.

Multiprogramming and Processor Management

A common approach on a multiprogramming system is to link the priority decision to the memory allocation scheme. For example, consider a simple fixed-partition system with two application program partitions (Fig. 13.10). One of these partitions can be designated as the *foreground*, while the other is known as the *background*. When the system is ready to give control to an application

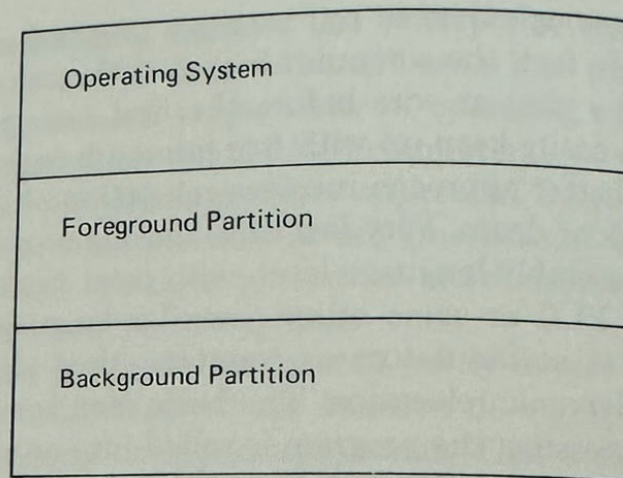


Fig. 13.10
Foreground and
background partitions.

program, it first checks the control fields for the foreground partition. If the foreground program is ready, it gets control; only if the foreground program is still waiting do the control fields for the background partition get checked. The foreground has high priority; the background has low priority.

This basic idea can easily be extended to larger system. With, for example, twelve application program partitions or regions, the *internal priority* module can be written to check partition A first, then partition B, and so on. The first partition checked has highest priority; the last has lowest priority. In fact, the only way the low-priority partition can get control is if *all* the higher-priority partitions are waiting for something.

How might such a priority scheme be implemented? One technique is to link the priority decision to the memory allocation tables described earlier in the chapter. It is necessary to keep track of the starting and ending address of each partition or region. Why not add a field to keep track of the status of the program in that partition? If you view the memory allocation table as an array, the priority decision would be implemented by starting with array element 1, checking the status bits, incrementing the subscript by 1, checking the next element's status bits, and continuing the search until a ready program is found.

Actually, there are a number of control fields that must be maintained in support of each main memory partition. Often, a single *control block* is created to hold these flags, constants, and variables (Fig. 13.11), with one control block for each partition (or region). A common approach is to maintain, in the system parameters region, a pointer to the high-priority partition's control block. The first control block points, in turn, to the second, which points to the third, and so on (Fig. 13.11, again). The priority module determines which program is to be given control by following this chain until a ready program is located. As you can see, under such a system, the priority of a program is fixed by its partition or region assignment.

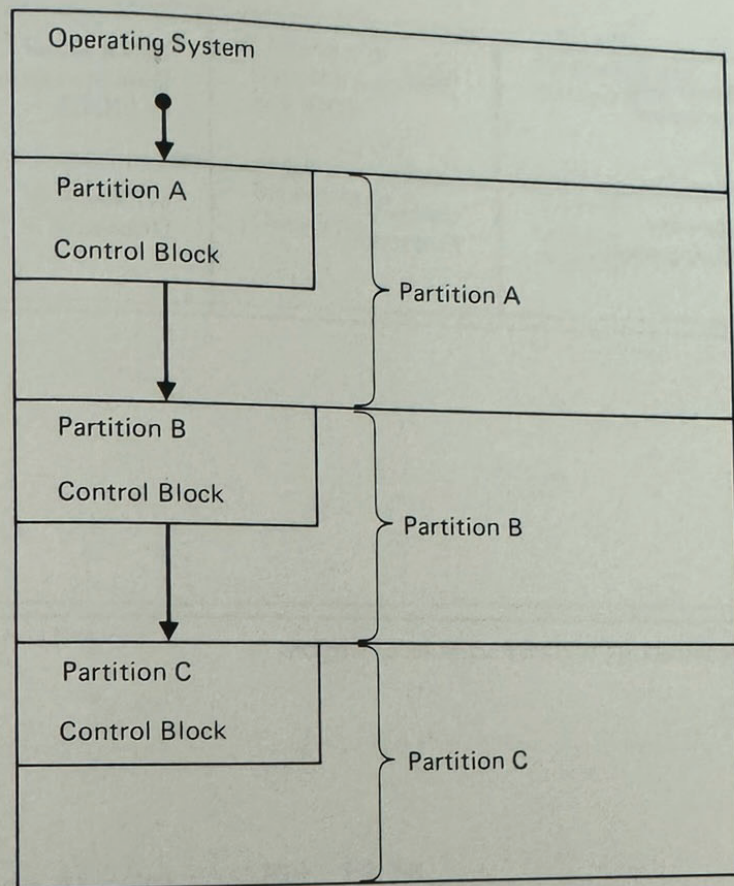


Fig. 13.11
Often, each partition
or region has a control
block to hold key
control information.

It is, of course, possible to implement more complex priority schemes. On some systems, for example, rather than always beginning the priority search with the same partition control block, the priority routine, often called the *scheduler* or *dispatcher*, can be designed to resume its search where it left off—this would tend to give all partitions roughly equal priority. Another alternative is to assign each program a priority through the command language, with this priority determining the order in which the control blocks are searched. Such factors as program size, time in memory, I/O device requirements, and other measures of the program's impact on system resources can be used to *compute* dynamically an internal priority, thus determining the search order. The more complex the scheduling algorithm, however, the more complex the scheduler operating system module (Fig. 13.12).

Interrupts and the Flow of Control

Throughout our discussion of the internal priority problem, we've been making an implicit assumption. The operating system, more particularly, the scheduler