

STALLINGS

COMPUTER ORGANIZATION
AND ARCHITECTURE

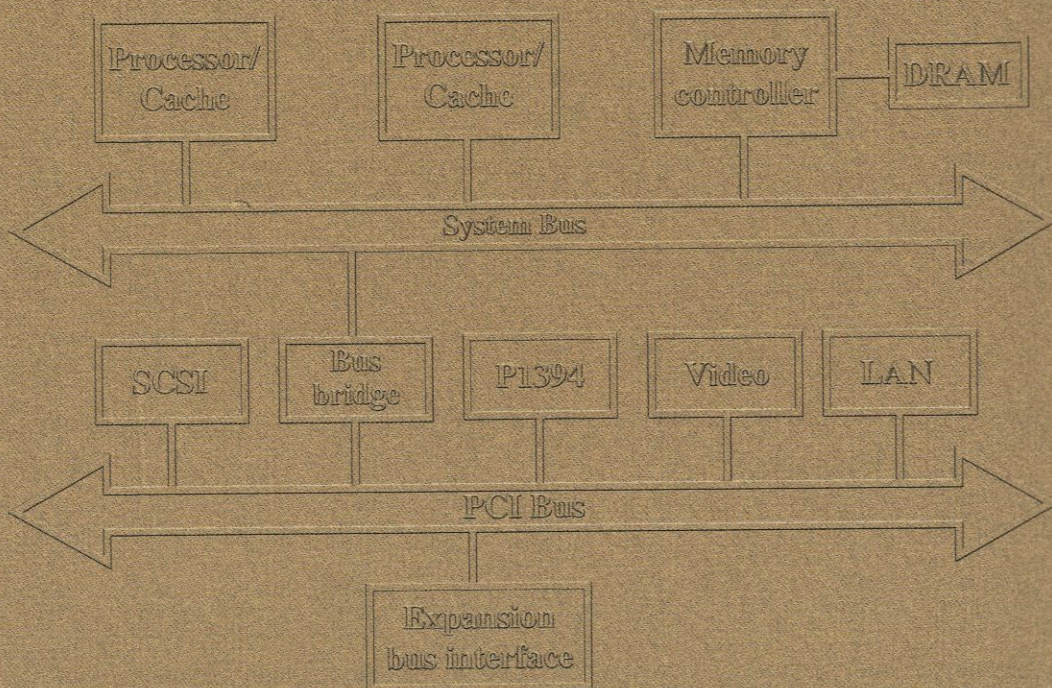
Designing for Performance
FOURTH EDITION

QA
76.9
C64
S73
1996
ENG

William Stallings

COMPUTER ORGANIZATION AND ARCHITECTURE

Designing for Performance



Fourth Edition

COMPUTER ORGANIZATION AND ARCHITECTURE

Designing for Performance

FOURTH EDITION

William Stallings

PRENTICE HALL

Upper Saddle River, New Jersey 07675

COMPUTER ORGANIZATION AND ARCHITECTURE

Designing for Performance

FOURTH EDITION

William Stallings



PRENTICE HALL

Upper Saddle River, New Jersey 07458

Library of Congress Cataloging-in-Publication Data

Stallings, William

Computer organization and architecture : designing for performance /

William Stallings. — 4th ed.

p. cm.

Includes bibliographical references and index.

ISBN 0-13-359985-X

1. Computer organization. 2. Computer architecture. I. Title.

QA76.9.C643S73 1996

004.2'2—dc20

96-9115

CIP

Acquisitions editor: Alan Apt
Editorial-production service: Electronic Publishing Services Inc.
Manufacturing buyer: Donna Sullivan
Cover design: Amy Rosen
Cover photograph: Jon Churchman
Production coordinator: Rick DeLorenzo
Editorial assistant: Shirley McGuire



© 1996 by Prentice-Hall, Inc.
Simon & Schuster / A Viacom Company
Upper Saddle River, New Jersey 07458

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, and arising out of, the furnishing, performance, or use of these programs.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Previous edition's copyright © 1993

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-359985-X

Prentice-Hall International (UK) Limited, London
Prentice-Hall of Austria 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

To my loving wife, Tricia

TABLE 7.4 Batch Programming Versus Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor utilization	Minimize response time
Source of instructions to operating system	Job control language instructions provided with the job	Commands entered at the terminal

well. This operating system is still marketed by Honeywell and is still considered an advanced system, particularly in the area of security. Many of the ideas developed for Multics were subsequently used at Bell Labs in the design of UNIX, which has become one of the most popular minicomputer operating systems and is now offered on many microcomputers.

7.2

SCHEDULING

The central task of modern operating systems is multiprogramming (with the exception of single-user microcomputers). With multiprogramming, multiple jobs or user programs are maintained in memory. Each job alternates between using the central processor and waiting for I/O to be performed. The processor keeps busy by executing one job while the others wait.

The key to multiprogramming is scheduling. In fact, three types of scheduling are typically involved (Table 7.5). We will explore these presently. But first, we introduce the concept of *process*. This term was first used by the designers of Mul-

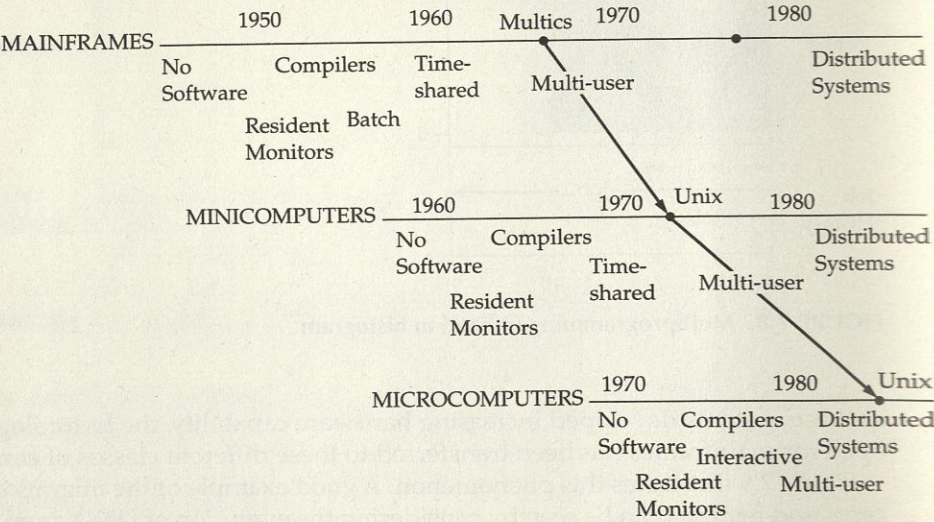


FIGURE 7.9. Migration of operating system concepts and features

TABLE 7.5 Scheduling For Multiprogramming

High-level scheduling	The decision to add to the pool of programs to be executed.
Short-term scheduling	The decision as to which available process shall be executed by the processor.
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device.

tics in the 1960s. It is a somewhat more general term than *job*. Many definitions have been given for the term *process*, including

- A program in execution.
- The "animated spirit" of a program.
- That entity to which a processor is assigned.

This concept should become clearer as we proceed.

High-Level Scheduling

The high-level scheduler determines which programs are admitted to the system for processing. Thus, the high-level scheduler controls the *degree of multiprogramming* (number of processes in memory). Once admitted, a job or program becomes a process and is added to a queue for the short-term scheduler. The more processes that are created, the smaller is the percentage of time that each process can be executed. The long-term scheduler may limit the degree of multiprogramming to provide satisfactory service to its current set of processes.

In a batch system, newly submitted jobs are routed to disk and held in a queue or waiting line. The high-level scheduler adds jobs from that queue when it can. There are two decisions involved here. First, the scheduler must decide that it can take on one or more additional processes. This decision is generally made each time an existing process is completed. Second, the scheduler must decide which job or jobs to accept and turn into processes. The criteria used may include priority, expected execution time, and I/O requirements.

For time sharing, a process request is generated by the act of a user attempting to connect to the system. Time-sharing users are not simply queued up and kept waiting until the system can accept them. Rather, the operating system will accept all authorized comers until the system is saturated. At that point, a connection request is met with a message indicating that the system is full and the user should try again later.

Short-Term Scheduling

The high-level scheduler executes relatively infrequently and makes the coarse-grained decision of whether or not to take on a new process, and which one to take. The short-term scheduler, also known as the *dispatcher*, executes frequently and makes the fine-grained decision of which job to execute next.

Process States

To understand the operation of the short-term scheduler, we need to consider the concept of a process state. During the lifetime of a process, its status will change a number of times. Its status at any point in time is referred to as a *state*. The *term state* is used because it connotes that certain information exists that defines the status at that point. Typically, there are five defined states for a process (Figure 7.10):

- **New:** A program is admitted by the high-level scheduler but is not yet ready to execute. The operating system will initialize the process, moving it to the ready state.
- **Ready:** The process is ready to execute and is awaiting access to the processor.
- **Running:** The process is being executed by the processor.
- **Waiting:** The process is suspended from execution waiting for some system resource, such as I/O.
- **Halted:** The process has terminated and will be destroyed by the operating system.

For each process in the system, the operating system must maintain state information indicating the status of the process and other information necessary for process execution. For this purpose, each process is represented in the operating system by a *process control block* (Figure 7.11), which typically contains

- **Identifier:** Each current process has a unique identifier.
- **State:** The current state of the process (new, ready, and so on).
- **Priority:** Relative priority level.
- **Program Counter:** The address of the next instruction in the program to be executed.
- **Memory Pointers:** The starting and ending locations of the process in memory.
- **Context Data:** These are data that are present in registers in the processor while the process is executing, and they will be discussed in Part III. For now, it is enough to say that these data represent the "context" of the process. The context data plus the program counter are saved when the process leaves the ready state. They are retrieved by the processor when it resumes execution of the process.
- **I/O Status Information:** Includes outstanding I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files assigned to the process, and so on.
- **Accounting Information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

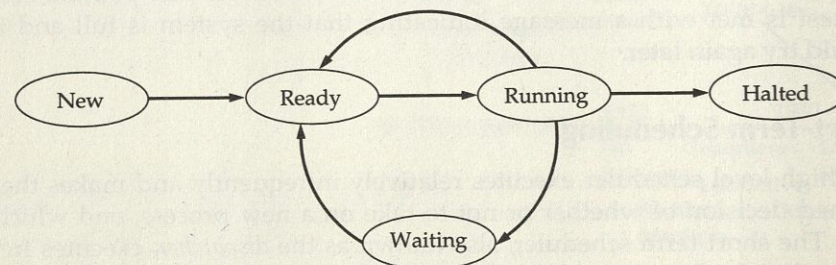


FIGURE 7.10. Process states

Identifier
State
Priority
Program Counter
Memory Pointers
Context Data
I/O Status Information
Accounting Information
⋮

FIGURE 7.11. Process control block

When the processor accepts a new job or user request for execution, it creates a blank process control block and places the associated process in the new state. After the system has properly filled in the process control block, the process is transferred to the ready state.

Scheduling Techniques

To understand how the operating system manages the scheduling of the various jobs in memory, let us begin by considering the simple example in Figure 7.10. The figure shows how main memory is partitioned at a given point in time. The nucleus of the operating system is, of course, always resident. In addition, there are a number of active processes, including A and B, each of which is allocated a portion of memory.

We begin at a point in time (Figure 7.12a) when process A is running. The processor is executing instructions from the program contained in A's memory partition. At some later point in time (Figure 7.12b), the processor ceases to execute instructions in A and begins executing instructions in the operating system area. This will happen for one of three reasons:

1. Process A issues a service call (e.g., an I/O request) to the operating system. Execution of A is suspended until this call is satisfied by the operating system.
2. Process A causes an *interrupt*. An interrupt is a hardware-generated signal to the processor. When this signal is detected, the processor ceases to execute A and transfers to the interrupt handler in the operating system. A variety of events related to A will cause an interrupt. One example is an error, such as attempting to execute a privileged instruction. Another example is a timeout; to prevent any one process from monopolizing the processor, each process is only granted the processor for a short period at a time.

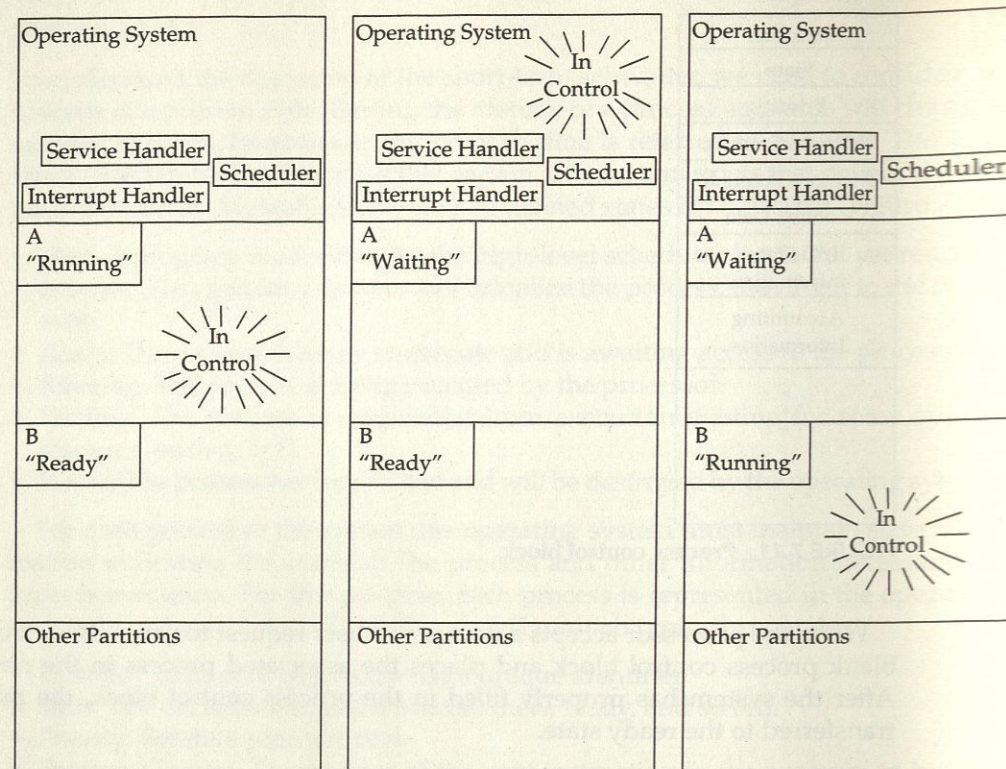


FIGURE 7.12. Scheduling example

- Some event unrelated to process A that requires attention causes an interrupt. An example is the completion of an I/O operation.

In any case, the result is the following. The processor saves the current **context** data and the program counter for A in A's process control block and then **begins** executing in the operating system. The operating system may perform **some** work, such as initiating an I/O operation. Then the short-term-scheduler **portion** of the operating system decides which process should be executed next. **In this** example, B is chosen. The operating system instructs the processor to restore B's context data and proceed with the execution of B where it left off (Figure 7.12c).

This simple example highlights the basic functioning of the short-term scheduler. Figure 7.13 shows the major elements of the operating system involved in the multiprogramming and scheduling of processes. The operating system **receives** control of the processor at the interrupt handler if an interrupt occurs and **at the** service-call handler if a service call occurs. Once the interrupt or service call is handled, the short-term scheduler is invoked to pick a process for execution.

To do its job, the operating system maintains a number of queues. Each **queue** is simply a waiting list of processes waiting for some resource. The *long-term queue* is a list of jobs waiting to use the system. As conditions permit, the high-level scheduler will allocate memory and create a process for one of the waiting items. The

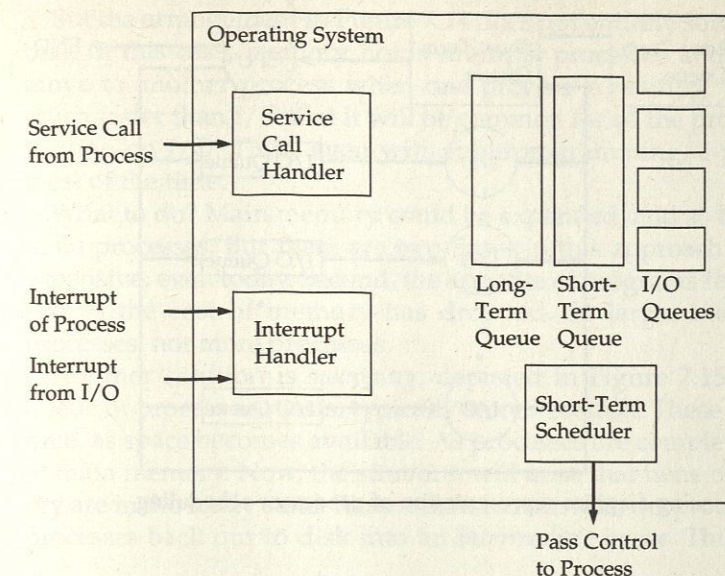


FIGURE 7.13. Key elements of an operating system for multiprogramming

short-term queue consists of all processes in the ready state. Any one of these processes could use the processor next. It is up to the short-term scheduler to pick one. Generally, this is done with a round-robin algorithm, giving each process some time in turn. Priority levels may also be used. Finally, there is an *I/O queue* for each I/O device. More than one process may request the use of the same I/O device. All processes waiting to use each device are lined up in that device's queue.

Figure 7.14 suggests how processes progress through the computer under the control of the operating system. Each process request (batch job, user-defined interactive job) is placed in the long-term queue. As resources become available, a process request becomes a process and is then placed in the ready state and put in the short-term queue. The processor alternates between executing operating system instructions and executing user processes. While the operating system is in control, it decides which process in the short-term queue should be executed next. When the operating system has finished its immediate tasks, it turns the processor over to the chosen process.

As was mentioned earlier, a process being executed may be suspended for a variety of reasons. If it is suspended because the process requests I/O, then it is placed in the appropriate I/O queue. If it is suspended because of a timeout or because the operating system must attend to pressing business, then it is placed in the ready state and put right back into the short-term queue.

Finally, we mention that the operating system also manages the I/O queues. When an I/O operation is completed, the operating system removes the satisfied process from that I/O queue and places it in the short-term queue. It then selects another waiting process (if any) and signals for the I/O device to satisfy that process's request.

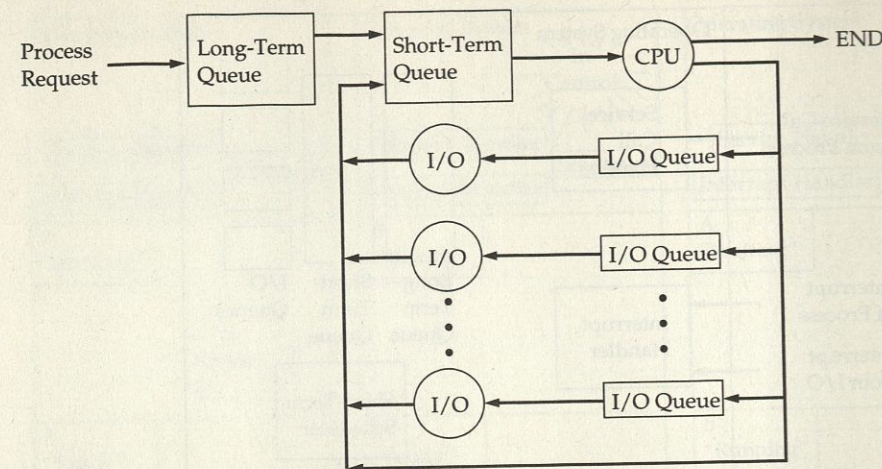


FIGURE 7.14. Queuing diagram representation of processor scheduling

We have said that the major task of the operating system is to manage the system's resources. The queuing technique depicted in Figure 7.14 illustrates how the operating system manages processor time and I/O devices. There is one remaining major resource to manage, namely memory, and we now turn to this topic.

7.3

MEMORY MANAGEMENT

In a uniprogramming system, main memory is divided into two parts: one part for the operating system (resident monitor) and one part for the program currently being executed. In a multiprogramming system, the "user" part of memory **must** be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as *memory management*.

Effective memory management is vital in a multiprogramming system. If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle. Thus, memory needs to be allocated efficiently to pack as many processes into memory as possible.

Swapping

Referring back to Figure 7.14, we have discussed three types of queues: the long-term queue of requests for new processes, the short-term queue of processes ready to use the processor, and the various I/O queues of processes that are not ready to use the processor. Recall that the reason for this elaborate machinery is that I/O activities are much slower than computation and therefore the processor in a uniprogramming system is idle most of the time.

But the arrangement in Figure 7.14 does not entirely solve the problem. It is true that, in this case, memory holds multiple processes and that the processor can move to another process when one process is waiting. But the processor is so much faster than I/O that it will be common for *all* the processes in memory to be waiting on I/O. Thus, even with multiprogramming, a processor could be idle most of the time.

What to do? Main memory could be expanded, and so be able to accommodate more processes. But there are two flaws in this approach. First, main memory is expensive, even today. Second, the appetite of programs for memory has grown as fast as the cost of memory has dropped. So larger memory results in larger processes, not more processes.

Another solution is *swapping*, depicted in Figure 7.15. We have a long-term queue of process requests, typically stored on disk. These are brought in, one at a time, as space becomes available. As processes are completed, they are moved out of main memory. Now, the situation will arise that none of the processes in memory are in the ready state. Rather than remain idle, the processor *swaps* one of these processes back out to disk into an *intermediate queue*. This is a queue of existing

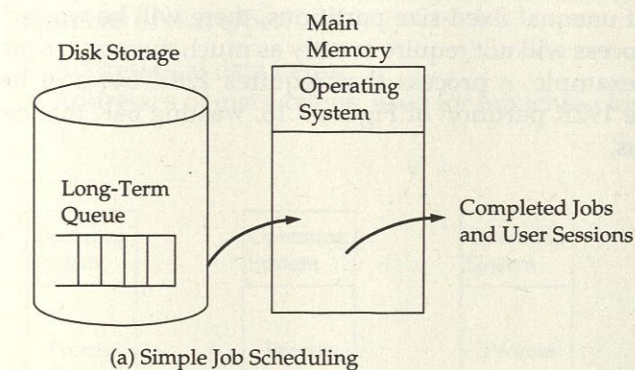
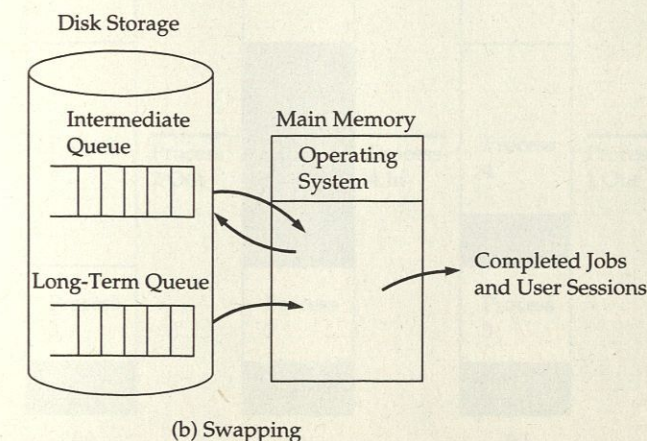


FIGURE 7.15. The use of swapping