

**SILBERSCHATZ
GALVIN
GAGNE**

**OPERATING
SYSTEM
CONCEPTS**

SIXTH EDITION

ACQUISITIONS EDITOR	Paul Crockett
SENIOR MARKETING MANAGER	Katherine Hepburn
SENIOR PRODUCTION EDITOR	Ken Santor
COVER DESIGNER	Madelyn Lesure
COVER ART	Susan E. Cyr
SENIOR ILLUSTRATION COORDINATOR	Anna Melhorn

This book was set in Palatino by Abraham Silberschatz and printed and bound by Courier-Westford. The cover was printed by Phoenix Color Corporation.

This book is printed on acid-free paper.

The paper in this book was manufactured by a mill whose forest management programs include sustained yield harvesting of its timberlands. Sustained yield harvesting principles ensure that the numbers of trees cut each year does not exceed the amount of new growth.

Copyright © 2002 John Wiley & Sons, Inc. All rights reserved.

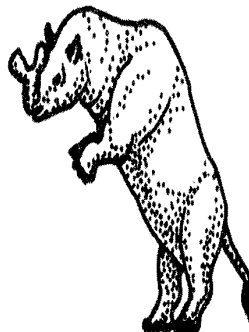
No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (508) 750-8400, fax (508) 750-4470. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, (212) 850-6011, fax (212) 850-6008, E-Mail: PERMREQ@WILEY.COM.

ISBN 0-471-41743-2

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Chapter 9



MEMORY MANAGEMENT

In Chapter 6, we showed how the CPU can be shared by a set of processes. As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer's response to its users. To realize this increase in performance, however, we must keep several processes in memory; that is, we must *share* memory.

In this chapter, we discuss various ways to manage memory. The memory-management algorithms vary from a primitive bare-machine approach to paging and segmentation strategies. Each approach has its own advantages and disadvantages. Selection of a memory-management method for a specific system depends on many factors, especially on the *hardware* design of the system. As we shall see, many algorithms require hardware support, although recent designs have closely integrated the hardware and operating system.

9.1 ■ Background

As we saw in Chapter 1, memory is central to the operation of a modern computer system. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the

operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore *how* a memory address is generated by a program. We are interested in only the sequence of memory addresses generated by the running program.

9.1.1 Address Binding

Usually, a program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The collection of processes on the disk that is waiting to be brought into memory for execution forms the **input queue**.

The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process does not need to be 00000. This arrangement affects the addresses that the user program can use. In most cases, a user program will go through several steps—some of which may be optional—before being executed (Figure 9.1). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as *count*). A compiler will typically **bind** these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”). The linkage editor or loader will in turn bind these relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time:** If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know a priori that a user process resides starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are absolute code bound at compile time.
- **Load time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case,

fir
ne
• E
or
tit
bc
th

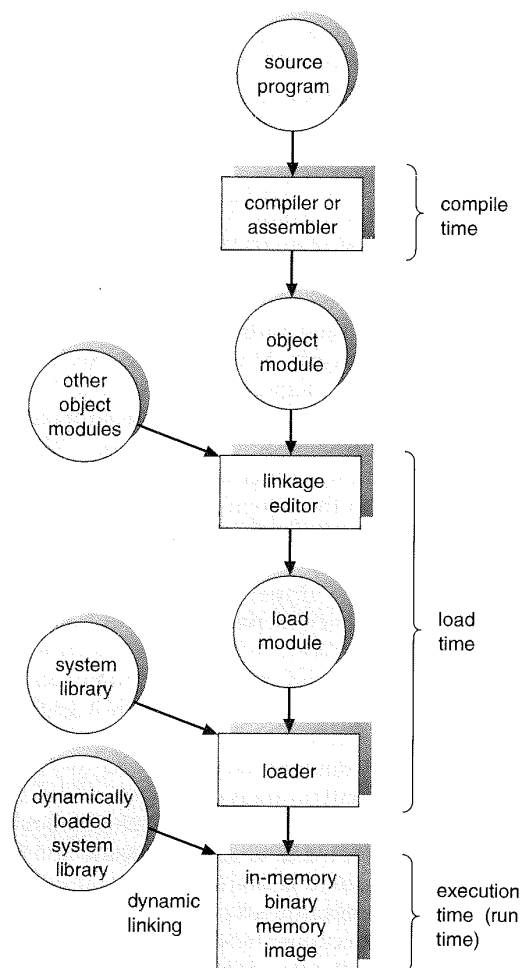


Figure 9.1 Multistep processing of a user program.

final binding is delayed until load time. If the starting address changes, we need only to reload the user code to incorporate this changed value.

- **Execution time:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed in Section 9.1.2. Most general-purpose operating systems use this method.

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts



Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research



With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips



Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.