

Chapter 1 – Introduction to Operating Systems

Outline

- 1.1 **Introduction**
- 1.2 **What Is an Operating System?**
- 1.3 **Early History: The 1940s and 1950s**
- 1.4 **The 1960s**
- 1.5 **The 1970s**
- 1.6 **The 1980s**
- 1.7 **History of the Internet and World Wide Web**
- 1.8 **The 1990s**
- 1.9 **2000 and Beyond**
- 1.10 **Application Bases**
- 1.11 **Operating System Environments**
- 1.12 **Operating System Components and Goals**
 - 1.12.1 **Core Operating System Components**
 - 1.12.2 **Operating System Goals**



Chapter 1 – Introduction to Operating Systems

Outline (continued)

- 1.13 **Operating System Architectures**
 - 1.13.1 **Monolithic Architecture**
 - 1.13.2 **Layered Architecture**
 - 1.13.3 **Microkernel Architecture**
 - 1.13.4 **Networked and Distributed Operating Systems**



Objectives

- After reading this chapter, you should understand:
 - what an operating system is.
 - a brief history of operating systems.
 - a brief history of the Internet and the World Wide Web.
 - core operating system components.
 - goals of operating systems.
 - operating system architectures.

1.1 Introduction

- Unprecedented growth of computing during the past several decades.
- Desktop workstations execute billions of instructions per second (BIPS)
- Supercomputers can execute over a trillion instructions per second
- Computers are now employed in almost every aspect of life.

1.2 What Is an Operating System?

- Some years ago an operating system was defined as the software that controls the hardware.
- Landscape of computer systems has evolved significantly, requiring a more complicated definition.
- Applications are now designed to execute concurrently.

1.2 What Is an Operating System?

- Separates applications from the hardware they access
 - Software layer
 - Manages software and hardware to produce desired results
- Operating systems primarily are resource managers
 - Hardware
 - Processors
 - Memory
 - Input/output devices
 - Communication devices
 - Software applications

1.3 Early History: The 1940s and 1950s

- Operating systems evolved through several phases
 - 1940s
 - Early computers did not include operating systems
 - 1950s
 - Executed one job at a time
 - Included technologies to smooth job-to-job transitions
 - Single-stream batch-processing systems
 - Programs and data submitted consecutively on tape

1.4 The 1960s

- 1960s
 - Still batch-processing systems
 - Process multiple jobs at once
 - Multiprogramming
 - One job could use processor while other jobs used peripheral devices
 - Advanced operating systems developed to service multiple interactive users
- 1964
 - IBM announced System/360 family of computers

1.4 The 1960s

- Timesharing systems
 - Developed to support many simultaneous interactive users
 - Turnaround time was reduced to minutes or seconds
 - Time between submission of job and the return of its results
 - Real-time systems
 - Supply response within certain bounded time period
 - Improved development time and methods
 - MIT used CTSS system to develop its own successor, Multics
 - TSS, Multics and CP/CMS all incorporated virtual memory
 - Address more memory locations than actually exist

1.5 The 1970s

- Primarily multimode timesharing systems
 - Supported batch processing, timesharing and real-time applications
 - Personal computing only in incipient stages
 - Fostered by early developments in microprocessor technology
- Department of Defense develops TCP/IP
 - Standard communications protocol
 - Widely used in military and university settings
 - Security problems
 - Growing volumes of information passed over vulnerable communications lines.

1.6 The 1980s

- 1980s
 - Decade of personal computers and workstations
 - Computing distributed to sites at which it was needed
 - Personal computers proved relatively easy to learn and use
 - Graphical user interfaces (GUI)
 - Transferring information between computers via networks became more economical and practical

1.6 The 1980s

- Client/server computing model became widespread
 - Clients request various services
 - Servers perform requested services
- Software engineering field continued to evolve
 - Major thrust by the United States government aimed at tighter control of Department of Defense software projects
 - Realizing code reusability
 - Greater degree of abstraction in programming languages
 - Multiple threads of instructions that could execute independently

1.7 History of the Internet and World Wide Web

- Advanced Research Projects Agency (ARPA)
 - Department of Defense
 - In late 1960s, created and implemented ARPAnet
 - Grandparent of today's Internet
 - Networked main computer systems of ARPA-funded institutions
 - Capable of near-instant communication via e-mail
 - Designed to operate without centralized control

1.7 History of the Internet and World Wide Web

- Transmission Control Protocol/Internet Protocol
 - Set of rules for communicating over ARPANet
 - TCP/IP manages communication between applications
 - Ensure that messages routed properly from sender to receiver
 - Error-correction
 - Later opened to general commercial use

1.7 History of the Internet and World Wide Web

- World Wide Web (WWW)
 - Locate and view multimedia-based documents on almost any subject
 - Early development begun in 1989 at CERN by Tim Berners-Lee
 - Technology for sharing information via hyperlinked text documents
 - HyperText Markup Language (HTML)
 - Defines documents on WWW
 - Hypertext Transfer Protocol (HTTP)
 - Communications backbone used to transfer documents across WWW

1.8 The 1990s

- Hardware performance improved exponentially
 - Inexpensive processing power and storage
 - Execute large, complex programs on personal computers.
 - Economical machines for extensive database and processing jobs
 - Mainframes rarely necessary
 - Shift toward distributed computing rapidly accelerated
 - Multiple independent computers performing common task

1.8 The 1990s

- Operating system support for networking tasks became standard
 - Increased productivity and communication
- Microsoft Corporation became dominant
 - Windows operating systems
 - Employed many concepts used in early Macintosh operating systems
 - Enabled users to navigate multiple concurrent applications with ease.
- Object technology became popular in many areas of computing
 - Many applications written in object-oriented programming languages
 - For example, C++ or Java
 - Object-oriented operating systems (OOOS)
 - Objects represent components of the operating system
 - Concepts such as inheritance and interfaces
 - Exploited to create modular operating systems
 - Easier to maintain and extend than systems built with previous techniques

1.8 The 1990s

- Most commercial software sold as object code
 - The source code not included
 - Enables vendors to hide proprietary information and programming techniques
- Free and open-source software became increasingly common in the 1990s
 - Open-source software distributed with the source code
 - Allows individuals to examine and modify software
 - Linux operating system and Apache Web server both open-source
- Richard Stallman launched the GNU project
 - Recreate and extend tools for AT&T's UNIX operating system
 - He disagreed with concept of paying for permission to use software

1.8 The 1990s

- Open Source Initiative (OSI)
 - Founded to further benefits of open-source programming
 - Facilitates enhancements to software products
 - Permits anyone to test, debug and enhance applications
 - Increases chance that subtle bugs will be caught and fixed
 - Crucial for security errors which need to be fixed quickly
 - Individuals and corporations can modify the source
 - Create custom software to meet needs of certain environment

1.8 The 1990s

- Operating systems became increasingly user friendly
 - GUI features pioneered by Apple widely used and improved
 - “Plug-and-play” capabilities built into operating systems
 - Enable users to add and remove hardware components dynamically
 - No need to manually reconfigure operating system

1.9 2000 and Beyond

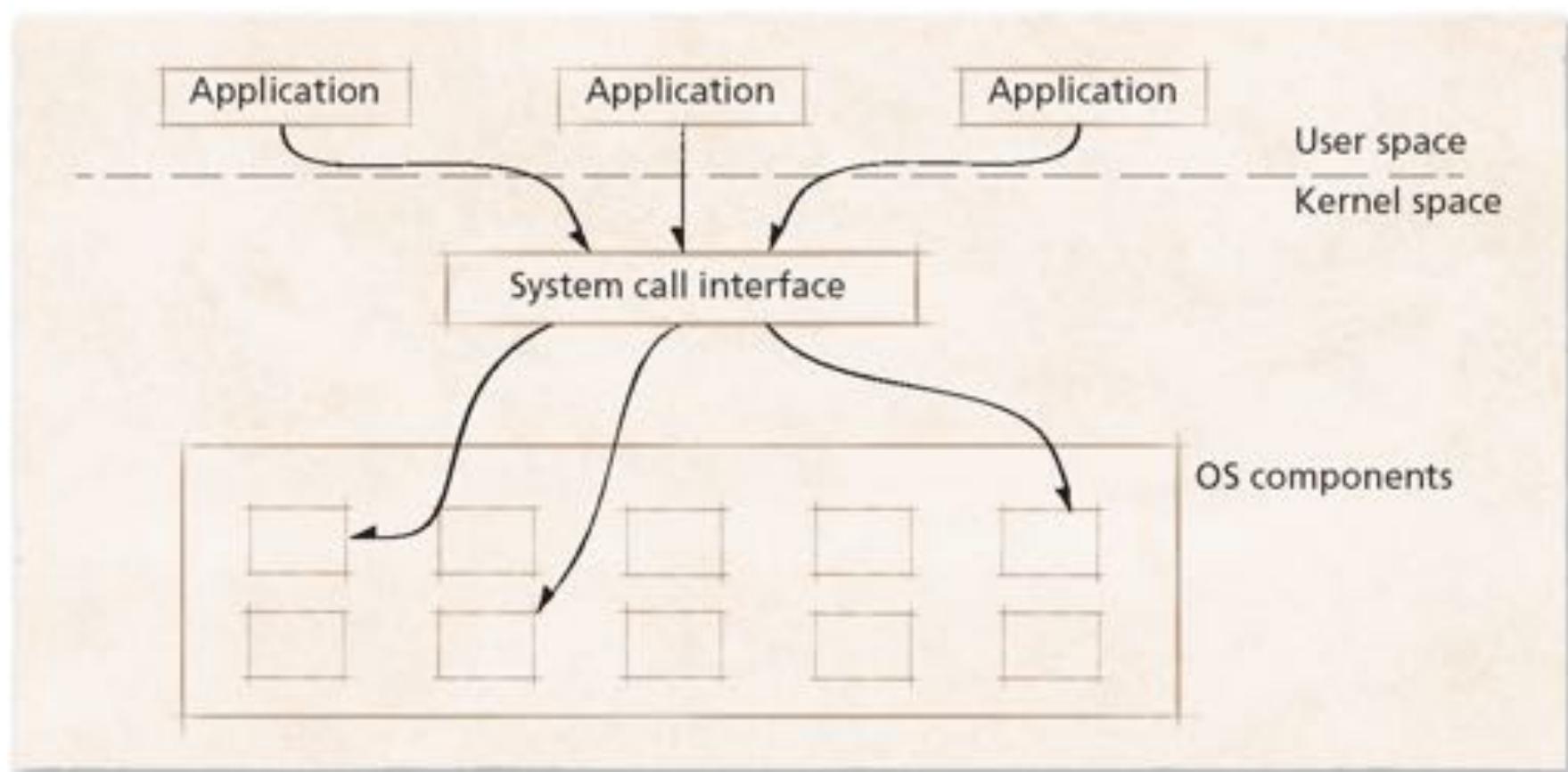
- **Middleware**
 - Links two separate applications
 - Often over a network and between incompatible machines
 - Particularly important for Web services
 - Simplifies communication across multiple architectures
- **Web services**
 - Encompass set of related standards
 - Ready-to-use pieces of software on the Internet
 - Enable any two applications to communicate and exchange data

1.10 Application Bases

- IBM PC immediately spawned a huge software industry
 - Independent software vendors (ISVs) market software packages to run under MS-DOS operating system.
 - Operating system must present environment conducive to rapid and easy application development
 - Otherwise unlikely to be adopted widely
- Application base
 - Combination of hardware and operating system used to develop applications
 - Developers and users unwilling to abandon established application base
 - Increased financial cost and time spent relearning

1.10 Application Bases

Figure 1.1 Interaction between applications and the operating system.



1.11 Operating System Environments

- Operating systems intended for high-end environments
 - Special design requirements and hardware support needs
 - Large main memory
 - Special-purpose hardware
 - Large numbers of processes
- Embedded systems
 - Characterized by small set of specialized resources
 - Provide functionality to devices such as cell phones and PDAs
 - Efficient resource management key to building successful operating system

1.11 Operating System Environments

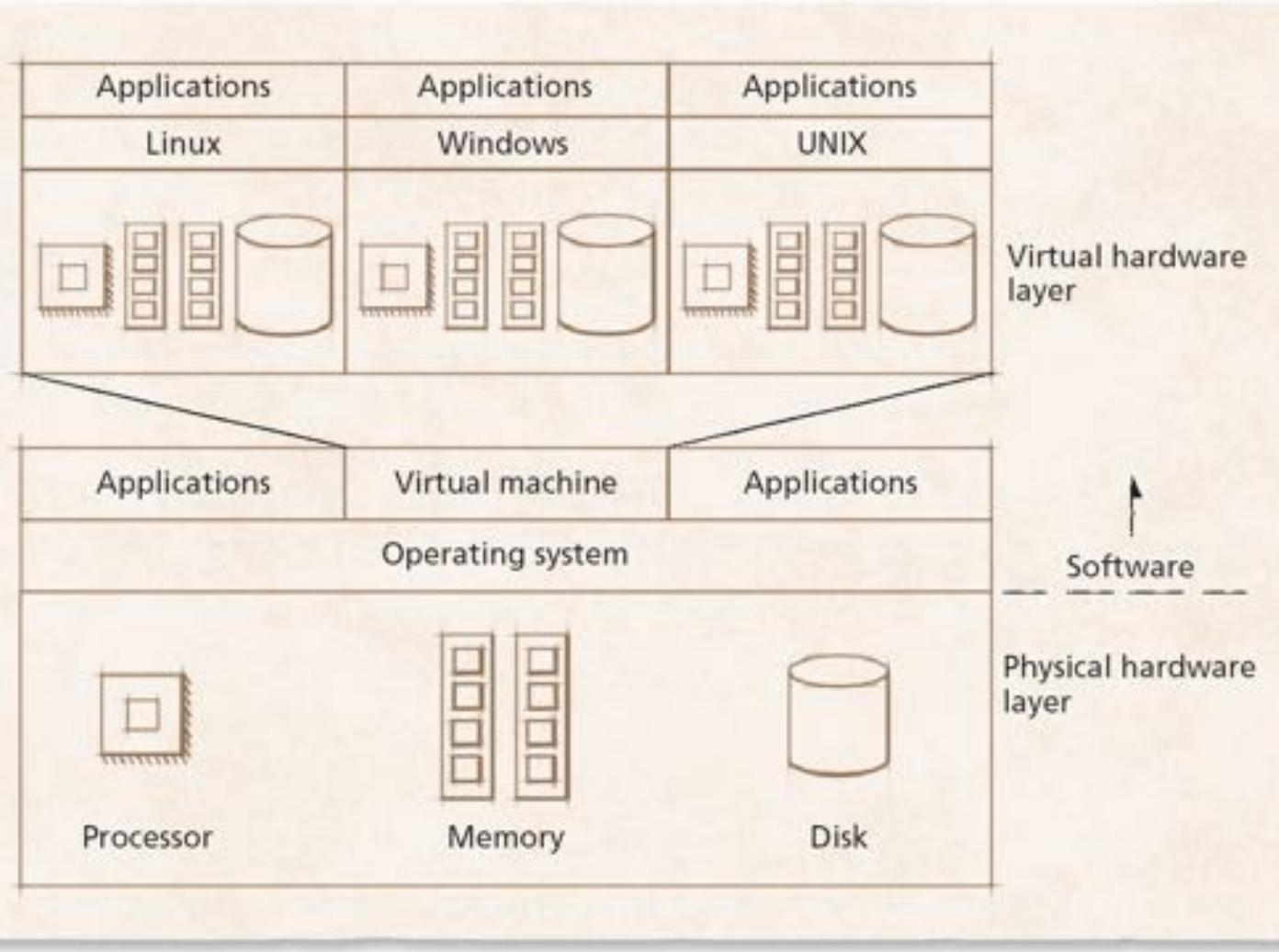
- Real-time systems
 - Require that tasks be performed within particular (often short) time frame
 - Autopilot feature of an aircraft must constantly adjust speed, altitude and direction
 - Such actions cannot wait indefinitely—and sometimes cannot wait at all

1.11 Operating System Environments

- Virtual machines (VMs)
 - Software abstraction of a computer
 - Often executes on top of native operating system
- Virtual machine operating system
 - Manages resources provided by virtual machine
- Applications of virtual machines
 - Allow multiple instances of an operating system to execute concurrently
 - Emulation
 - Software or hardware mimics functionality of hardware or software not present in system
 - Promote portability

1.11 Operating System Environments

Figure 1.2 Schematic of a virtual machine.



1.12 Operating System Components and Goals

- Computer systems have evolved
 - Early systems contained no operating system,
 - Later gained multiprogramming and timesharing machines
 - Personal computers and finally truly distributed systems
 - Filled new roles as demand changed and grew

1.12.1 Core Operating System Components

- User interaction with operating system
 - Often, through special application called a shell
 - Kernel
 - Software that contains core components of operating system
- Typical operating system components include:
 - Processor scheduler
 - Memory manager
 - I/O manager
 - Interprocess communication (IPC) manager
 - File system manager

1.12.1 Core Operating System Components

- Multiprogrammed environments now common
 - Kernel manages the execution of processes
 - Program components which execute independently but use single memory space to share data are called threads.
 - To access I/O device, process must issue system call
 - Handled by device driver
 - Software component that interacts directly with hardware
 - Often contains device-specific commands

1.12.2 Operating System Goals

- Users expect certain properties of operating systems
 - Efficiency
 - Robustness
 - Scalability
 - Extensibility
 - Portability
 - Security
 - Protection
 - Interactivity
 - Usability

1.13 Operating System Architectures

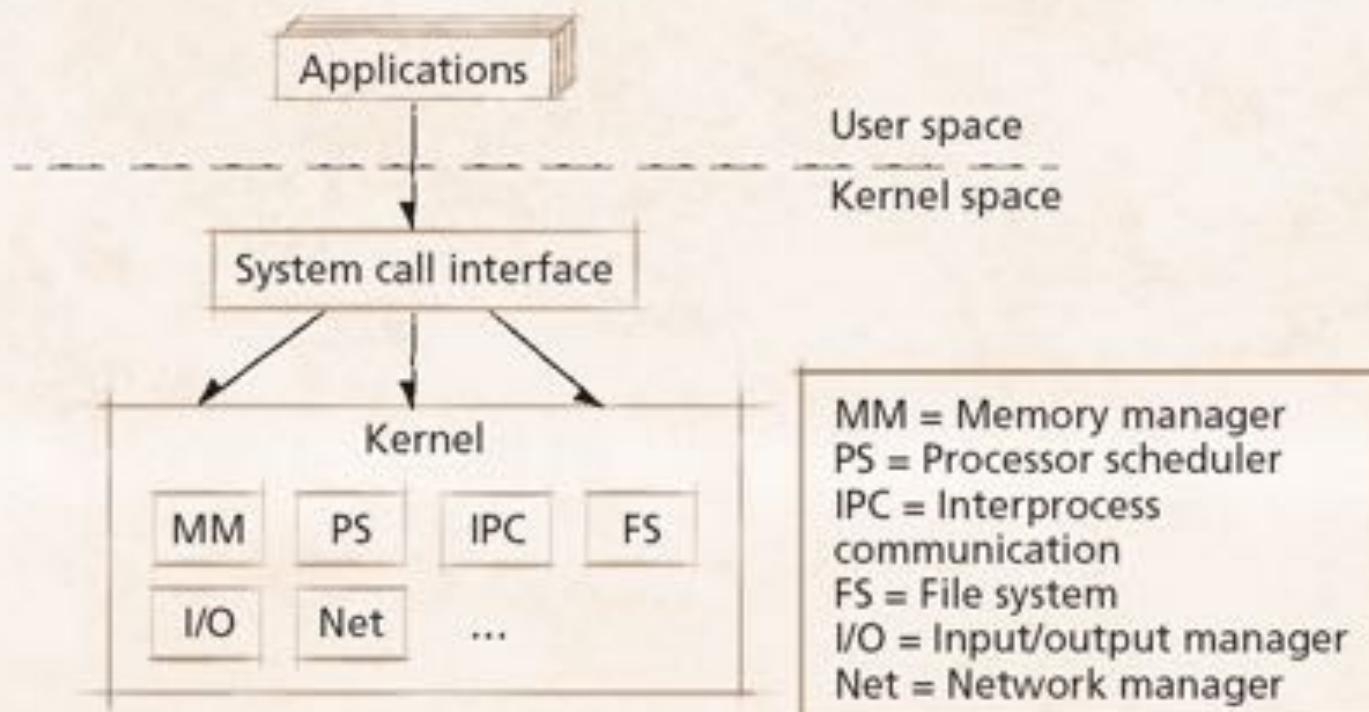
- Today's operating systems tend to be complex
 - Provide many services
 - Support variety of hardware and software
 - Operating system architectures help manage this complexity
 - Organize operating system components
 - Specify privilege with which each component executes

1.13.1 Monolithic Architecture

- Monolithic operating system
 - Every component contained in kernel
 - Any component can directly communicate with any other
 - Tend to be highly efficient
 - Disadvantage is difficulty determining source of subtle errors

1.13.1 Monolithic Architecture

Figure 1.3 Monolithic operating system kernel architecture.

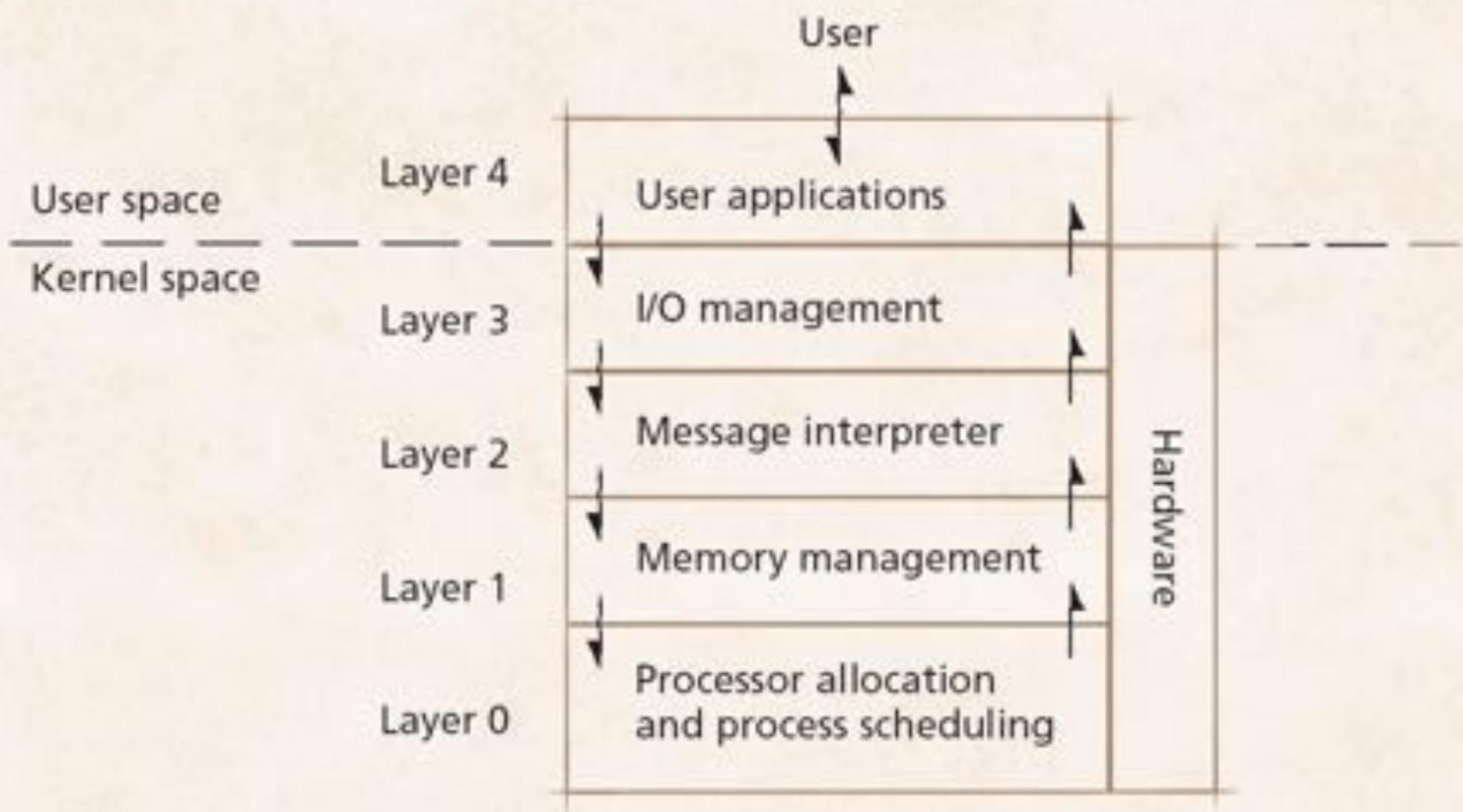


1.13.2 Layered Architecture

- Layered approach to operating systems
 - Tries to improve on monolithic kernel designs
 - Groups components that perform similar functions into layers
 - Each layer communicates only with layers immediately above and below it
 - Processes' requests might pass through many layers before completion
 - System throughput can be less than monolithic kernels
 - Additional methods must be invoked to pass data and control

1.13.2 Layered Architecture

Figure 1.4 Layers of the THE operating system.

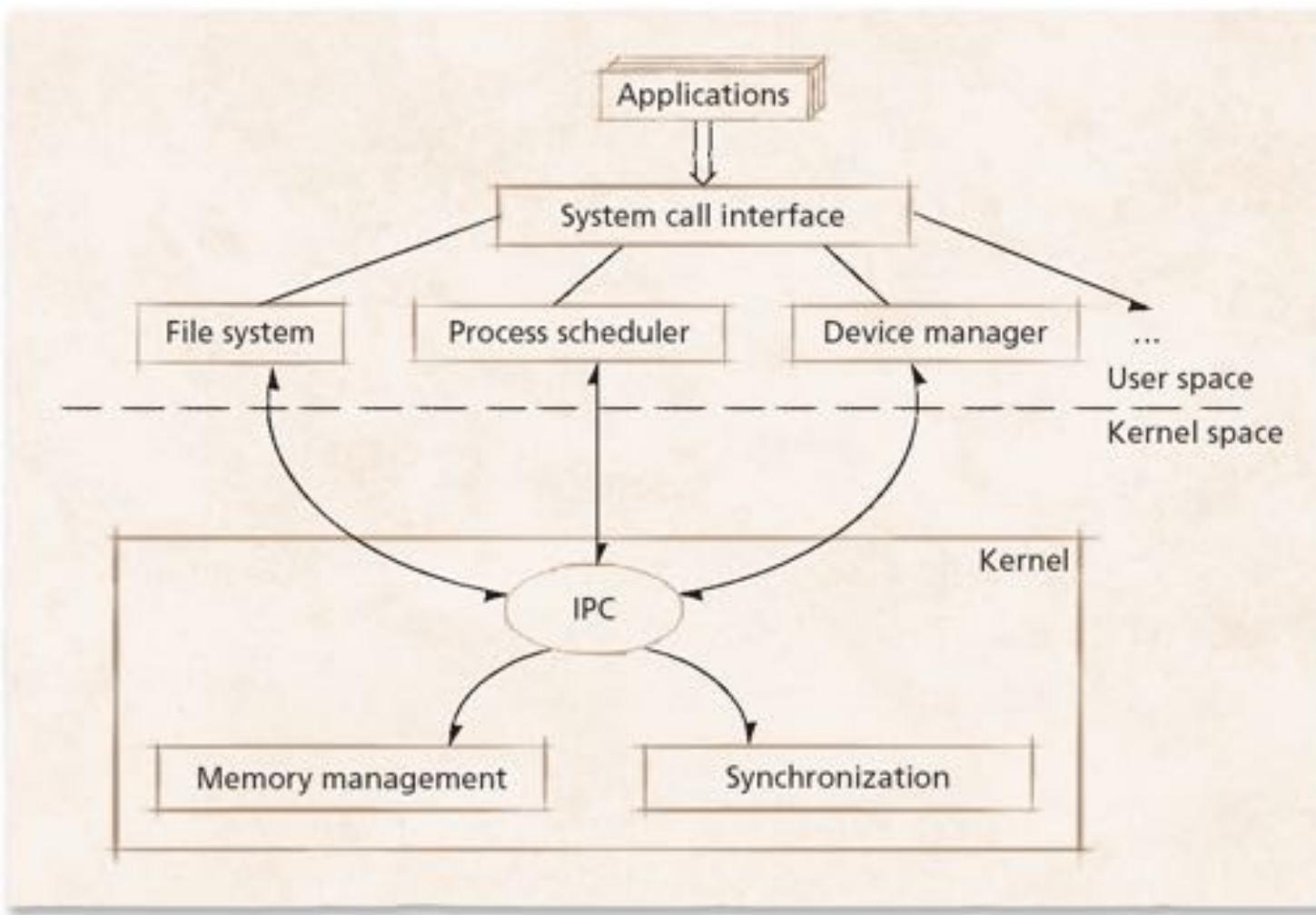


1.13.3 Microkernel Architecture

- Microkernel operating system architecture
 - Provides only small number of services
 - Attempt to keep kernel small and scalable
 - High degree of modularity
 - Extensible, portable and scalable
 - Increased level of intermodule communication
 - Can degrade system performance

1.13.3 Microkernel Architecture

Figure 1.5 Microkernel operating system architecture.

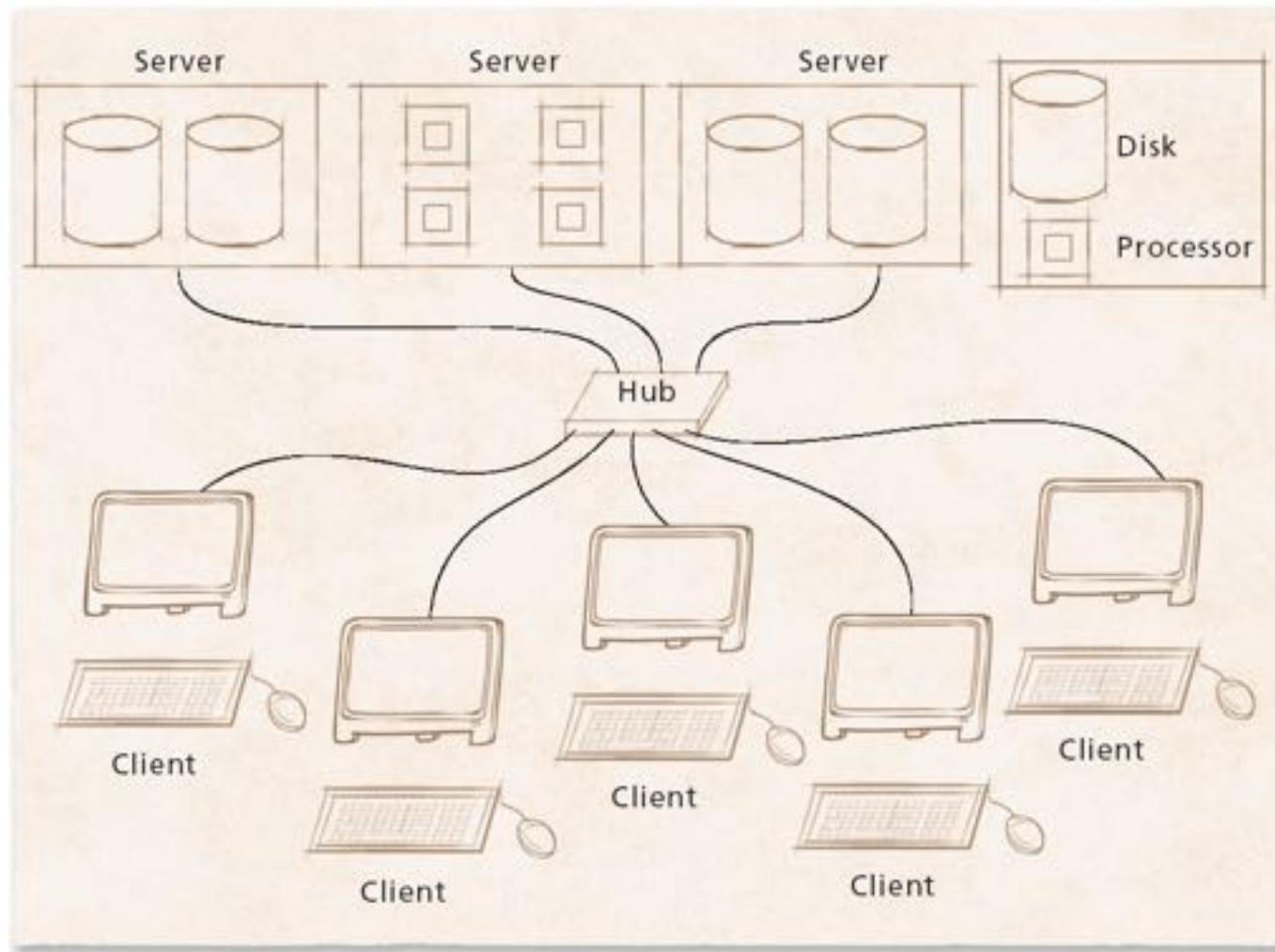


1.13.4 Networked and Distributed Operating Systems

- Network operating system
 - Runs on one computer
 - Allows its processes to access resources on remote computers
- Distributed operating system
 - Single operating system
 - Manages resources on more than one computer system
 - Goals include:
 - Transparent performance
 - Scalability
 - Fault tolerance
 - Consistency

1.13.4 Networked and Distributed Operating Systems

Figure 1.6 Client/server networked operating system model.



Chapter 2 – Hardware and Software Concepts

Outline

- 2.1 **Introduction**
- 2.2 **Evolution of Hardware Devices**
- 2.3 **Hardware Components**
 - 2.3.1 **Mainboards**
 - 2.3.2 **Processors**
 - 2.3.3 **Clocks**
 - 2.3.4 **Memory Hierarchy**
 - 2.3.5 **Main Memory**
 - 2.3.6 **Secondary Storage**
 - 2.3.7 **Buses**
 - 2.3.8 **Direct Memory Access (DMA)**
 - 2.3.9 **Peripheral Devices**
- 2.4 **Hardware Support for Operating Systems**
 - 2.4.1 **Processor**
 - 2.4.2 **Timers and Clocks**

Chapter 2 – Hardware and Software Concepts

Outline (continued)

- 2.4.3 Bootstrapping
- 2.4.4 Plug and Play
- 2.5 Caching and Buffering
- 2.6 Software overview
 - 2.6.1 Machine Language and Assembly Language
 - 2.6.2 Interpreters and Compilers
 - 2.6.3 High-Level Languages
 - 2.6.4 Structured Programming
 - 2.6.5 Object-Oriented Programming
- 2.7 Application Programming Interfaces (APIs)
- 2.8 Compiling, Linking and Loading
 - 2.8.1 Compiling
 - 2.8.2 Linking
 - 2.8.3 Loading
- 2.9 Firmware
- 2.10 Middleware

Objectives

- After reading this chapter, you should understand:
 - hardware components that must be managed by an operating system.
 - how hardware has evolved to support operating system functions.
 - how to optimize performance of various hardware devices.
 - the notion of an application programming interface (API).
 - the process of compilation, linking and loading.

2.1 Introduction

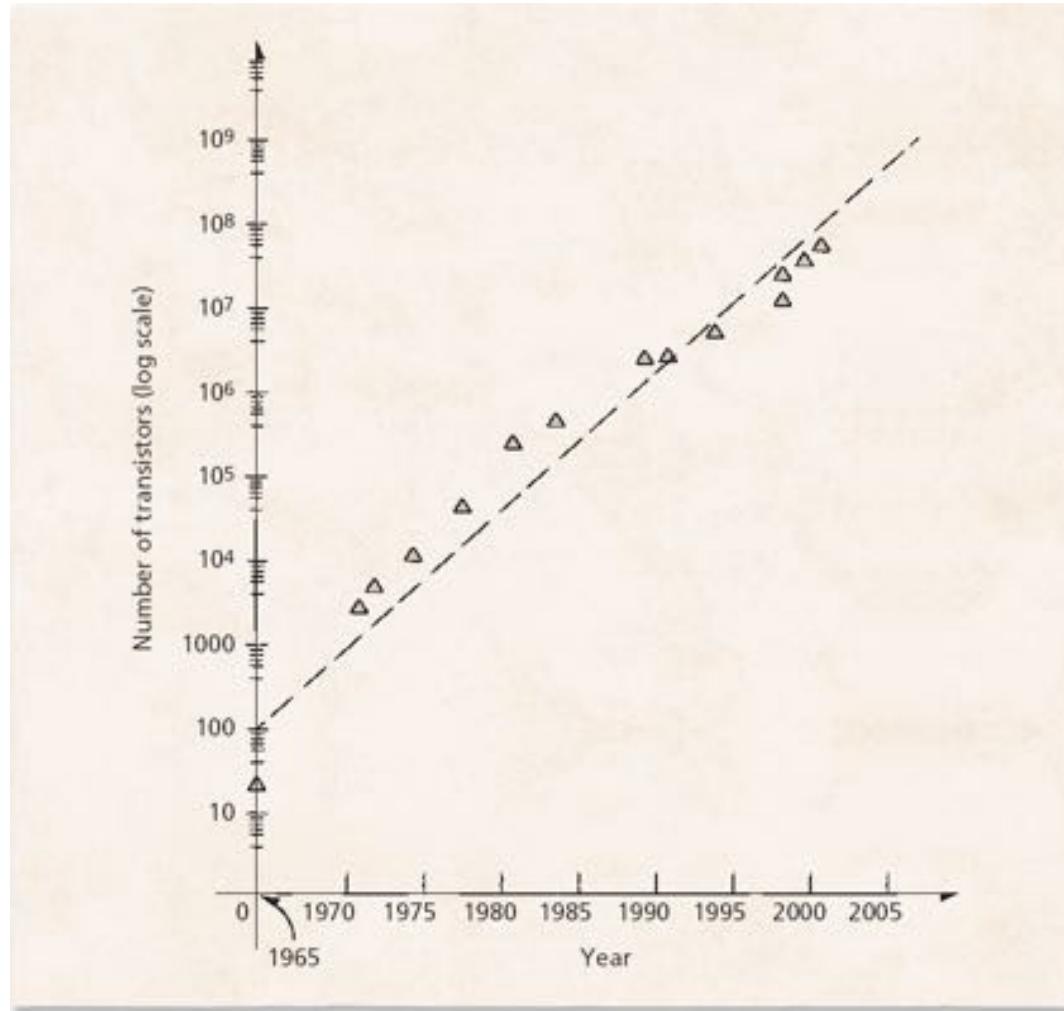
- An operating system is primarily a resource manager
 - Design is tied to the hardware and software resources the operating system must manage
 - processors
 - memory
 - secondary storage (such as hard disks)
 - other I/O devices
 - processes
 - threads
 - files
 - databases

2.2 Evolution of Hardware Devices

- Most operating systems are independent of hardware configurations
 - Operating systems use device drivers to perform device-specific I/O operations
 - For example, plug-and-play devices when connected instruct the operating system on which driver to use without user interaction

2.2 Evolution of Hardware Devices

Figure 2.1 Transistor count plotted against time for Intel processors.



2.3 Hardware Components

- A computer's hardware consists of:
 - processor(s)
 - main memory
 - input/output devices

2.3.1 Mainboards

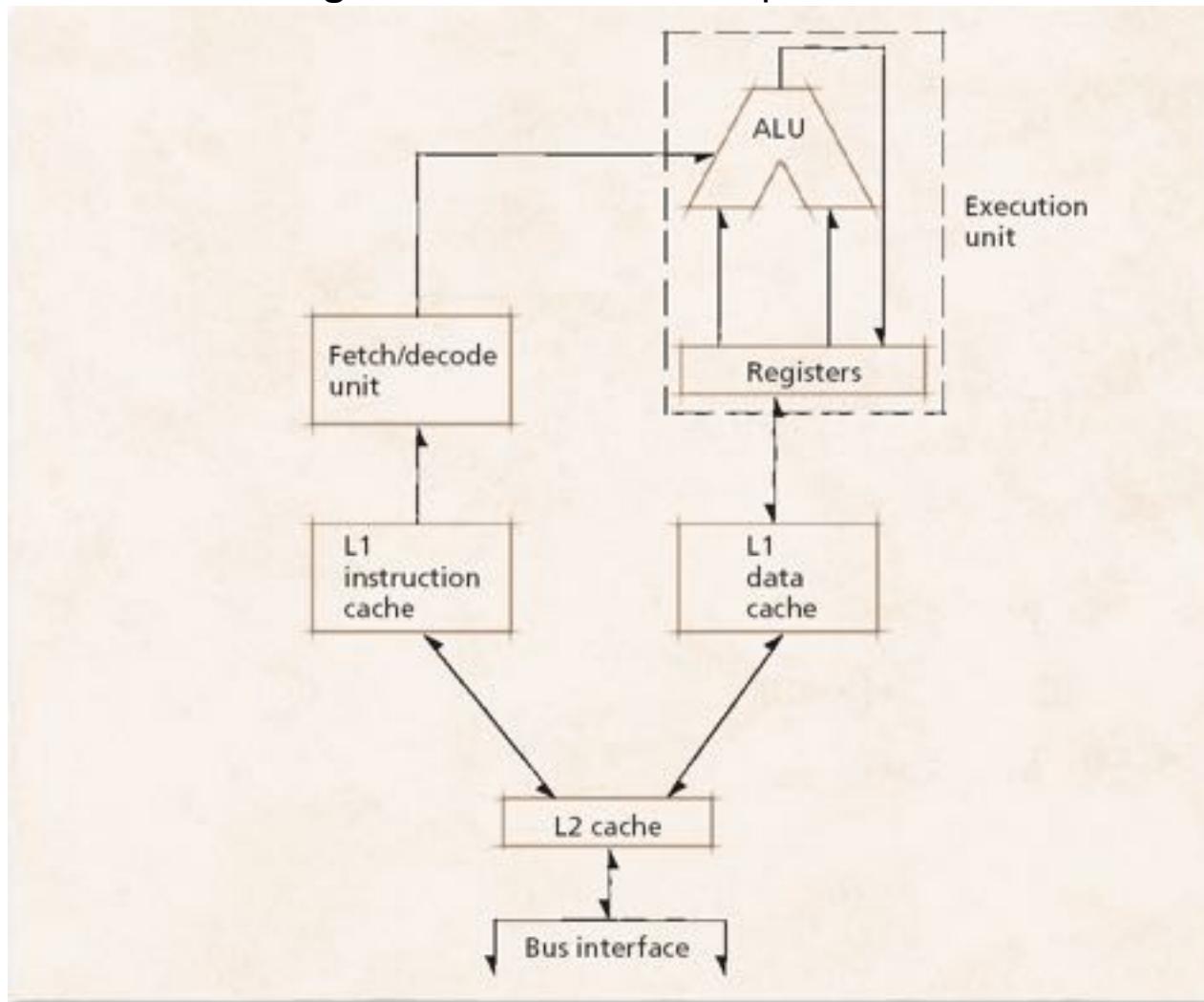
- Printed Circuit Board
 - Hardware component that provides electrical connections between devices
 - The mainboard is the central PCB in a system
 - Devices such as processors and main memory are attached
 - Include chips to perform low-level operations (e.g., BIOS)

2.3.2 Processors

- A processor is hardware that executes machine-language
 - CPU executes the instructions of a program
 - Coprocessor executes special-purpose instructions
 - Ex., graphics or audio coprocessors
 - Registers are high-speed memory located on processors
 - Data must be in registers before a processor can operate on it
 - Instruction length is the size of a machine-language instruction
 - Some processors support multiple instruction lengths

2.3.2 Processors

Figure 2.2 Processor components.



2.3.3 Clocks

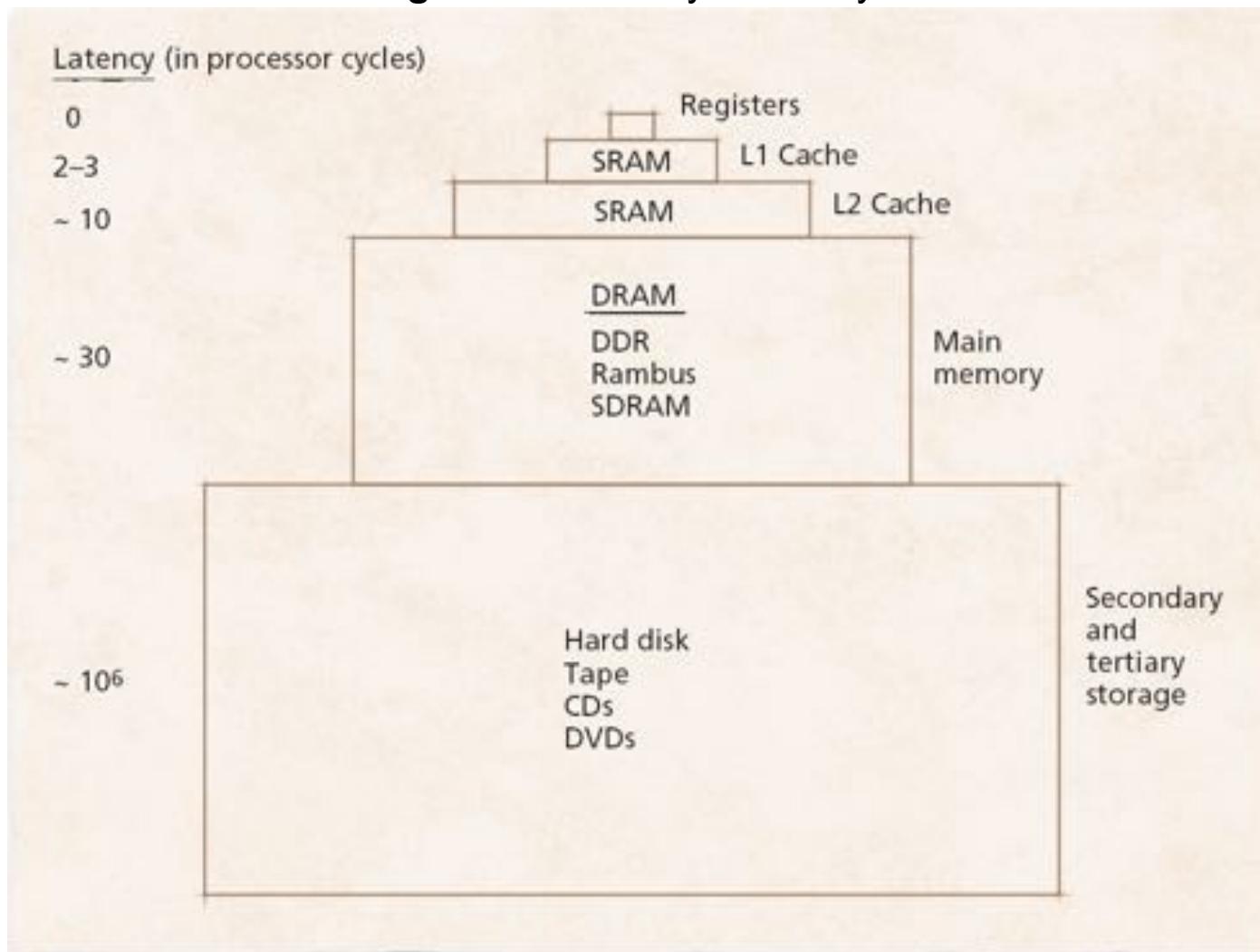
- Computer time is measured in cycles
 - One complete oscillation of an electrical signal
 - Provided by system clock generator
 - Processor speeds are measured in GHz (billions of cycles per second)
 - Modern desktops execute at hundreds of megahertz or several GHz

2.3.4 Memory Hierarchy

- The memory hierarchy is a scheme for categorizing memory
 - Fastest and most expensive at the top, slowest and least expensive at the bottom
 - Registers
 - L1 Cache
 - L2 Cache
 - Main Memory
 - Secondary and tertiary storage (CDs, DVDs and floppy disks)
 - Main memory is the lowest data referenced directly by processor
 - Volatile – loses its contents when the system loses power

2.3.4 Memory Hierarchy

Figure 2.3 Memory hierarchy.



2.3.5 Main Memory

- Main memory consists of volatile random access memory (RAM)
 - Processes can access data locations in any order
 - Common forms of RAM include:
 - dynamic RAM (DRAM) – requires refresh circuit
 - static RAM (SRAM) – does not require refresh circuit
 - Bandwidth is the amount of data that can be transferred per unit of time

2.3.6 Secondary Storage

- Secondary storage stores large quantities of persistent data at low cost
 - Accessing data on a hard disk is slower than main memory
 - Mechanical movement of read/write head
 - Rotational latency
 - Transfer time
 - Removable secondary storage facilitates data backup and transfer
 - CDs (CD-R, CD-RW)
 - DVDs (DVD-R, DVD+R)
 - Zip disks
 - Floppy disks
 - Flash memory cards
 - Tapes

2.3.7 Buses

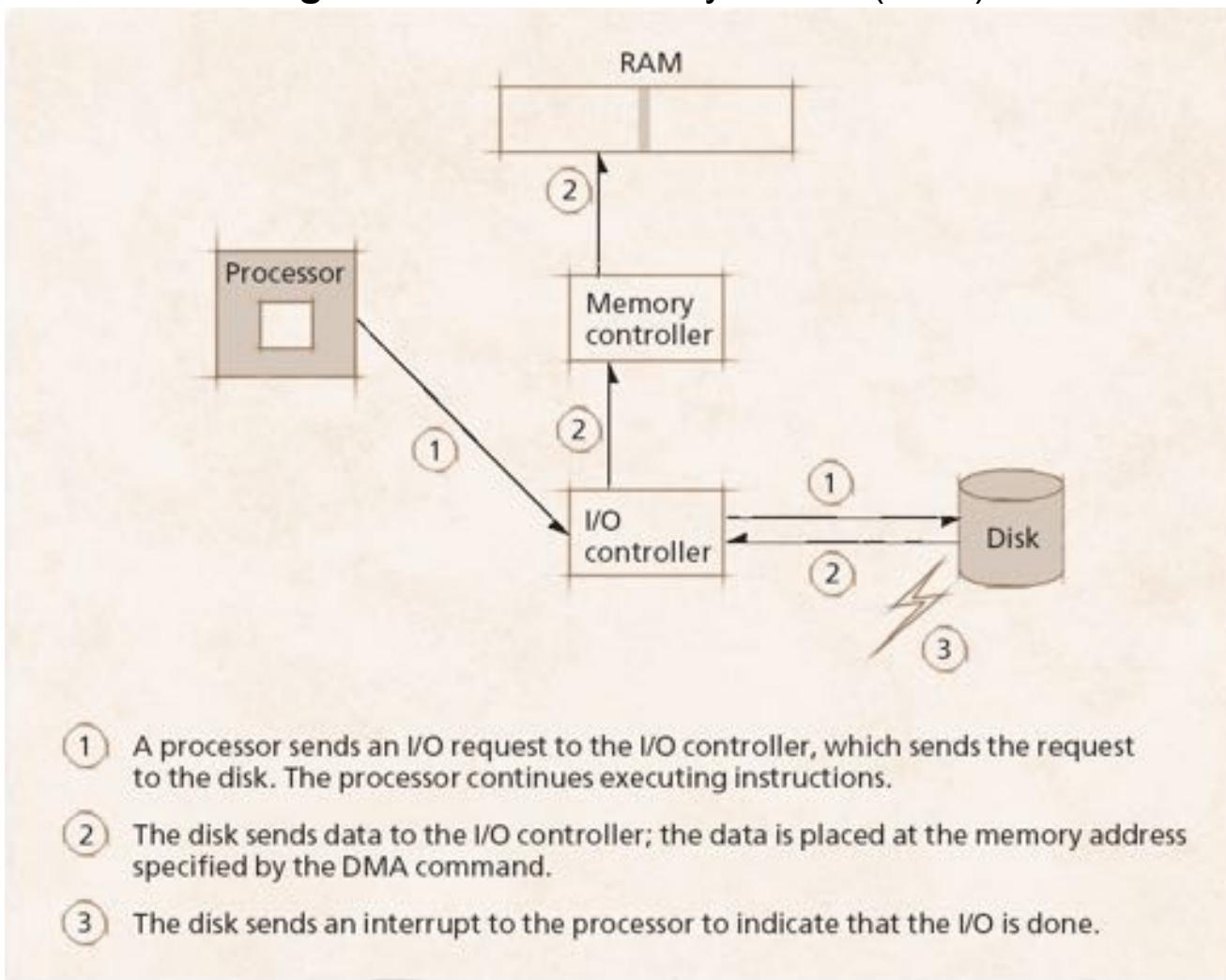
- A bus is a collection of traces
 - Traces are thin electrical connections that transport information between hardware devices
 - A port is a bus that connects exactly two devices
 - An I/O channel is a bus shared by several devices to perform I/O operations
 - Handle I/O independently of the system's main processors
 - Example, the frontside bus (FSB) connects a processor to main memory

2.3.8 Direct Memory Access (DMA)

- DMA improves data transfer between memory and I/O devices
 - Devices and controllers transfer data to and from main memory directly
 - Processor is free to execute software instructions
 - DMA channel uses an I/O controller to manage data transfer
 - Notifies processor when I/O operation is complete
 - Improves performance in systems that perform large numbers of I/O operations (e.g., mainframes and servers)

2.3.8 Direct Memory Access (DMA)

Figure 2.4 Direct memory access (DMA).



2.3.9 Peripheral Devices

- Peripheral devices
 - Any device not required for a computer to execute software instructions
 - Internal devices are referred to as integrated peripheral devices
 - Network interface cards, modems, sound cards
 - Hard disk, CD and DVD drives
 - Character devices transfer data one bit at a time
 - Keyboards and mice
 - Can be attached to a computer via ports and other buses
 - Serial ports, parallel ports, USB, IEEE 1394 ports and SCSI

2.3.9 Peripheral Devices

Figure 2.5 Peripheral devices (1 of 2).

<i>Device</i>	<i>Description</i>
CD-RW drive	Reads data from, and writes data to, optical disks.
Zip drive	Transfers data to and from a removable, durable magnetic disk.
Floppy drive	Reads data from, and writes data to, removable magnetic disks.
Mouse	Transmits the change in location of a pointer or cursor in a graphical user interface (GUI).
Keyboard	Transmits characters or commands that a user types.
Multifunction printer	Can print, copy, fax and scan documents.
Sound card	Converts digital signals to audio signals for speakers. Also can receive audio signals via a microphone and produce a digital signal.

2.3.9 Peripheral Devices

Figure 2.5 Peripheral devices (2 of 2).

Video accelerator	Displays graphics on the screen; accelerates two- and three-dimensional graphics.
Network card	Sends data to and receives data from other computers.
Digital camera	Records, and often displays, digital images.
Biometric device	Scans human characteristics, such as fingerprints and retinas, typically for identification and authentication purposes.
Infrared device	Communicates data between devices via a line-of-sight wireless connection.
Wireless device	Communicates data between devices via an omnidirectional wireless connection.

2.4 Hardware Support for Operating Systems

- Computer architectures contain:
 - Features that perform operating system functions quickly in hardware to improve performance
 - Features that enable the operating system to rigidly enforce protection

2.4.1 Processor

- A processor implements operating system protection mechanisms
 - Prevents processes from accessing privileged instructions or memory
 - Computer systems generally have several different execution modes:
 - User mode (user state or problem state)
 - User may execute only a subset of instructions
 - Kernel mode (supervisor state)
 - Processor may access privileged instructions and resources on behalf of processes

2.4.1 Processor

- Memory protection and management
 - Prevents processes from accessing memory that has not been assigned to them
 - Implemented using processor registers modified only by privileged instructions
- Interrupts and Exceptions
 - Most devices send a signal called an interrupt to the processor when an event occurs
 - Exceptions are interrupts generated in response to errors
 - The OS can respond to an interrupt by notifying processes that are waiting on such events

2.4.2 Timers and Clocks

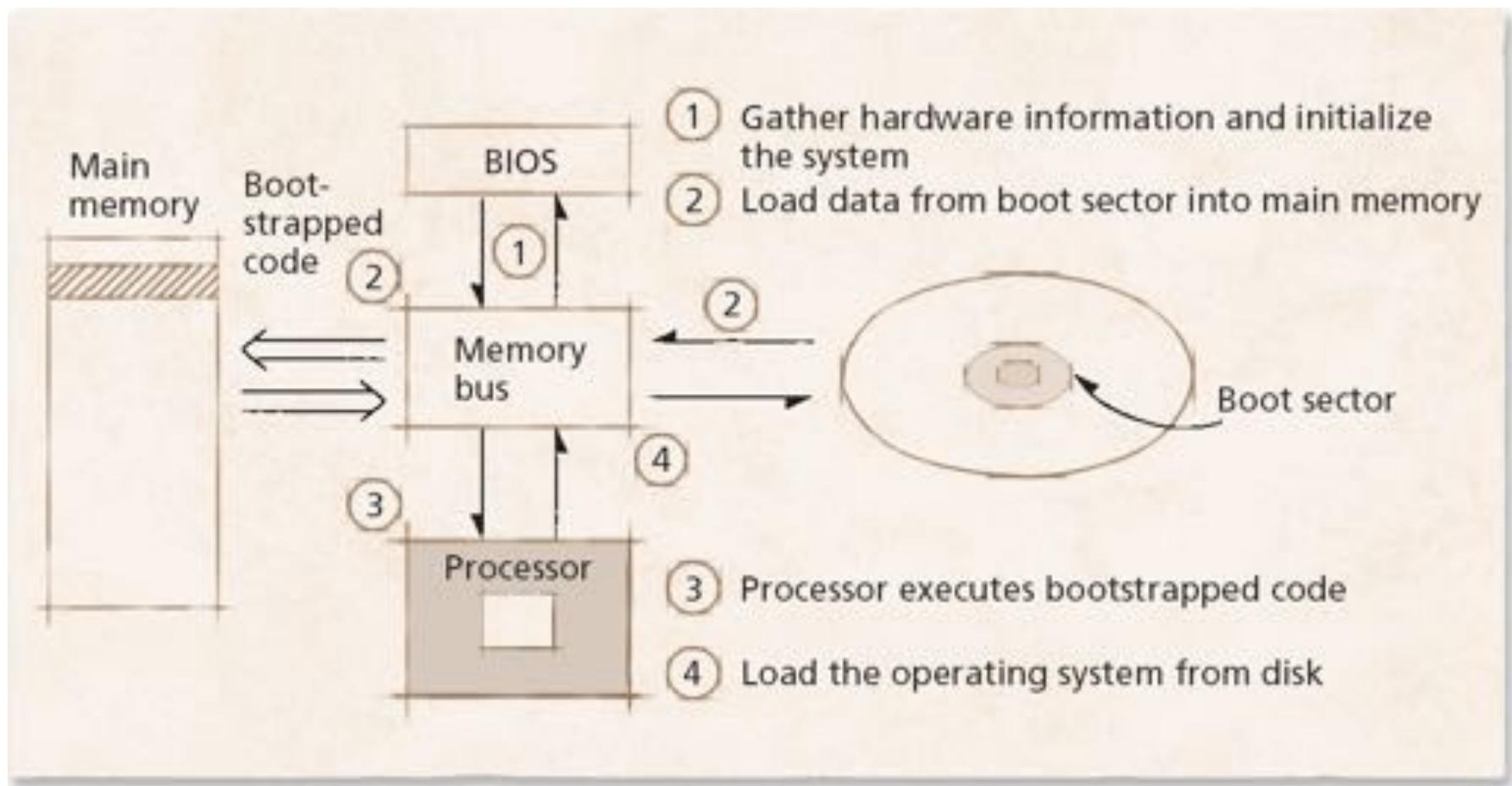
- **Timers**
 - An interval timer periodically generates an interrupt
 - Operating systems use interval timers to prevent processes from monopolizing the processor
- **Clocks**
 - Provide a measure of continuity
 - A time-of-day clock enables an OS to determine the current time and date

2.4.3 Bootstrapping

- Bootstrapping: loading initial OS components into memory
 - Performed by a computer's Basic Input/Output System (BIOS)
 - Initializes system hardware
 - Loads instructions into main memory from a region of secondary storage called the boot sector
 - If the system is not loaded, the user will be unable to access any of the computer's hardware

2.4.3 Bootstrapping

Figure 2.6 Bootstrapping.



2.4.4 Plug and Play

- Plug and Play technology
 - Allows operating systems to configure newly installed hardware without user interaction
 - To support plug and play, a hardware device must:
 - Uniquely identify itself to the operating system
 - Communicate with the OS to indicate the resources and services the device requires to function properly
 - Identify the driver that supports the device and allows software to configure the device (e.g., assign the device to a DMA channel)

2.5 Caching and Buffering

- Caches
 - Relatively fast memory
 - Maintain copies of data that will be accessed soon
 - Increase program execution speed
 - Examples include:
 - L1 and L2 processor caches
 - Main memory can be viewed as a cache for hard disks and other secondary storage devices

2.5 Caching and Buffering

- **Buffers**
 - Temporary storage area that holds data during I/O transfers
 - Primarily used to:
 - Coordinate communications between devices operating at different speeds
 - Store data for asynchronous processing
 - Allow signals to be delivered asynchronously
- **Spooling**
 - Buffering technique in which an intermediate device, such as a disk, is interposed between a process and a low-speed I/O device
 - Allows processes to request operations from a peripheral device without requiring that the device be ready to service the request

2.6 Software Overview

- Programming languages
 - Some are directly understandable by computers, others require translation
 - Classified generally as either:
 - Machine language
 - Assembly language
 - High-level language

2.6.1 Machine Language and Assembly Language

- Machine language
 - Defined by the computer's hardware design
 - Consists of streams of numbers (1s and 0s) that instruct computers how to perform elementary operations
 - A computer can understand only its own machine language
- Assembly language
 - Represents machine-language instructions using English-like abbreviations
 - Assemblers convert assembly language to machine language
 - Speeds programming, reduces potential for bugs

2.6.2 Interpreters and Compilers

- High-level languages
 - Instructions look similar to everyday English
 - Accomplish more substantial tasks with fewer statements
 - Require compilers and interpreters
- Compiler
 - Translator program that converts high-level language programs into machine language
- Interpreter
 - Program that directly executes source code or code that has been reduced to a low-level language that is not machine code

2.6.3 High-level languages

- Popular high-level languages
 - Typically are procedural or object-oriented
 - Fortran
 - Used for scientific and engineering applications
 - COBOL
 - For business applications that manipulate large volumes of data
 - C
 - Development language of the UNIX OS
 - C++/Java
 - Popular object-oriented languages
 - C#
 - Object-oriented development language for the .NET platform

2.6.4 Structured programming

- Disciplined approach to creating programs
 - Programs are clear, provably correct and easy to modify
 - Structured programming languages include:
 - Pascal
 - Designed for teaching structured programming
 - Ada
 - Developed by the US Department of Defense
 - Fortran

2.6.5 Object-Oriented Programming

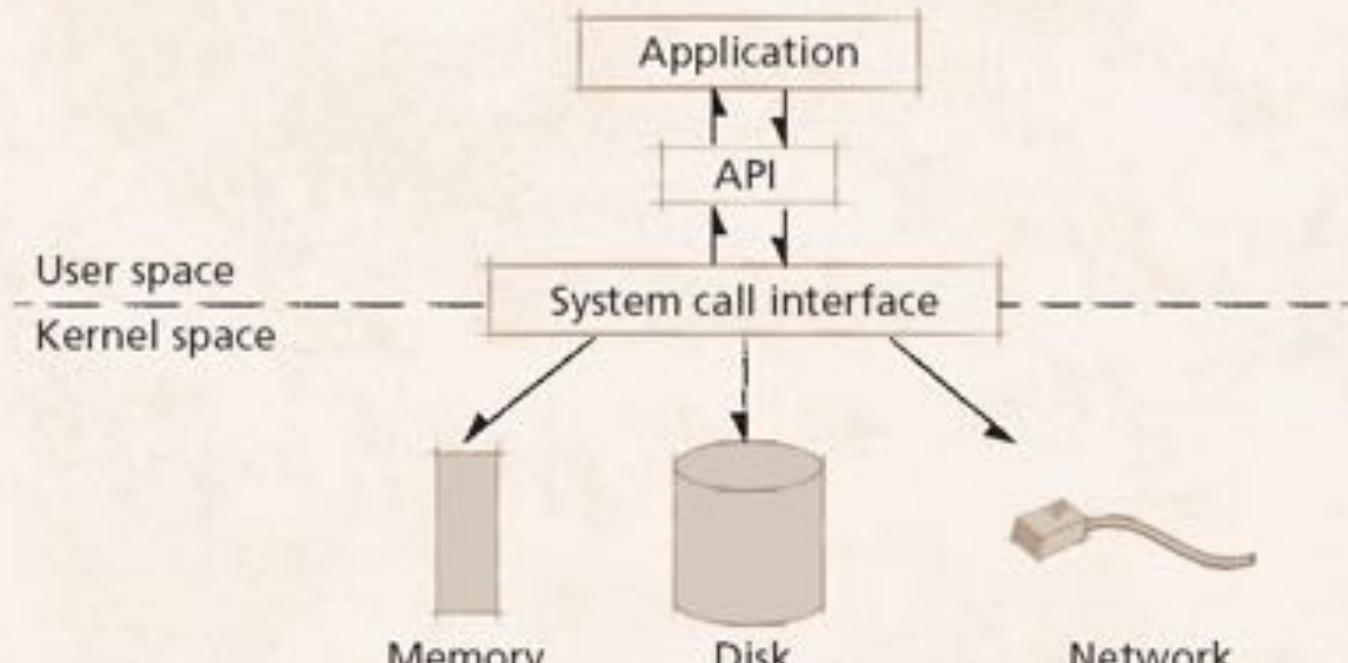
- **Objects**
 - Reusable software unit (any noun can be represented)
 - Easy to modify and understand
 - Have properties (e.g., color) and perform actions (e.g., moving)
- **Classes**
 - Types of related objects
 - Specify the general format of an object and the properties and actions available to it
- **Object-oriented programming**
 - Focuses on behaviors and interactions, not implementation
 - C++, Java and C# are popular object-oriented languages

2.7 Application Programming Interfaces (APIs)

- A set of routines
 - Programmers use routines to request services from the operating system
 - Programs call API functions, which may access the OS by making system calls
 - Examples of APIs include:
 - Portable Operating System Interface (POSIX) standard
 - Windows API

2.7 Application Programming Interfaces (APIs)

Figure 2.7 Application programming interface (API).



2.8 Compiling, Linking and Loading

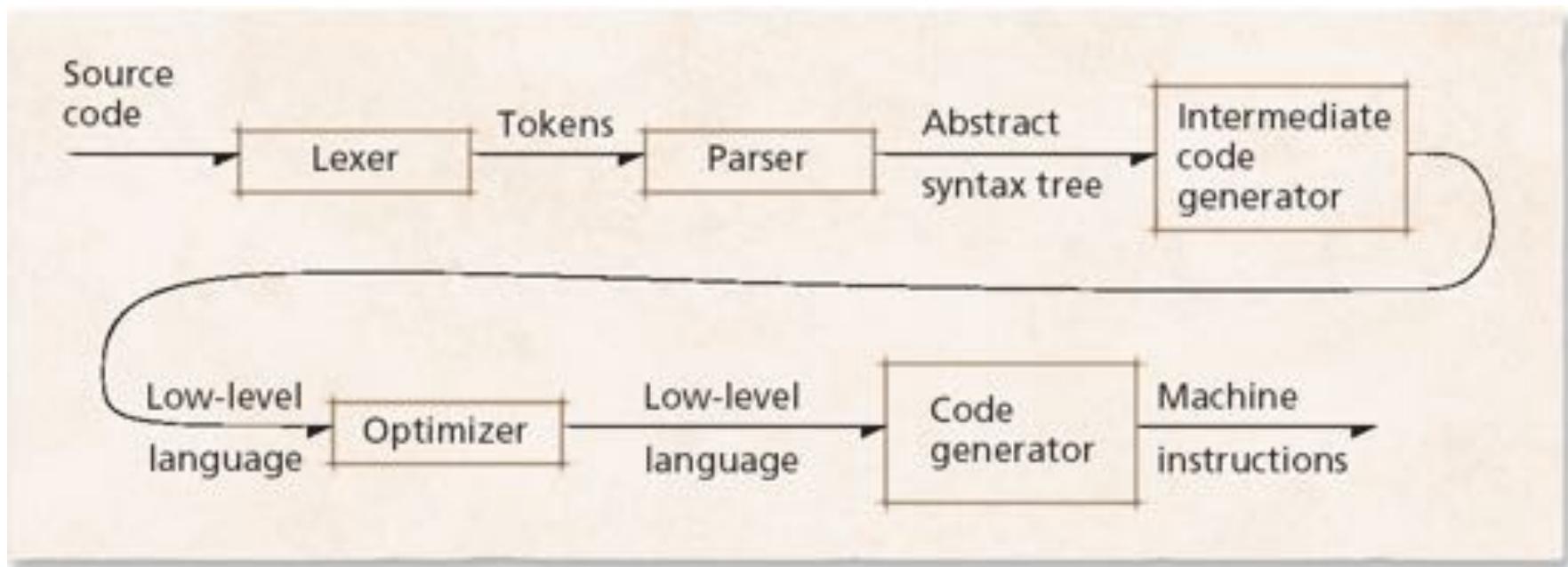
- Before a high-level-language program can execute, it must be:
 - Translated into machine language
 - Linked with various other machine-language programs on which it depends
 - Loaded into memory

2.8.1 Compiling

- Translating high-level code to machine code
 - Accepts source code as input and returns object code
 - Compilation phases include:
 - Lexer
 - Separates the characters of a program's source into tokens
 - Parser
 - Groups tokens into syntactically correct statements
 - Intermediate code generator
 - Converts statements into a stream of simple instructions
 - Optimizer
 - Improves code execution efficiency and memory requirements
 - Code generator
 - Produces the object file containing the machine-language

2.8.1 Compiling

Figure 2.8 Compiler phases.

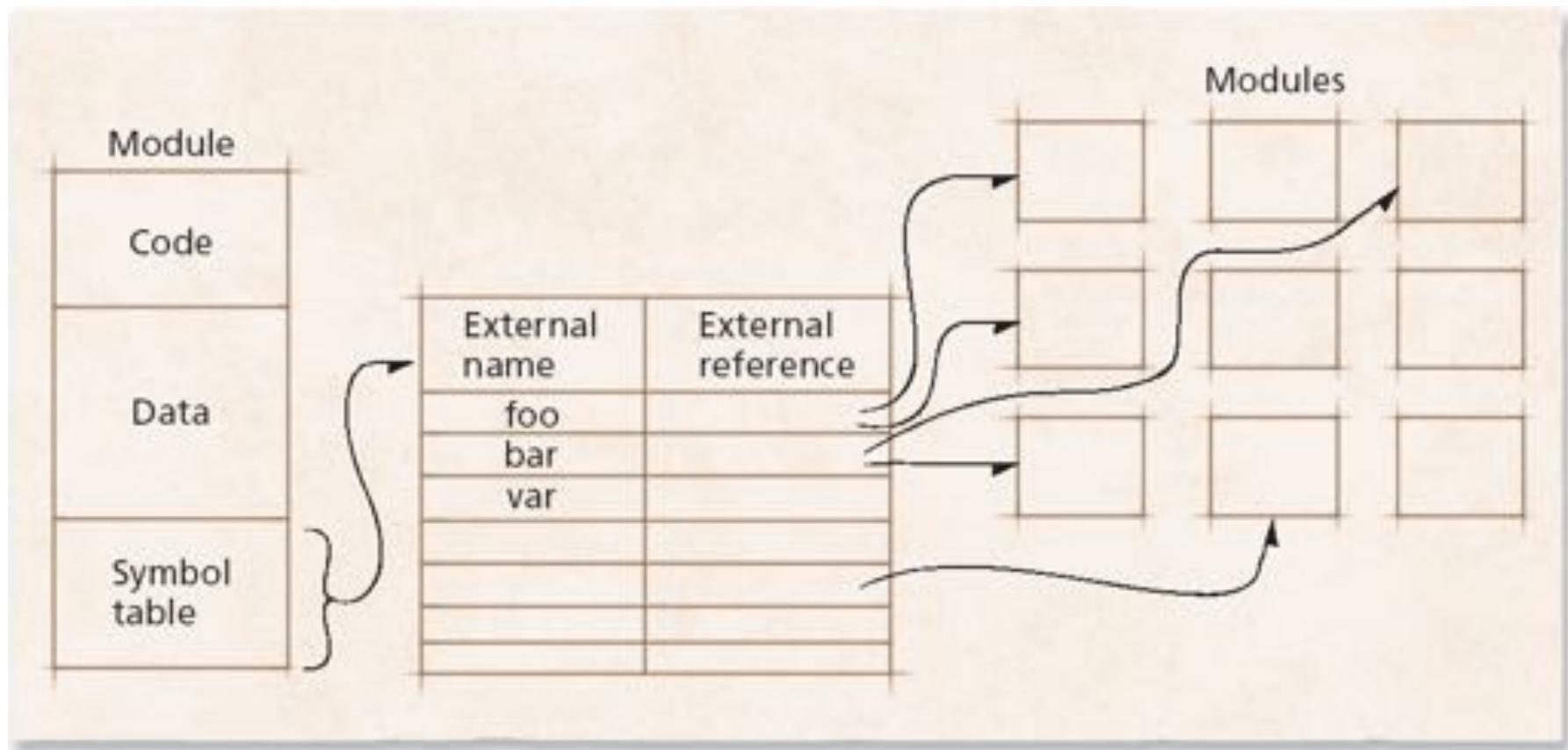


2.8.2 Linking

- **Linkers**
 - Create a single executable unit
 - Integrate precompiled modules called libraries referenced by a program
 - Assign relative addresses to different program or data units
 - Resolve all external references between subprograms
 - Produce an integrated module called a load module
 - Linking can be performed at compile time, before loading, at load time or at runtime

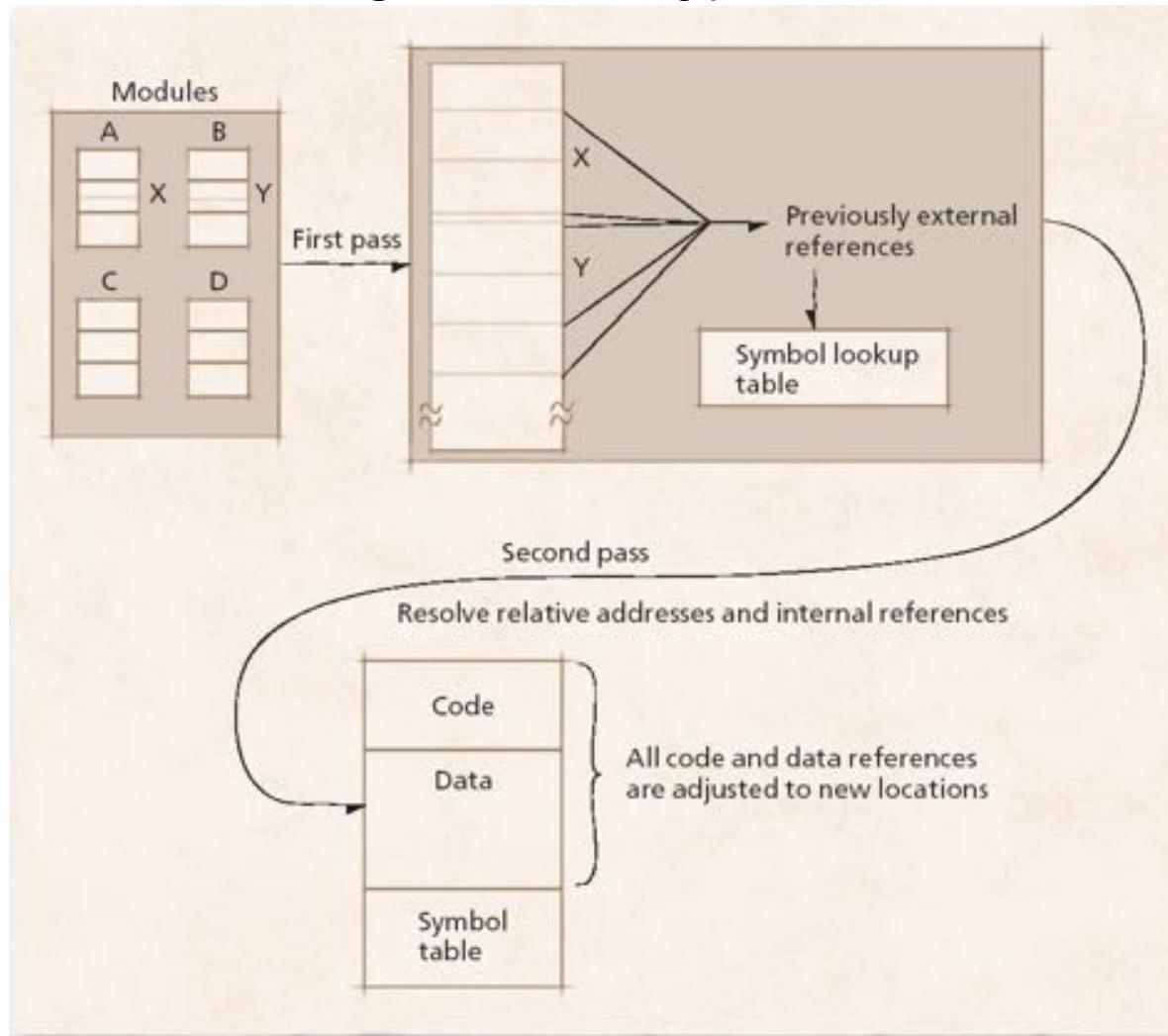
2.8.2 Linking

Figure 2.9 Object module.



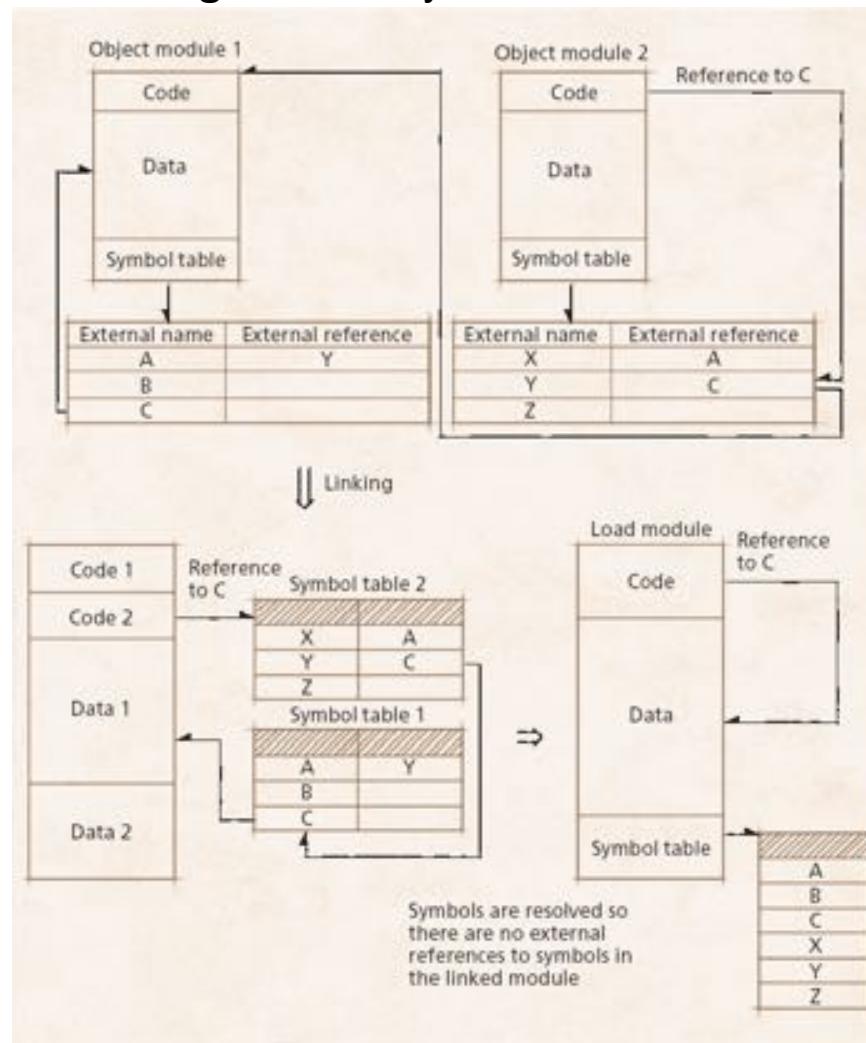
2.8.2 Linking

Figure 2.10 Linking process.



2.8.2 Linking

Figure 2.11 Symbol resolution.

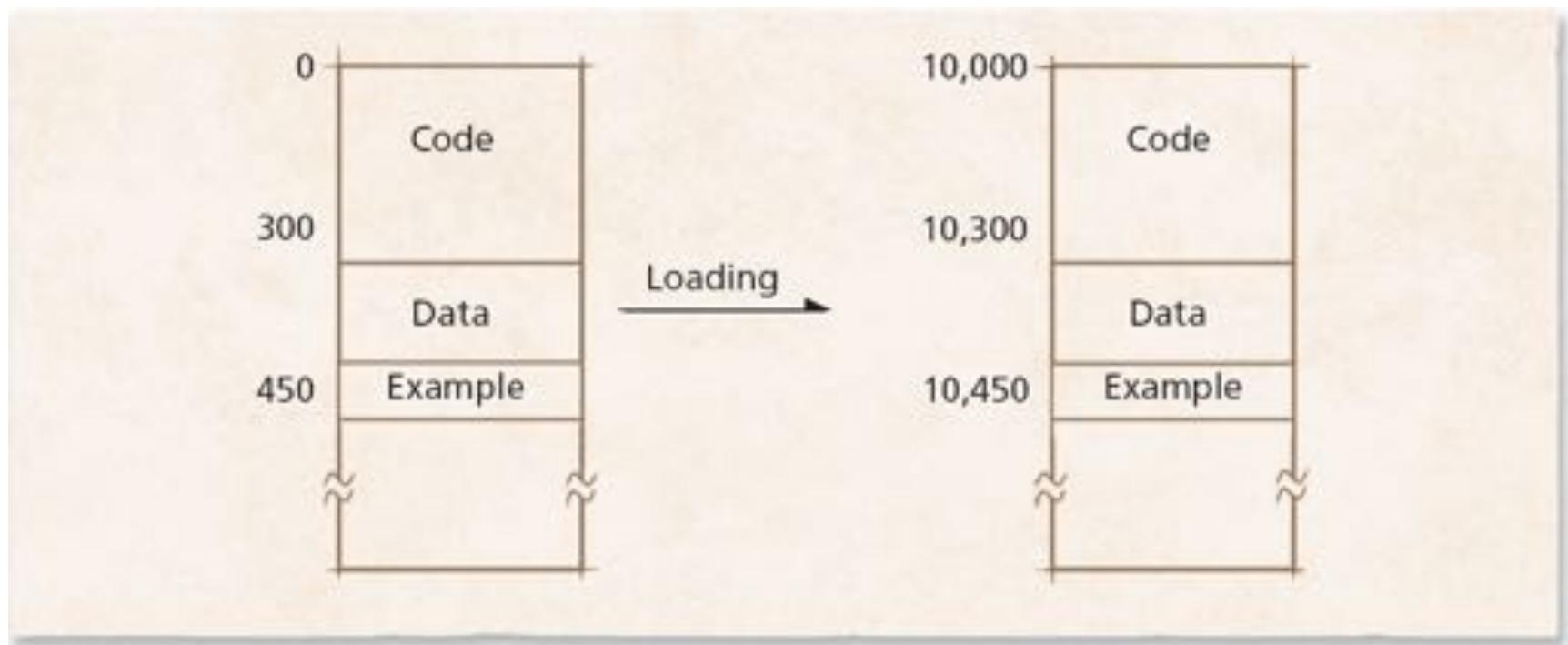


2.8.3 Loading

- Loaders
 - Convert relative addresses to physical addresses
 - Place each instruction and data unit in main memory
- Techniques for loading a program into memory
 - Absolute loading
 - Place program at the addresses specified by programmer or compiler (assuming addresses are available)
 - Relocatable loading
 - Relocate the program's addresses to correspond to its actual location in memory
 - Dynamic loading
 - Load program modules upon first use

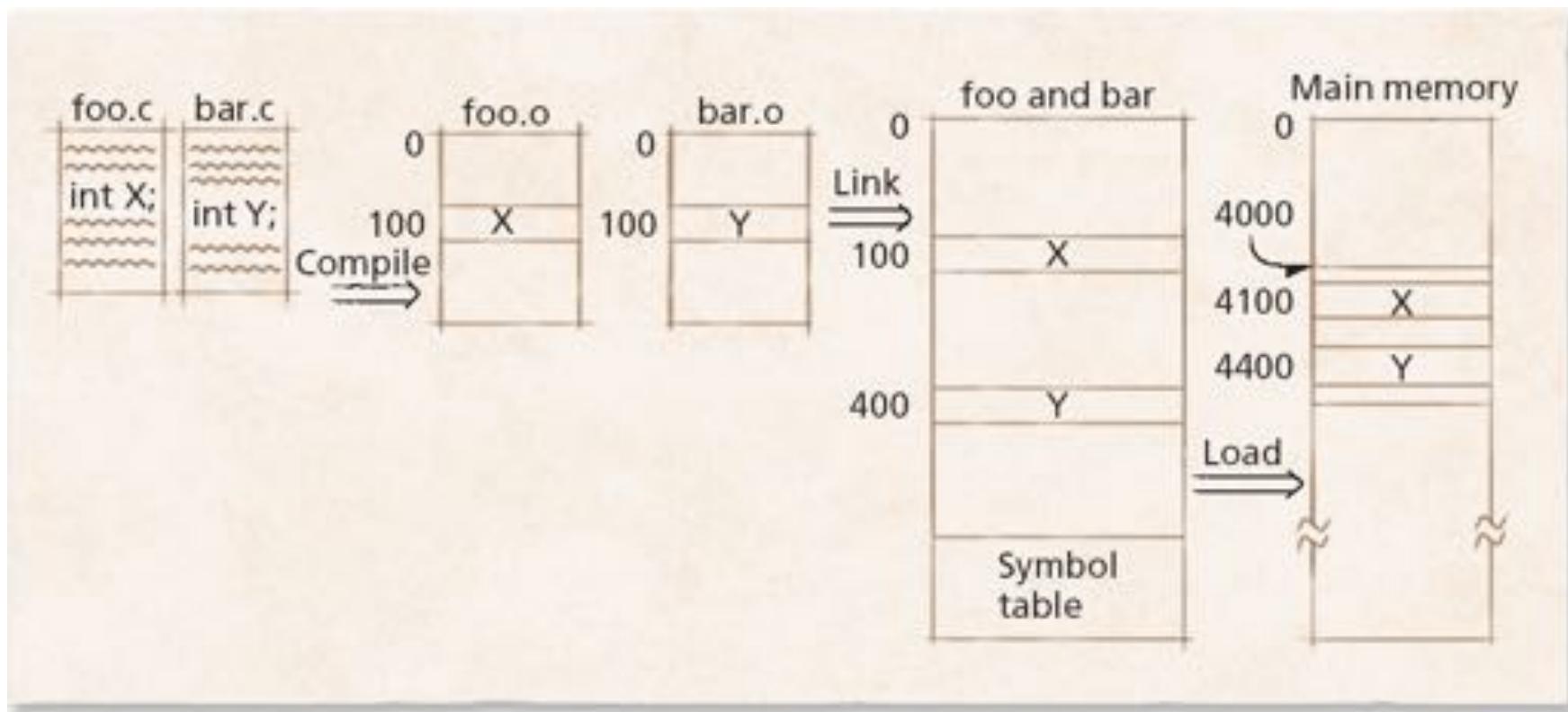
2.8.3 Loading

Figure 2.12 Loading.



2.8.3 Loading

Figure 2.13 Compiling, linking and loading.



2.9 Firmware

- Firmware contains executable instructions stored in persistent memory attached to a device
 - Programmed with microprogramming
 - Layer of programming below a computer's machine-language
 - Microcode
 - Simple, fundamental instruction necessary to implement all machine-language operations

2.10 Middleware

- **Middleware is software for distributed systems**
 - Enables interactions among multiple processes running on one or more computers across a network
 - Facilitates heterogeneous distributed systems
 - Simplifies application programming
 - Example, Open DataBase Connectivity (ODBC)
 - Permits applications to access databases through middleware called an ODBC driver

Chapter 3 – Process Concepts

Outline

- 3.1 **Introduction**
- 3.1.1 **Definition of Process**
- 3.2 **Process States: Life Cycle of a Process**
- 3.3 **Process Management**
- 3.3.1 **Process States and State Transitions**
- 3.3.2 **Process Control Blocks (PCBs)/Process Descriptors**
- 3.3.3 **Process Operations**
- 3.3.4 **Suspend and Resume**
- 3.3.5 **Context Switching**
- 3.4 **Interrupts**
- 3.4.1 **Interrupt Processing**
- 3.4.2 **Interrupt Classes**
- 3.5 **Interprocess Communication**
- 3.5.1 **Signals**
- 3.5.2 **Message Passing**
- 3.6 **Case Study: UNIX Processes**

Objectives

- After reading this chapter, you should understand:
 - the concept of a process.
 - the process life cycle.
 - process states and state transitions.
 - process control blocks (PCBs)/process descriptors.
 - how processors transition between processes via context switching.
 - how interrupts enable hardware to communicate with software.
 - how processes converse with one another via interprocess communication (IPC).
 - UNIX processes.

3.1 Introduction

- Computers perform operations concurrently
 - For example, compiling a program, sending a file to a printer, rendering a Web page, playing music and receiving e-mail
 - Processes enable systems to perform and track simultaneous activities
 - Processes transition between process states
 - Operating systems perform operations on processes such as creating, destroying, suspending, resuming and waking

3.1.1 Definition of Process

- A program in execution
 - A process has its own address space consisting of:
 - Text region
 - Stores the code that the processor executes
 - Data region
 - Stores variables and dynamically allocated memory
 - Stack region
 - Stores instructions and local variables for active procedure calls

3.2 Process States: Life Cycle of a Process

- A process moves through a series of discrete process states:
 - *Running* state
 - The process is executing on a processor
 - *Ready* state
 - The process could execute on a processor if one were available
 - *Blocked* state
 - The process is waiting for some event to happen before it can proceed
- The OS maintains a *ready* list and a *blocked* list to store references to processes not running

3.3 Process Management

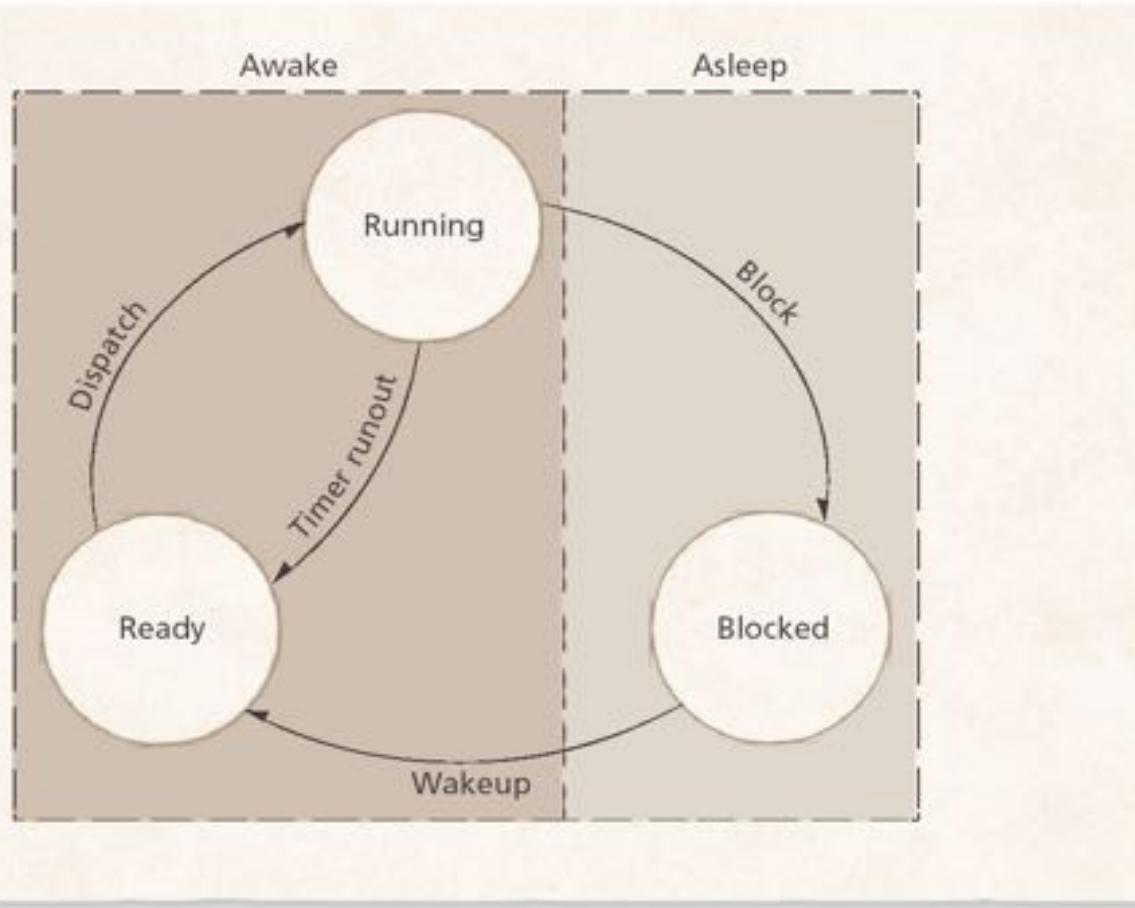
- Operating systems provide fundamental services to processes including:
 - Creating processes
 - Destroying processes
 - Suspending processes
 - Resuming processes
 - Changing a process's priority
 - Blocking processes
 - Waking up processes
 - Dispatching processes
 - Interprocess communication (IPC)

3.3.1 Process States and State Transitions

- Process states
 - The act of assigning a processor to the first process on the ready list is called dispatching
 - The OS may use an interval timer to allow a process to run for a specific time interval or quantum
 - Cooperative multitasking lets each process run to completion
- State Transitions
 - At this point, there are four possible state transitions
 - When a process is dispatched, it transitions from *ready* to *running*
 - When the quantum expires, it transitions from *running* to *ready*
 - When a process blocks, it transitions from *running* to *blocked*
 - When the event occurs, it transitions from *blocked* to *ready*

3.3.1 Process States and State Transitions

Figure 3.1 Process state transitions.



3.3.2 Process Control Blocks (PCBs)/Process Descriptors

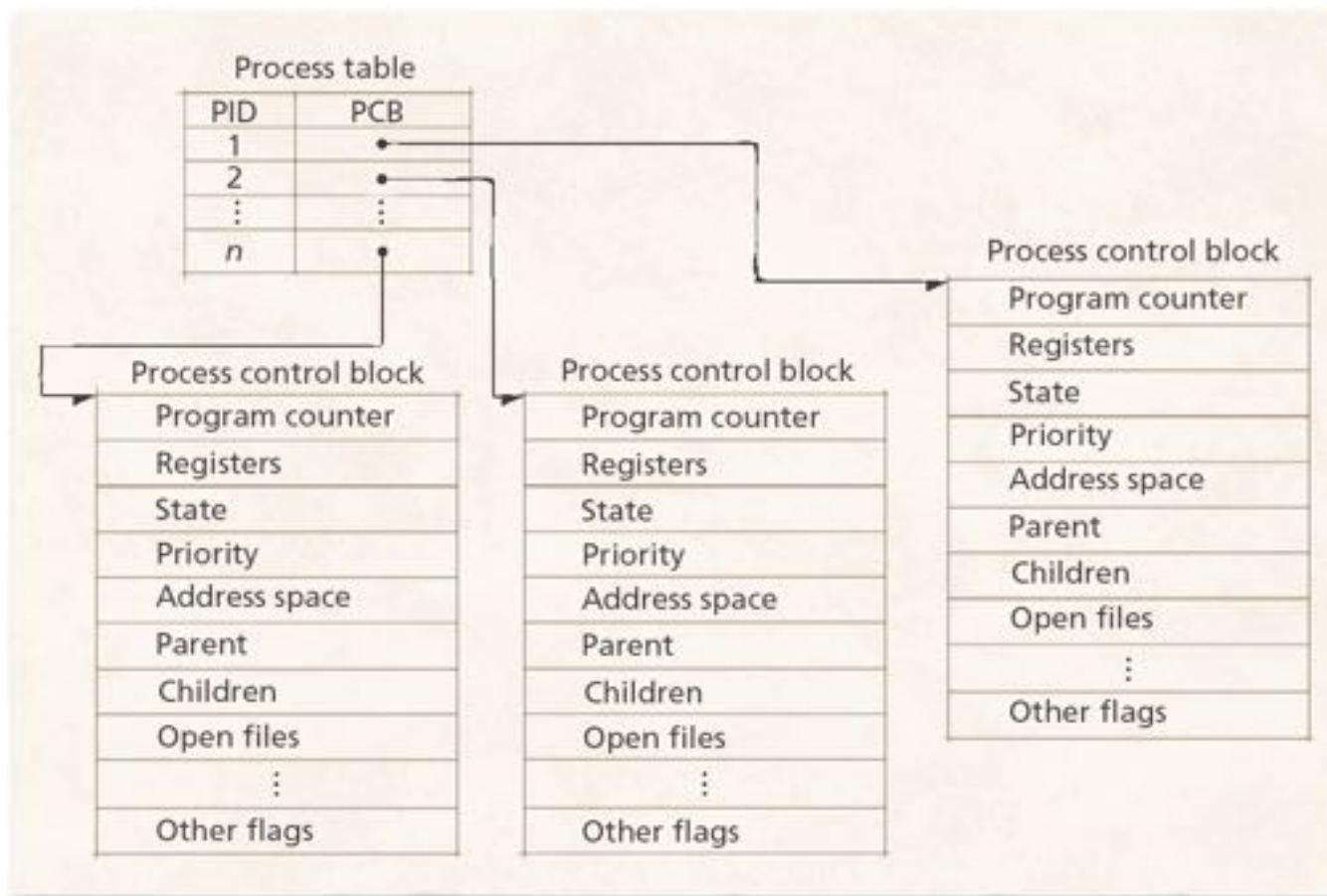
- PCBs maintain information that the OS needs to manage the process
 - Typically include information such as
 - Process identification number (PID)
 - Process state
 - Program counter
 - Scheduling priority
 - Credentials
 - A pointer to the process's parent process
 - Pointers to the process's child processes
 - Pointers to locate the process's data and instructions in memory
 - Pointers to allocated resources

3.3.2 Process Control Blocks (PCBs)/Process Descriptors

- **Process table**
 - The OS maintains pointers to each process's PCB in a system-wide or per-user process table
 - Allows for quick access to PCBs
 - When a process is terminated, the OS removes the process from the process table and frees all of the process's resources

3.3.2 Process Control Blocks (PCBs)/Process Descriptors

Figure 3.2 Process table and process control blocks.

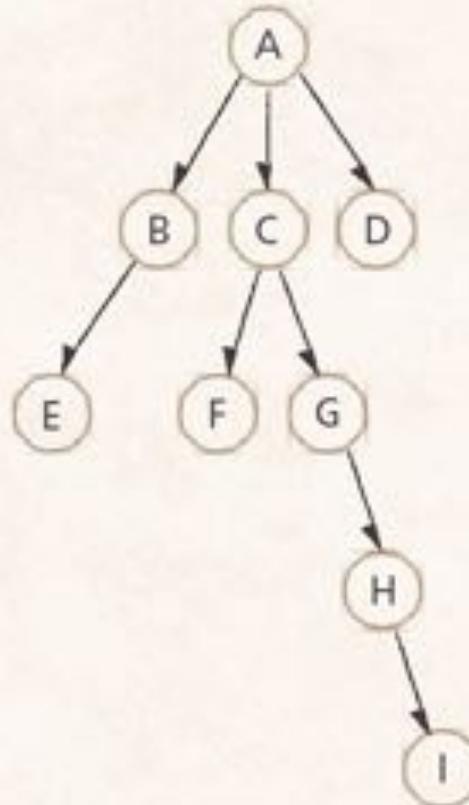


3.3.3 Process Operations

- A process may spawn a new process
 - The creating process is called the parent process
 - The created process is called the child process
 - Exactly one parent process creates a child
 - When a parent process is destroyed, operating systems typically respond in one of two ways:
 - Destroy all child processes of that parent
 - Allow child processes to proceed independently of their parents

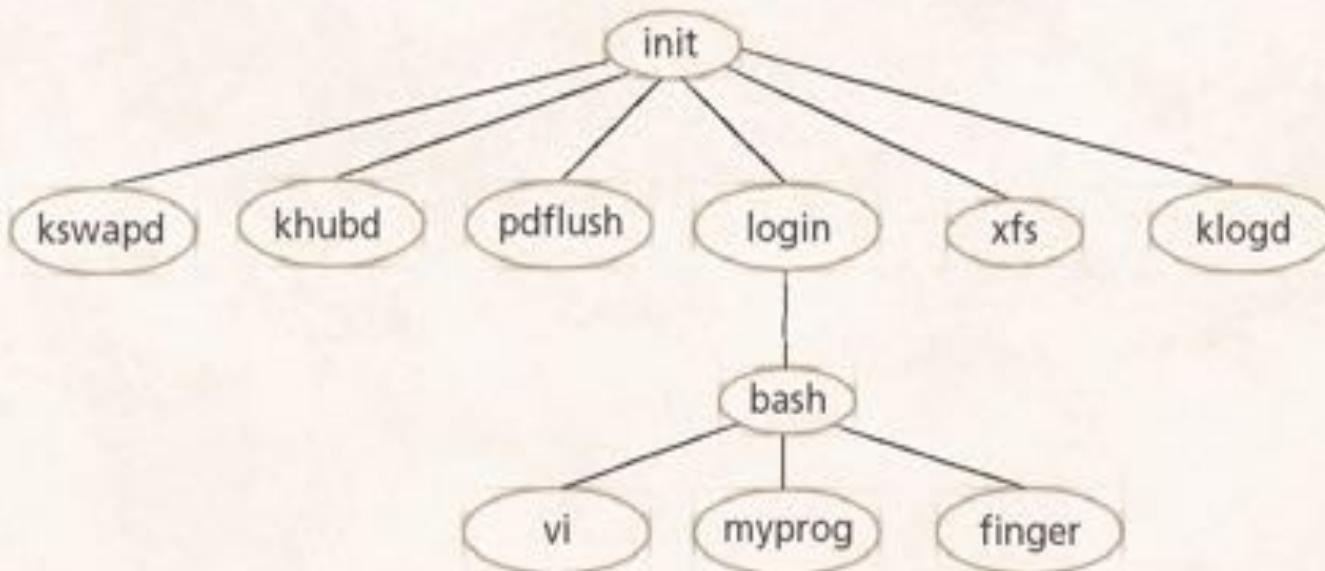
3.3.3 Process Operations

Figure 3.3 Process creation hierarchy.



3.3.3 Process Operations

Figure 3.4 Process hierarchy in Linux.

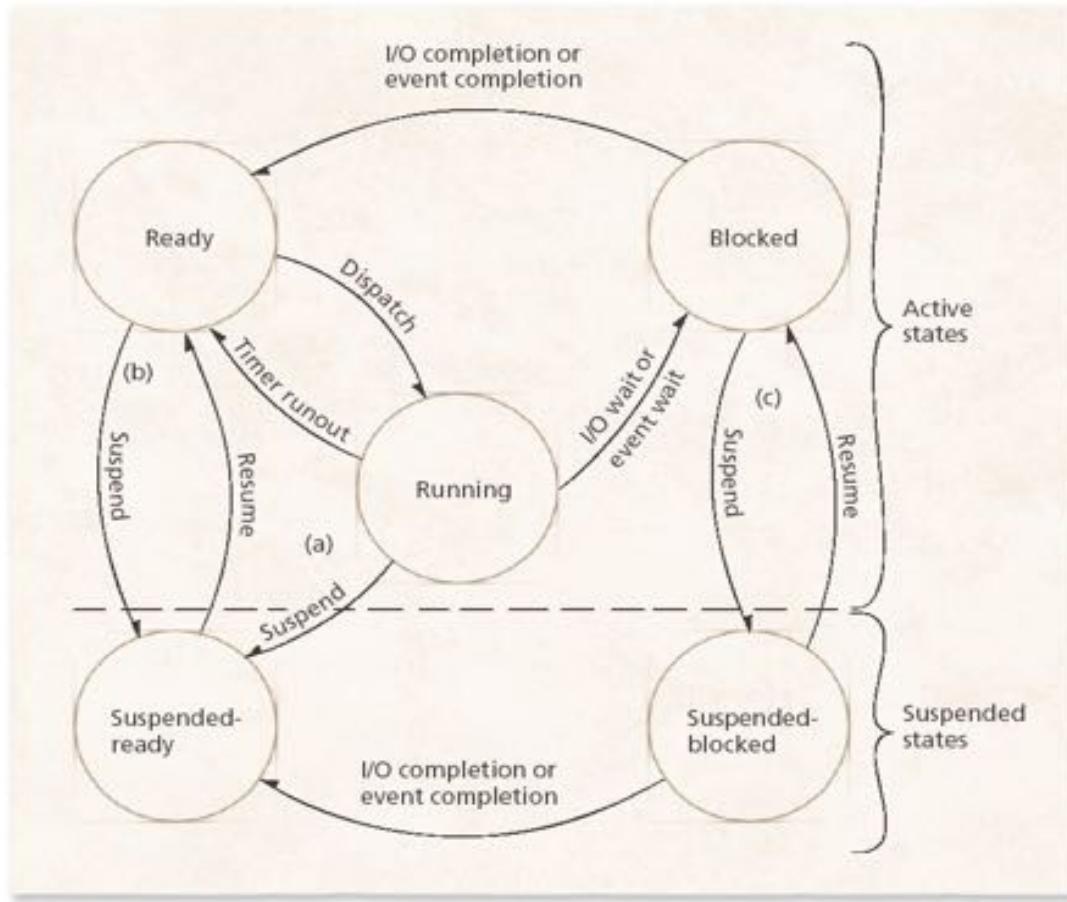


3.3.4 Suspend and Resume

- Suspending a process
 - Indefinitely removes it from contention for time on a processor without being destroyed
 - Useful for detecting security threats and for software debugging purposes
 - A suspension may be initiated by the process being suspended or by another process
 - A suspended process must be resumed by another process
 - Two suspended states:
 - *suspended ready*
 - *suspended blocked*

3.3.4 Suspend and Resume

Figure 3.5 Process state transitions with suspend and resume.

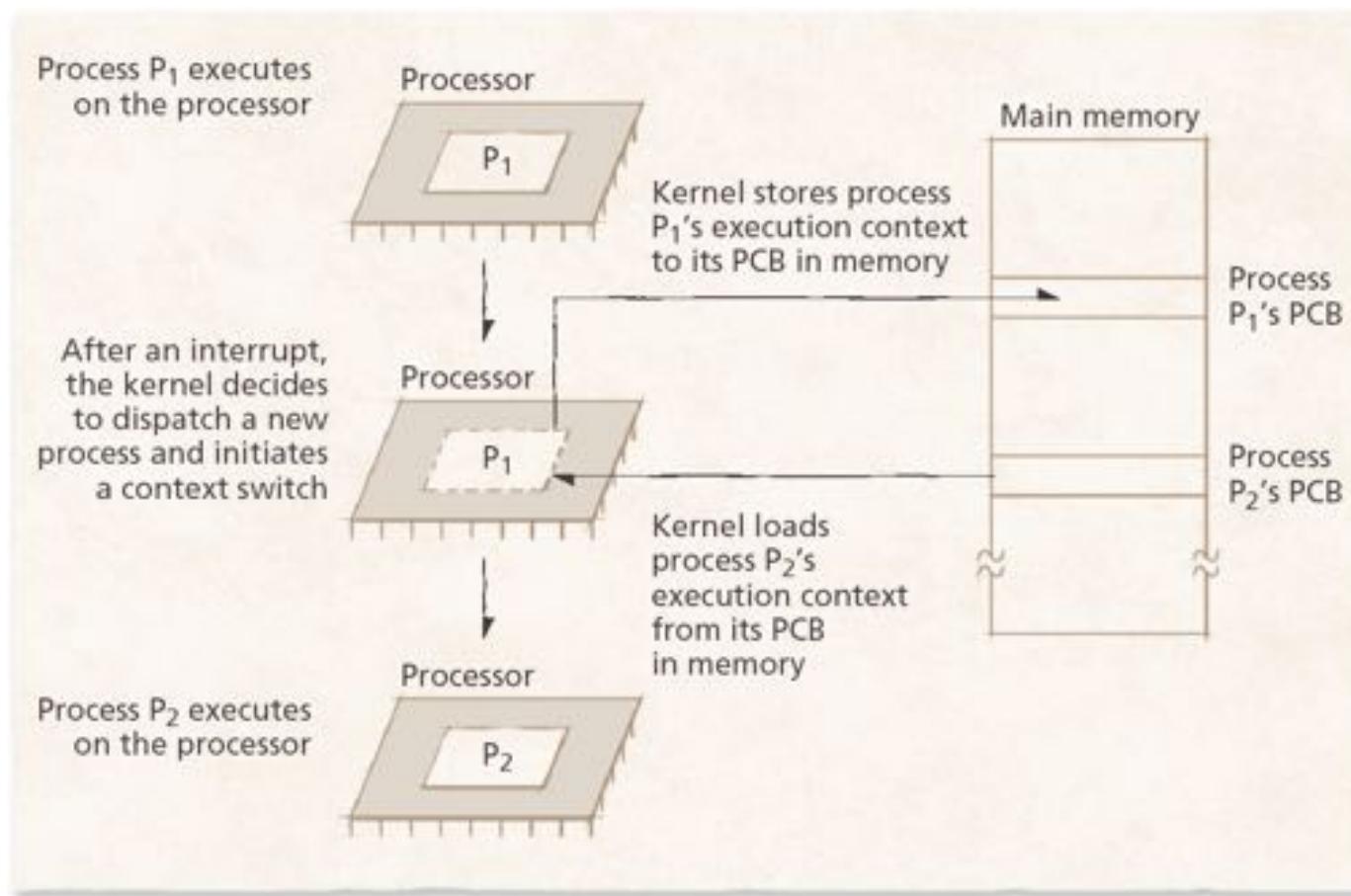


3.3.5 Context Switching

- Context switches
 - Performed by the OS to stop executing a *running* process and begin executing a previously *ready* process
 - Save the execution context of the *running* process to its PCB
 - Load the *ready* process's execution context from its PCB
 - Must be transparent to processes
 - Require the processor to not perform any “useful” computation
 - OS must therefore minimize context-switching time
 - Performed in hardware by some architectures

3.3.5 Context Switching

Figure 3.6 Context switch.



3.4 Interrupts

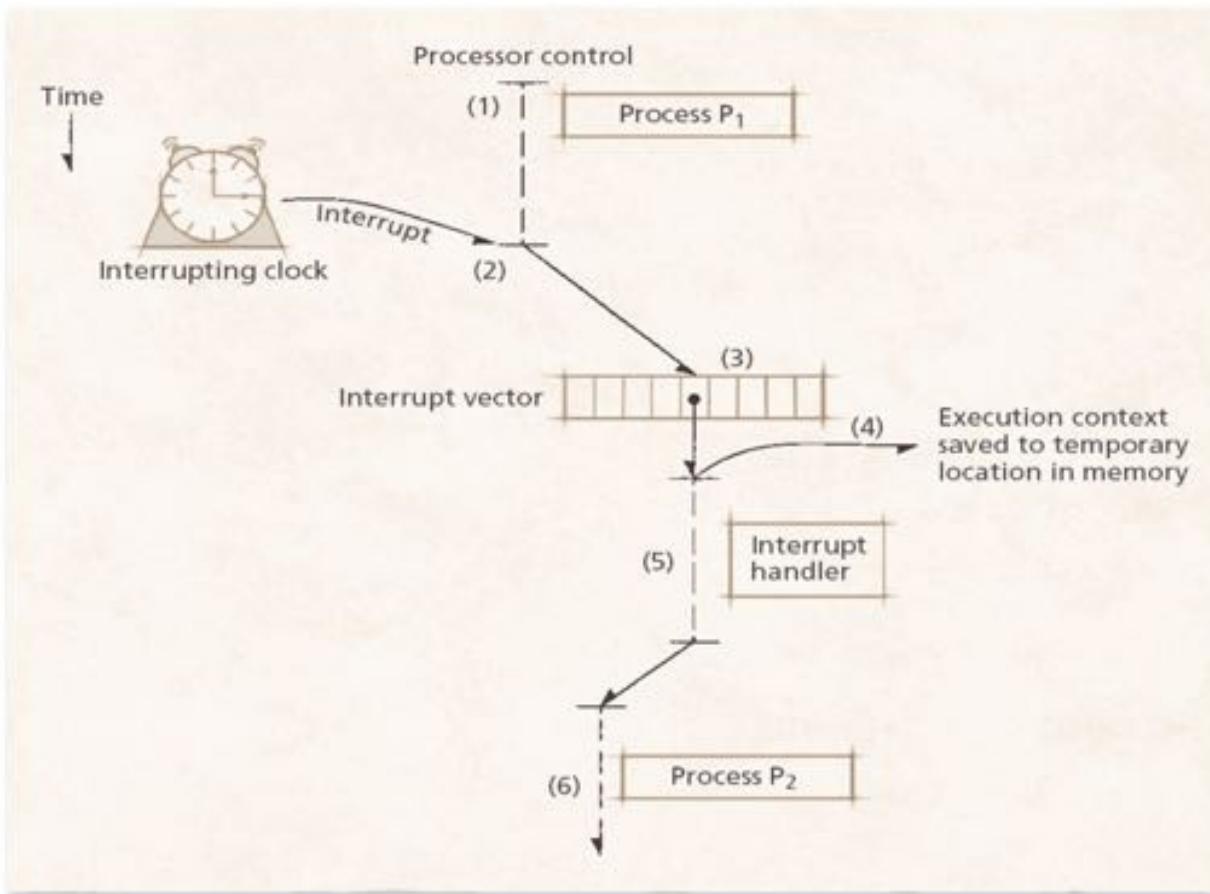
- Interrupts enable software to respond to signals from hardware
 - May be initiated by a running process
 - Interrupt is called a trap
 - Synchronous with the operation of the process
 - For example, dividing by zero or referencing protected memory
 - May be initiated by some event that may or may not be related to the running process
 - Asynchronous with the operation of the process
 - For example, a key is pressed on a keyboard or a mouse is moved
 - Low overhead
- Polling is an alternative approach
 - Processor repeatedly requests the status of each device
 - Increases in overhead as the complexity of the system increases

3.4.1 Interrupt Processing

- Handling interrupts
 - After receiving an interrupt, the processor completes execution of the current instruction, then pauses the current process
 - The processor will then execute one of the kernel's interrupt-handling functions
 - The interrupt handler determines how the system should respond
 - Interrupt handlers are stored in an array of pointers called the interrupt vector
 - After the interrupt handler completes, the interrupted process is restored and executed or the next process is executed

3.4.1 Interrupt Processing

Figure 3.7 Handling interrupts.



3.4.2 Interrupt Classes

- Supported interrupts depend on a system's architecture
 - The IA-32 specification distinguishes between two types of signals a processor may receive:
 - Interrupts
 - Notify the processor that an event has occurred or that an external device's status has changed
 - Generated by devices external to a processor
 - Exceptions
 - Indicate that an error has occurred, either in hardware or as a result of a software instruction
 - Classified as faults, traps or aborts

3.4.2 Interrupt Classes

Figure 3.8 Common interrupt types recognized in the Intel IA-32 architecture.

<i>Interrupt Type</i>	<i>Description of Interrupts in Each Type</i>
I/O	These are initiated by the input/output hardware. They notify a processor that the status of a channel or device has changed. I/O interrupts are caused when an I/O operation completes, for example.
Timer	A system may contain devices that generate interrupts periodically. These interrupts can be used for tasks such as timekeeping and performance monitoring. Timers also enable the operating system to determine if a process's quantum has expired.
Interprocessor interrupts	These interrupts allow one processor to send a message to another in a multiprocessor system.

3.4.2 Interrupt Classes

Figure 3.9 Intel IA-32 exception classes.

<i>Exception Class</i>	<i>Description of Exceptions in Each Class</i>
Fault	These are caused by a wide range of problems that may occur as a program's machine-language instructions are executed. These problems include division by zero, data (being operated upon) in the wrong format, attempt to execute an invalid operation code, attempt to reference a memory location beyond the limits of real memory, attempt by a user process to execute a privileged instruction and attempt to reference a protected resource.
Trap	These are generated by exceptions such as overflow (when the value stored by a register exceeds the capacity of the register) and when program control reaches a breakpoint in code.
Abort	This occurs when the processor detects an error from which a process cannot recover. For example, when an exception-handling routine itself causes an exception, the processor may not be able to handle both errors sequentially. This is called a double-fault exception, which terminates the process that initiated it.

3.5 Interprocess Communication

- Many operating systems provide mechanisms for interprocess communication (IPC)
 - Processes must communicate with one another in multiprogrammed and networked environments
 - For example, a Web browser retrieving data from a distant server
 - Essential for processes that must coordinate activities to achieve a common goal

3.5.1 Signals

- Software interrupts that notify a process that an event has occurred
 - Do not allow processes to specify data to exchange with other processes
 - Processes may catch, ignore or mask a signal
 - Catching a signal involves specifying a routine that the OS calls when it delivers the signal
 - Ignoring a signal relies on the operating system's default action to handle the signal
 - Masking a signal instructs the OS to not deliver signals of that type until the process clears the signal mask

3.5.2 Message Passing

- **Message-based interprocess communication**
 - Messages can be passed in one direction at a time
 - One process is the sender and the other is the receiver
 - Message passing can be bidirectional
 - Each process can act as either a sender or a receiver
 - Messages can be blocking or nonblocking
 - Blocking requires the receiver to notify the sender when the message is received
 - Nonblocking enables the sender to continue with other processing
 - Popular implementation is a pipe
 - A region of memory protected by the OS that serves as a buffer, allowing two or more processes to exchange data

3.5.2 Message Passing

- IPC in distributed systems
 - Transmitted messages can be flawed or lost
 - Acknowledgement protocols confirm that transmissions have been properly received
 - Timeout mechanisms retransmit messages if acknowledgements are not received
 - Ambiguously named processes lead to incorrect message referencing
 - Messages are passed between computers using numbered ports on which processes listen, avoiding this problem
 - Security is a significant problem
 - Ensuring authentication

3.6 Case Study: UNIX Processes

- **UNIX processes**
 - All processes are provided with a set of memory addresses, called a virtual address space
 - A process's PCB is maintained by the kernel in a protected region of memory that user processes cannot access
 - A UNIX PCB stores:
 - The contents of the processor registers
 - PID
 - The program counter
 - The system stack
 - All processes are listed in the process table

3.6 Case Study: UNIX Processes

- **UNIX processes continued**
 - All processes interact with the OS via system calls
 - A process can spawn a child process by using the fork system call, which creates a copy of the parent process
 - Child receives a copy of the parent's resources as well
 - Process priorities are integers between -20 and 19 (inclusive)
 - A lower numerical priority value indicates a higher scheduling priority
 - UNIX provides IPC mechanisms, such as pipes, to allow unrelated processes to transfer data

3.6 Case Study: UNIX Processes

Figure 3.10 UNIX system calls.

<i>System Call</i>	<i>Description</i>
fork	Spawns a child process and allocates to that process a copy of its parent's resources.
exec	Loads a process's instructions and data into its address space from a file.
wait	Causes the calling process to block until its child process has terminated.
signal	Allows a process to specify a signal handler for a particular signal type.
exit	Terminates the calling process.
nice	Modifies a process's scheduling priority.

Chapter 4 – Thread Concepts

Outline

- 4.1 **Introduction**
- 4.2 **Definition of Thread**
- 4.3 **Motivation for Threads**
- 4.4 **Thread States: Life Cycle of a Thread**
- 4.5 **Thread Operations**
- 4.6 **Threading Models**
 - 4.6.1 **User-Level Threads**
 - 4.6.2 **Kernel-Level Threads**
 - 4.6.3 **Combining User- and Kernel-Level Threads**
- 4.7 **Thread Implementation Considerations**
 - 4.7.1 **Thread Signal Delivery**
 - 4.7.2 **Thread Termination**
- 4.8 **POSIX and Pthreads**
- 4.9 **Linux Threads**
- 4.10 **Windows XP Threads**
- 4.11 **Java Multithreading Case Study, Part 1: Introduction to Java Threads**

Objectives

- After reading this chapter, you should understand:
 - the motivation for creating threads.
 - the similarities and differences between processes and threads.
 - the various levels of support for threads.
 - the life cycle of a thread.
 - thread signaling and cancellation.
 - the basics of POSIX, Linux, Windows XP and Java threads.

4.1 Introduction

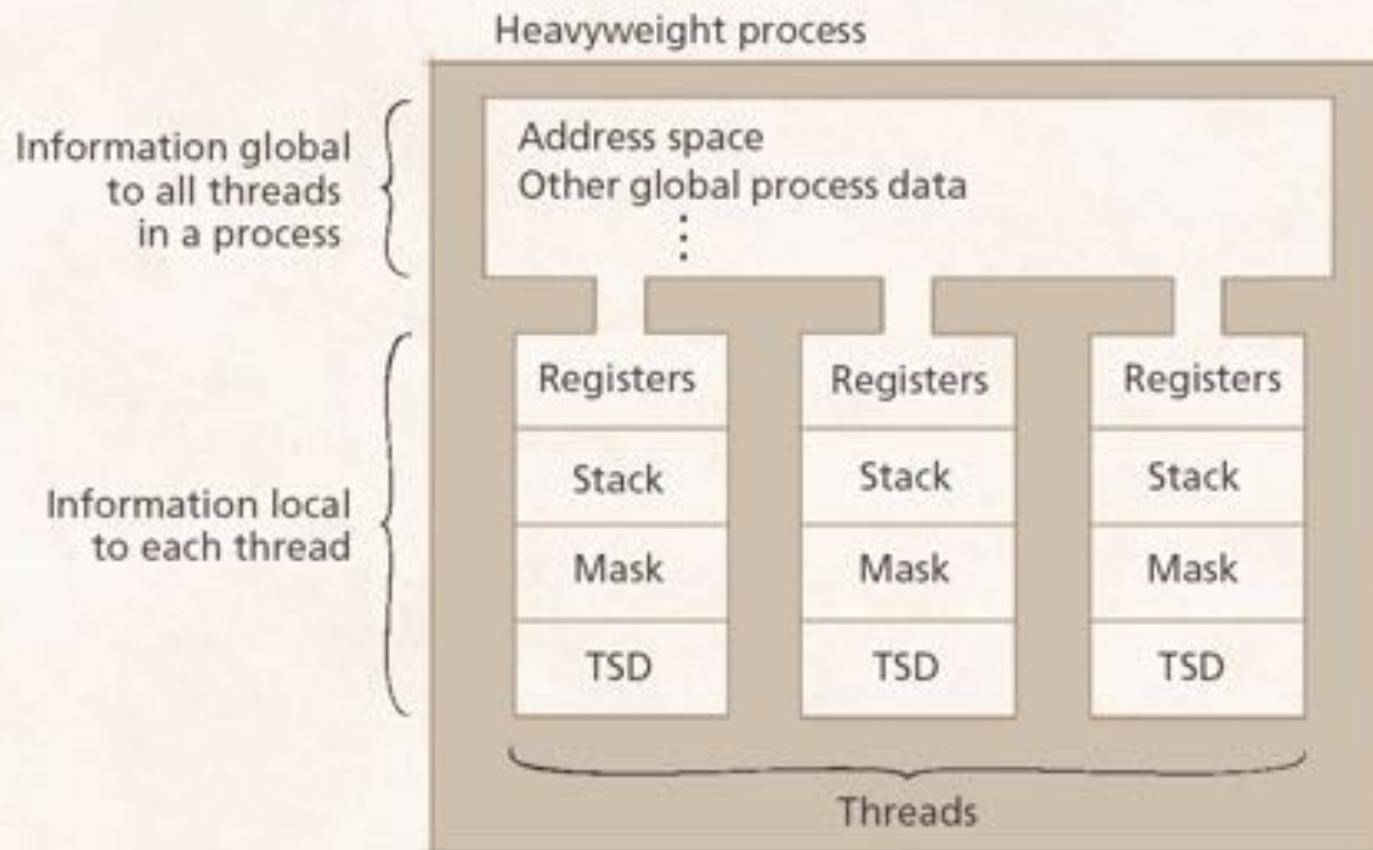
- General-purpose languages such as Java, C#, Visual C++ .NET, Visual Basic .NET and Python have made concurrency primitives available to applications programmer
- Multithreading
 - Programmer specifies applications contain threads of execution
 - Each thread designate a portion of a program that may execute concurrently with other threads

4.2 Definition of Thread

- Thread
 - Lightweight process (LWP)
 - Threads of instructions or thread of control
 - Shares address space and other global information with its process
 - Registers, stack, signal masks and other thread-specific data are local to each thread
- Threads may be managed by the operating system or by a user application
- Examples: Win32 threads, C-threads, Pthreads

4.2 Definition of Thread

Figure 4.1 Thread Relationship to Processes.



4.3 Motivation for Threads

- Threads have become prominent due to trends in
 - Software design
 - More naturally expresses inherently parallel tasks
 - Performance
 - Scales better to multiprocessor systems
 - Cooperation
 - Shared address space incurs less overhead than IPC

4.3 Motivation for Threads

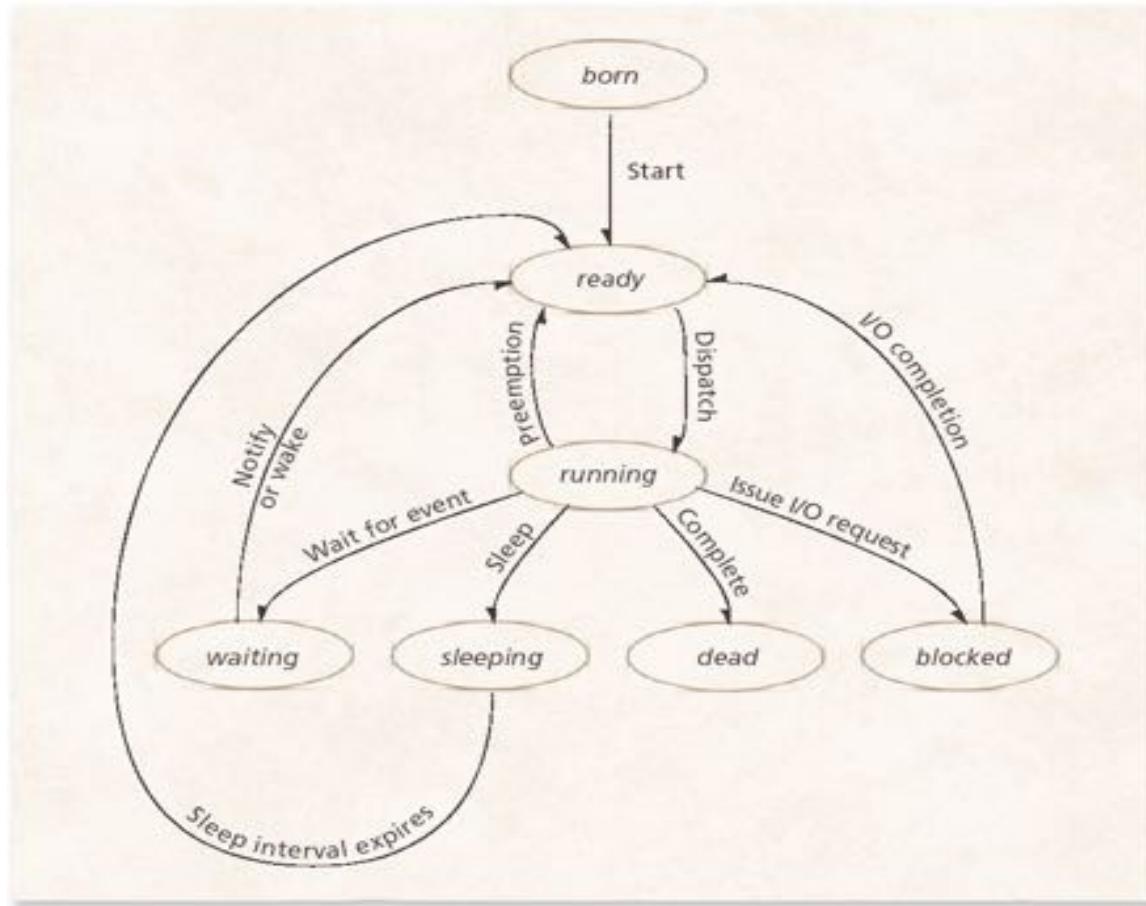
- Each thread transitions among a series of discrete thread states
- Threads and processes have many operations in common (e.g. create, exit, resume, and suspend)
- Thread creation does not require operating system to initialize resources that are shared between parent processes and its threads
 - Reduces overhead of thread creation and termination compared to process creation and termination

4.4 Thread States: Life Cycle of a Thread

- Thread states
 - *Born* state
 - *Ready* state (*runnable* state)
 - *Running* state
 - *Dead* state
 - *Blocked* state
 - *Waiting* state
 - *Sleeping* state
 - Sleep interval specifies for how long a thread will sleep

4.4 Thread States: Life Cycle of a Thread

Figure 4.2 Thread life cycle.



4.5 Thread Operations

- Threads and processes have common operations
 - Create
 - Exit (terminate)
 - Suspend
 - Resume
 - Sleep
 - Wake

4.5 Thread Operations

- Thread operations do not correspond precisely to process operations
 - Cancel
 - Indicates that a thread should be terminated, but does not guarantee that the thread will be terminated
 - Threads can mask the cancellation signal
 - Join
 - A primary thread can wait for all other threads to exit by joining them
 - The joining thread blocks until the thread it joined exits

4.6 Threading Models

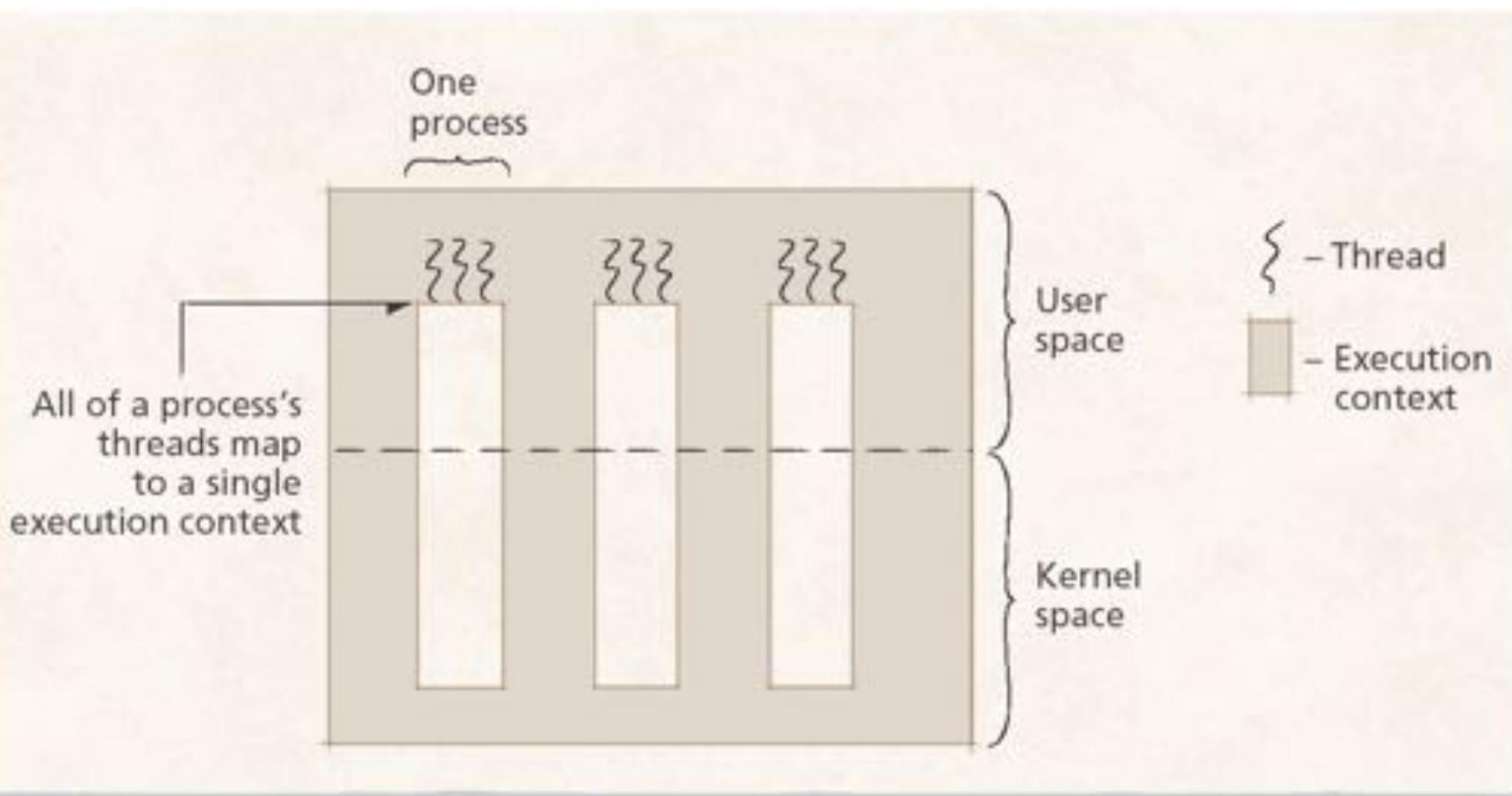
- Three most popular threading models
 - User-level threads
 - Kernel-level threads
 - Combination of user- and kernel-level threads

4.6.1 User-level Threads

- User-level threads perform threading operations in user space
 - Threads are created by runtime libraries that cannot execute privileged instructions or access kernel primitives directly
- User-level thread implementation
 - Many-to-one thread mappings
 - Operating system maps all threads in a multithreaded process to single execution context
 - Advantages
 - User-level libraries can schedule its threads to optimize performance
 - Synchronization performed outside kernel, avoids context switches
 - More portable
 - Disadvantage
 - Kernel views a multithreaded process as a single thread of control
 - Can lead to suboptimal performance if a thread issues I/O
 - Cannot be scheduled on multiple processors at once

4.6.1 User-level Threads

Figure 4.3 User-level threads.

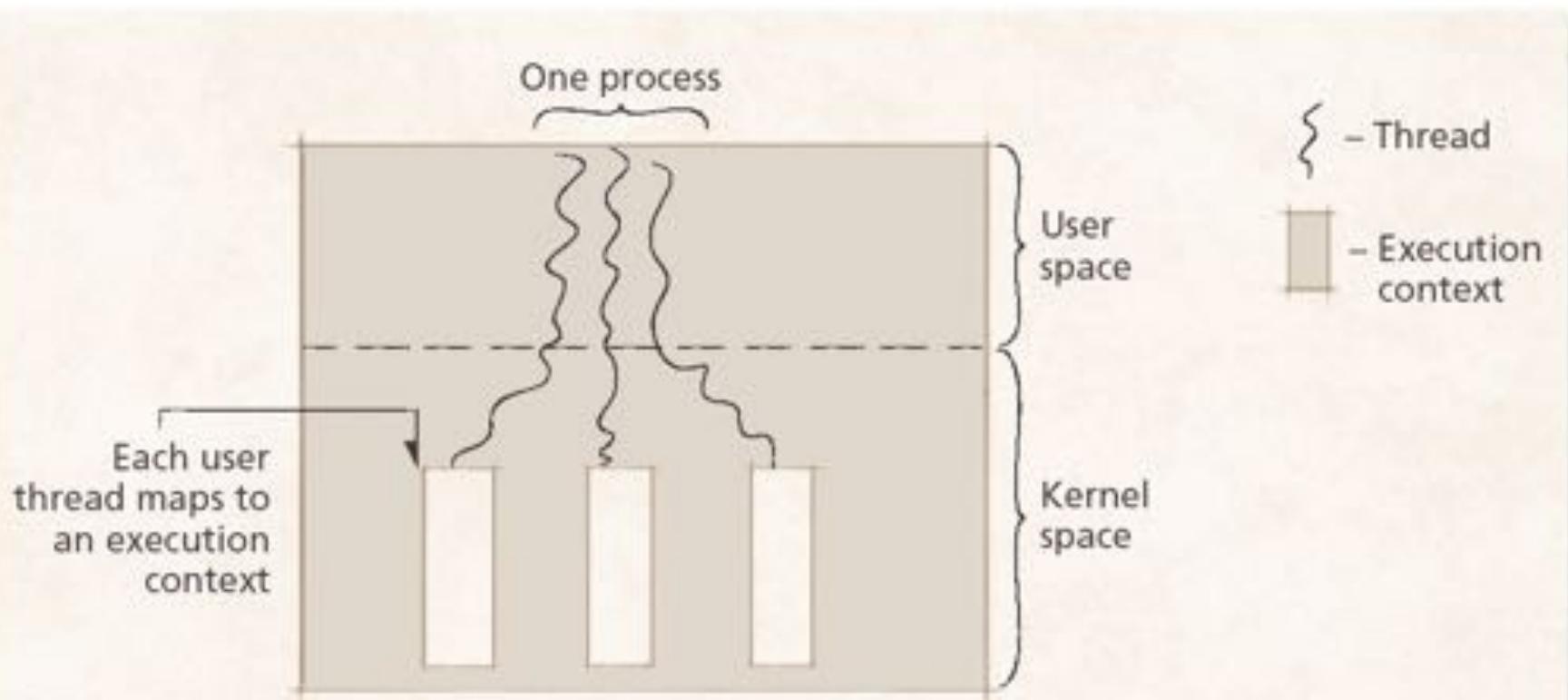


4.6.2 Kernel-level Threads

- Kernel-level threads attempt to address the limitations of user-level threads by mapping each thread to its own execution context
 - Kernel-level threads provide a one-to-one thread mapping
 - Advantages: Increased scalability, interactivity, and throughput
 - Disadvantages: Overhead due to context switching and reduced portability due to OS-specific APIs
- Kernel-level threads are not always the optimal solution for multithreaded applications

4.6.2 Kernel-level Threads

Figure 4.4 Kernel-level threads.

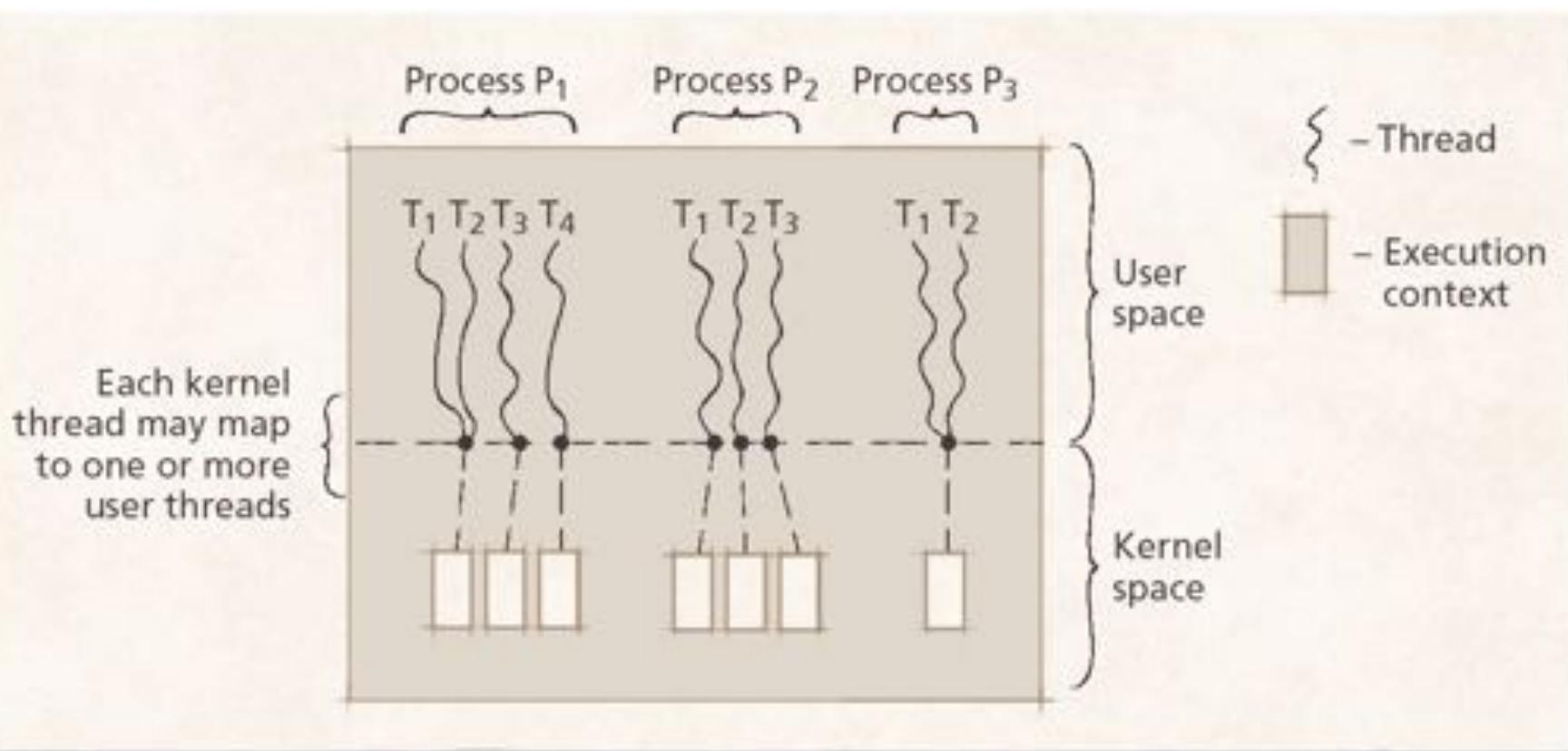


4.6.3 Combining User- and Kernel-level Threads

- The combination of user- and kernel-level thread implementation
 - Many-to-many thread mapping (m -to- n thread mapping)
 - Number of user and kernel threads need not be equal
 - Can reduce overhead compared to one-to-one thread mappings by implementing thread pooling
- Worker threads
 - Persistent kernel threads that occupy the thread pool
 - Improves performance in environments where threads are frequently created and destroyed
 - Each new thread is executed by a worker thread
- Scheduler activation
 - Technique that enables user-level library to schedule its threads
 - Occurs when the operating system calls a user-level threading library that determines if any of its threads need rescheduling

4.6.3 Combining User- and Kernel-level Threads

Figure 4.5 Hybrid threading model.

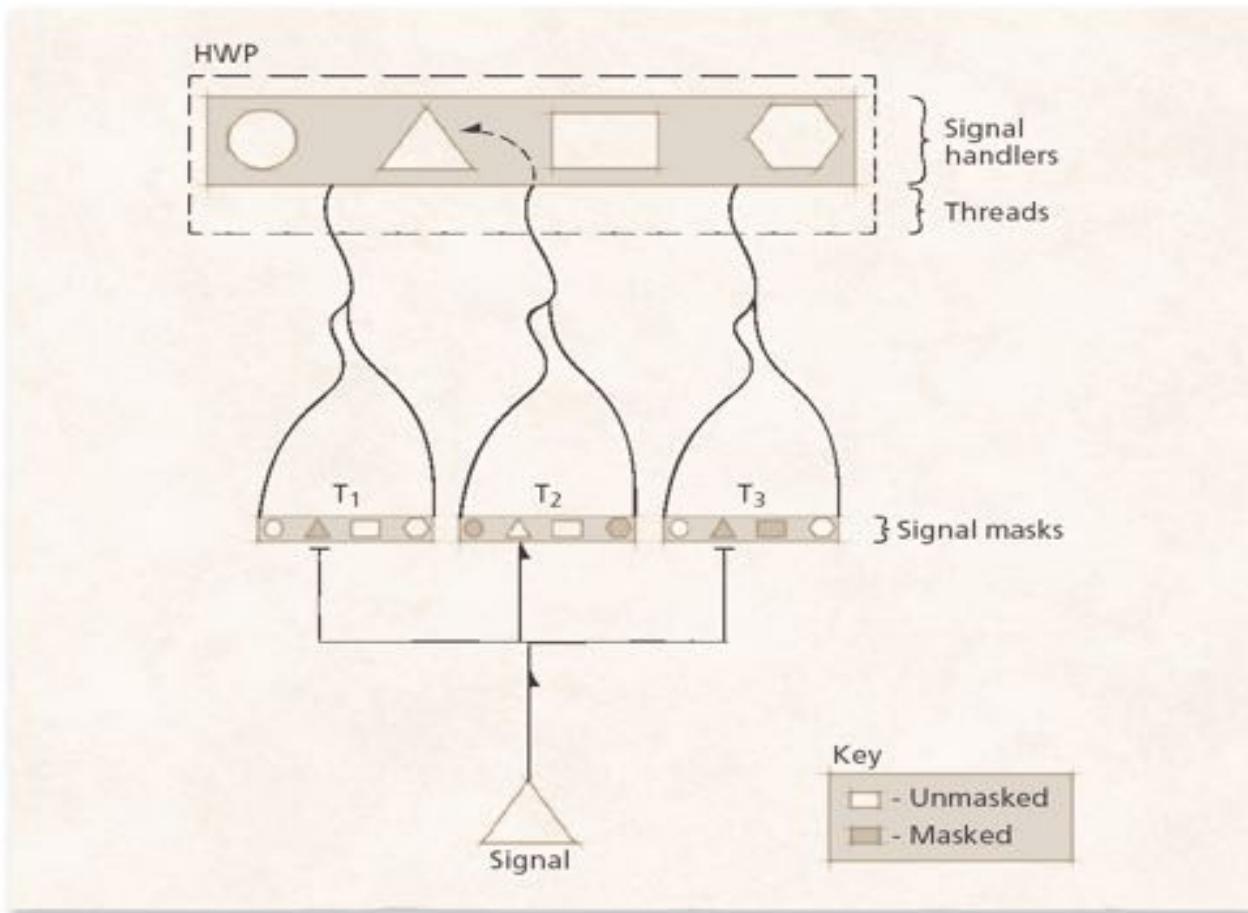


4.7.1 Thread Signal Delivery

- Two types of signals
 - Synchronous:
 - Occur as a direct result of program execution
 - Should be delivered to currently executing thread
 - Asynchronous
 - Occur due to an event typically unrelated to the current instruction
 - Threading library must determine each signal's recipient so that asynchronous signals are delivered properly
- Each thread is usually associated with a set of pending signals that are delivered when it executes
- Thread can mask all signals except those that it wishes to receive

4.7.1 Thread Signal Delivery

Figure 4.6 Signal masking.



4.7.2 Thread Termination

- Thread termination (cancellation)
 - Differs between thread implementations
 - Prematurely terminating a thread can cause subtle errors in processes because multiple threads share the same address space
 - Some thread implementations allow a thread to determine when it can be terminated to prevent process from entering inconsistent state

4.8 POSIX and Pthreads

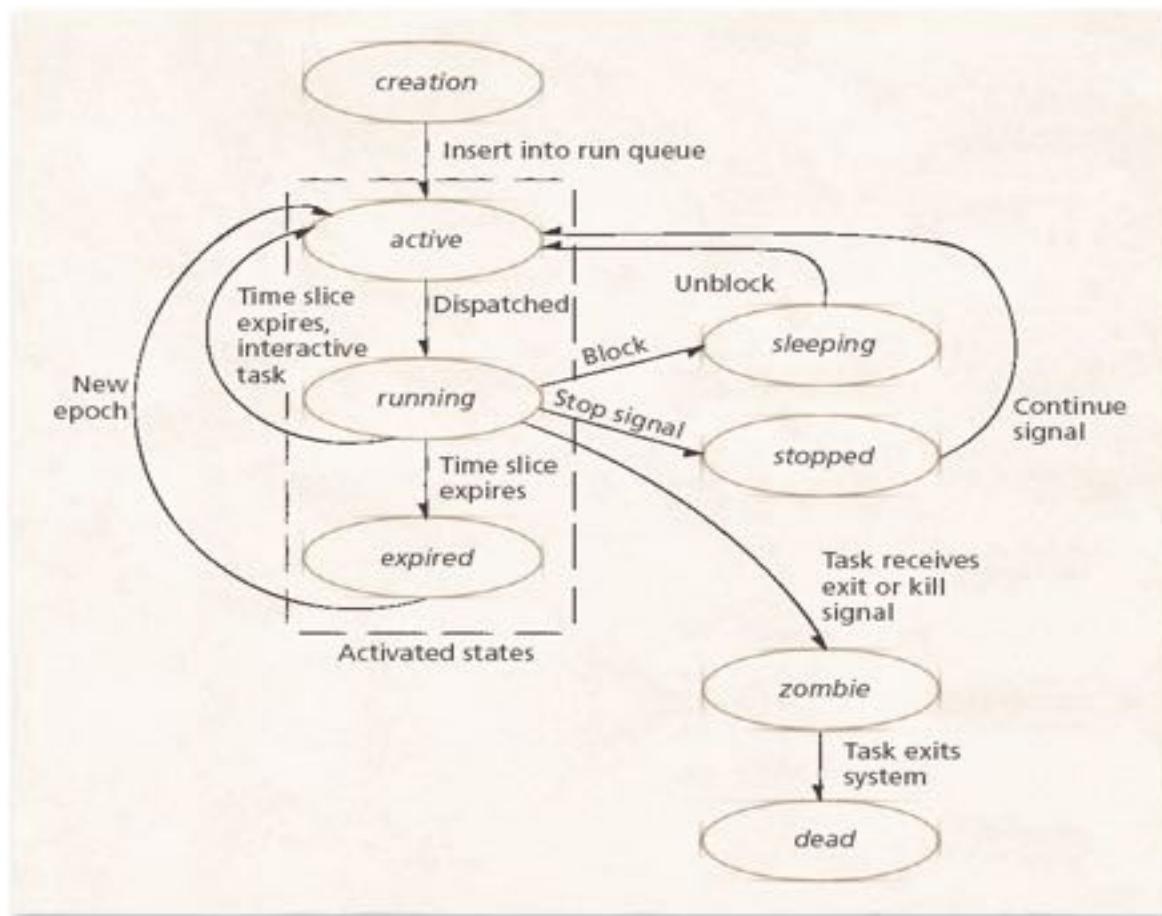
- Threads that use the POSIX threading API are called Pthreads
 - POSIX states that processor registers, stack and signal mask are maintained individually for each thread
 - POSIX specifies how operating systems should deliver signals to Pthreads in addition to specifying several thread-cancellation modes

4.9 Linux Threads

- Linux allocates the same type of process descriptor to processes and threads (tasks)
- Linux uses the UNIX-based system call `fork` to spawn child tasks
- To enable threading, Linux provides a modified version named `clone`
 - `clone` accepts arguments that specify which resources to share with the child task

4.9 Linux Threads

Figure 4.7 Linux task state-transition diagram.



4.10 Windows XP Threads

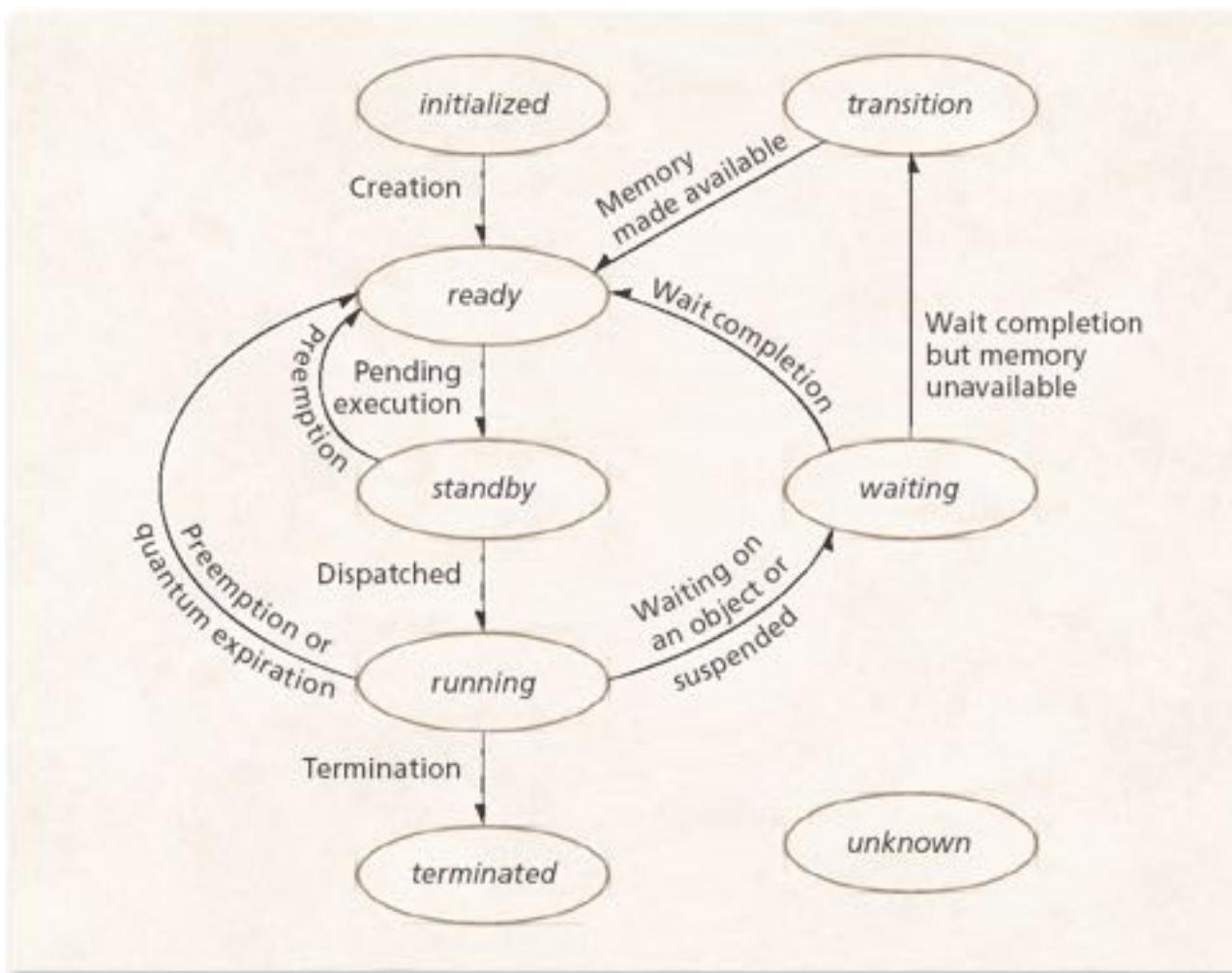
- Threads
 - Actual unit of execution dispatched to a processor
 - Execute a piece of the process's code in the process's context, using the process's resources
 - Execution context contains
 - Runtime stack
 - State of the machine's registers
 - Several attributes

4.10 Windows XP Threads

- Windows XP threads can create fibers
 - Fiber is scheduled for execution by the thread that creates it, rather than the scheduler
- Windows XP provides each process with a thread pool that consists of a number of worker threads, which are kernel threads that execute functions specified by user threads

4.10 Windows XP Threads

Figure 4.8 Windows XP thread state-transition diagram.



4.11 Java Multithreading Case Study, Part I: Introduction to Java Threads

- Java allows the application programmer to create threads that can port to many computing platforms
- Threads
 - Created by class `Thread`
 - Execute code specified in a `Runnable` object's `run` method
- Java supports operations such as naming, starting and joining threads

4.11 Java Multithreading Case Study, Part I: Introduction to Java Threads

Figure 4.9 Java threads being created, starting, sleeping and printing. (Part 1 of 4.)

```
1 // Fig. 4.9: ThreadTester.java
2 // Multiple threads printing at different intervals.
3
4 public class ThreadTester {
5
6     public static void main( String [] args )
7     {
8         // create and name each thread
9         PrintThread thread1 = new PrintThread( "thread1" );
10        PrintThread thread2 = new PrintThread( "thread2" );
11        PrintThread thread3 = new PrintThread( "thread3" );
12
13        System.err.println( "Starting threads" );
14
15        thread1.start(); // start thread1; place it in ready state
16        thread2.start(); // start thread2; place it in ready state
17        thread3.start(); // start thread3; place it in ready state
18
19        System.err.println( "Threads started, main ends\n" );
20
21    } // end main
22
23 } // end class ThreadTester
24
```

4.11 Java Multithreading Case Study, Part I: Introduction to Java Threads

Figure 4.9 Java threads being created, starting, sleeping and printing. (Part 2 of 4.)

```
25 // class PrintThread controls thread execution
26 class PrintThread extends Thread {
27     private int sleepTime;
28
29     // assign name to thread by calling superclass constructor
30     public PrintThread( String name )
31     {
32         super( name );
33
34         // pick random sleep time between 0 and 5 seconds
35         sleepTime = ( int ) ( Math.random() * 5001 );
36     } // end PrintThread constructor
37
38     // method run is the code to be executed by new thread
39     public void run()
40     {
41         // put thread to sleep for sleepTime amount of time
42         try {
43             System.err.println( getName() + " going to sleep for " +
44                         sleepTime + " milliseconds" );
45         }
```

4.11 Java Multithreading Case Study, Part I: Introduction to Java Threads

Figure 4.9 Java threads being created, starting, sleeping and printing. (Part 3 of 4.)

```
46         Thread.sleep( sleepTime );
47     } // end try
48
49     // if thread interrupted during sleep, print stack trace
50     catch ( InterruptedException exception ) {
51         exception.printStackTrace();
52     } // end catch
53
54     // print thread name
55     System.err.println( getName() + " done sleeping" );
56
57 } // end method run
58
59 } // end class PrintThread
```

4.11 Java Multithreading Case Study, Part I: Introduction to Java Threads

Figure 4.9 Java threads being created, starting, sleeping and printing. (Part 4 of 4.)

Sample Output 1:

```
Starting threads
Threads started, main ends

thread1 going to sleep for 1217 milliseconds
thread2 going to sleep for 3989 milliseconds
thread3 going to sleep for 662 milliseconds
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping
```

Sample Output 2:

```
Starting threads
thread1 going to sleep for 314 milliseconds
thread2 going to sleep for 1990 milliseconds
Threads started, main ends

thread3 going to sleep for 3016 milliseconds
thread1 done sleeping
thread2 done sleeping
thread3 done sleeping
```

Chapter 5 – Asynchronous Concurrent Execution

Outline

- 5.1 **Introduction**
- 5.2 **Mutual Exclusion**
- 5.2.1 **Java Multithreading Case Study**
- 5.2.2 **Critical Sections**
- 5.2.3 **Mutual Exclusion Primitives**
- 5.3 **Implementing Mutual Exclusion Primitives**
- 5.4 **Software Solutions to the Mutual Exclusion Problem**
- 5.4.1 **Dekker's Algorithm**
- 5.4.2 **Peterson's Algorithm**
- 5.4.3 **N-Thread Mutual Exclusion: Lamport's Bakery Algorithm**
- 5.5 **Hardware Solutions to the Mutual Exclusion Problem**
- 5.5.1 **Disabling Interrupts**
- 5.5.2 **Test-and-Set Instruction**
- 5.5.3 **Swap Instruction**

Chapter 5 – Asynchronous Concurrent Execution

Outline (continued)

5.6 Semaphores

5.6.1 Mutual Exclusion with Semaphores

5.6.2 Thread Synchronization with Semaphores

5.6.3 Counting Semaphores

5.6.4 Implementing Semaphores

Objectives

- After reading this chapter, you should understand:
 - the challenges of synchronizing concurrent processes and threads.
 - critical sections and the need for mutual exclusion.
 - how to implement mutual exclusion primitives in software.
 - hardware mutual exclusion primitives.
 - semaphore usage and implementation.

5.1 Introduction

- Concurrent execution
 - More than one thread exists in system at once
 - Can execute independently or in cooperation
 - Asynchronous execution
 - Threads generally independent
 - Must occasionally communicate or synchronize
 - Complex and difficult to manage such interactions

5.2 Mutual Exclusion

- Problem of two threads accessing data simultaneously
 - Data can be put in inconsistent state
 - Context switch can occur at anytime, such as before a thread finishes modifying value
 - Such data must be accessed in mutually exclusive way
 - Only one thread allowed access at one time
 - Others must wait until resource is unlocked
 - Serialized access
 - Must be managed such that wait time not unreasonable

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

- Producer/Consumer relationship
 - One thread creates data to store in shared object
 - Second thread reads data from that object
 - Large potential for data corruption if unsynchronized

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.1 Buffer interface used in producer/consumer examples.

```
1 // Fig. 5.1: Buffer.java
2 // Buffer interface specifies methods to access buffer data.
3
4 public interface Buffer
5 {
6     public void set( int value );    // place value into Buffer
7     public int get();                // return value from Buffer
8 }
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.2 Producer class represents the producer thread in a producer/consumer relationship. (1 of 3)

```
1 // Fig. 5.2: Producer.java
2 // Producer's run method controls a producer thread that
3 // stores values from 1 to 4 in Buffer sharedLocation.
4
5 public class Producer extends Thread
6 {
7     private Buffer sharedLocation; // reference to shared object
8
9     // Producer constructor
10    public Producer( Buffer shared )
11    {
12        super( "Producer" ); // create thread named "Producer"
13        sharedLocation = shared; // initialize sharedLocation
14    } // end Producer constructor
15
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.2 Producer class represents the producer thread in a producer/consumer relationship. (2 of 3)

```
16     // Producer method run stores values from
17     // 1 to 4 in Buffer sharedLocation
18     public void run()
19     {
20         for ( int count = 1; count <= 4; count++ )
21         {
22             // sleep 0 to 3 seconds, then place value in Buffer
23             try
24             {
25                 Thread.sleep( ( int )( Math.random() * 3001 ) );
26                 sharedLocation.set( count ); // write to the buffer
27             } // end try
28
29             // if sleeping thread interrupted, print stack trace
30             catch ( InterruptedException exception )
31             {
32                 exception.printStackTrace();
33             } // end catch
34
35     } // end for
```



5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.2 Producer class represents the producer thread in a producer/consumer relationship. (3 of 3)

```
36
37     System.err.println( getName() + " done producing." +
38         "\nTerminating " + getName() + "." );
39
40 } // end method run
41
42 } // end class Producer
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.3 Consumer class represents the consumer thread in a producer/consumer relationship. (1 of 3)

```
1 // Fig. 5.3: Consumer.java
2 // Consumer's run method controls a thread that loops four
3 // times and reads a value from sharedLocation each time.
4
5 public class Consumer extends Thread
6 {
7     private Buffer sharedLocation; // reference to shared object
8
9     // Consumer constructor
10    public Consumer( Buffer shared )
11    {
12        super( "Consumer" ); // create thread named "Consumer"
13        sharedLocation = shared; // initialize sharedLocation
14    } // end Consumer constructor
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.3 Consumer class represents the consumer thread in a producer/consumer relationship. (2 of 3)

```
15
16     // read sharedLocation's value four times and sum the values
17 public void run()
18 {
19     int sum = 0;
20
21     // alternate between sleeping and getting Buffer value
22     for ( int count = 1; count <= 4; ++count )
23     {
24         // sleep 0-3 seconds, read Buffer value and add to sum
25         try
26         {
27             Thread.sleep( ( int )( Math.random() * 3001 ) );
28             sum += sharedLocation.get();
29         }
30     }
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.3 Consumer class represents the consumer thread in a producer/consumer relationship. (3 of 3)

```
31         // if sleeping thread interrupted, print stack trace
32     catch ( InterruptedException exception )
33     {
34         exception.printStackTrace();
35     }
36 } // end for
37
38 System.err.println( getName() + " read values totaling: "
39                     + sum + ".\nTerminating " + getName() + ".");
40
41 } // end method run
42
43 } // end class Consumer
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.4 `UnsynchronizedBuffer` class maintains the shared integer that is accessed by a producer thread and a consumer thread via methods `set` and `get`. (1 of 2)

```
1 // Fig. 5.4: UnsynchronizedBuffer.java
2 // UnsynchronizedBuffer represents a single shared integer.
3
4 public class UnsynchronizedBuffer implements Buffer
5 {
6     private int buffer = -1; // shared by Producer and Consumer
7
8     // place value into buffer
9     public void set( int value )
10    {
11        System.err.println( Thread.currentThread().getName() +
12                            " writes " + value );
13
14        buffer = value;
15    } // end method set
16
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.4 UnsynchronizedBuffer class maintains the shared integer that is accessed by a producer thread and a consumer thread via methods set and get. (2 of 2)

```
17     // return value from buffer
18     public int get()
19     {
20         System.err.println( Thread.currentThread().getName() +
21             " reads " + buffer );
22
23         return buffer;
24     } // end method get
25
26 } // end class UnsynchronizedBuffer
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.5 SharedBuffer class enables threads to modify a shared object without synchronization. (1 of 4)

```
1 // Fig. 5.5: SharedBufferTest.java
2 // SharedBufferTest creates producer and consumer threads.
3
4 public class SharedBufferTest
5 {
6     public static void main( String [] args )
7     {
8         // create shared object used by threads
9         Buffer sharedLocation = new UnsynchronizedBuffer();
10
11         // create producer and consumer objects
12         Producer producer = new Producer( sharedLocation );
13         Consumer consumer = new Consumer( sharedLocation );
14
15         producer.start(); // start producer thread
16         consumer.start(); // start consumer thread
17
18     } // end main
19
20 } // end class SharedCell
```



5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.5 SharedBuffer class enables threads to modify a shared object without synchronization. (2 of 4)

Sample Output 1:

```
Consumer reads -1
Producer writes 1
Consumer reads 1
Consumer reads 1
Consumer reads 1
Consumer read values totaling: 2.
Terminating Consumer.
Producer writes 2
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.5 SharedBuffer class enables threads to modify a shared object without synchronization. (3 of 4)

Sample Output 2:

```
Producer writes 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer reads 4
Consumer read values totaling: 13.
Terminating Consumer.
```

5.2.1 Java Multithreading Case Study, Part II: A Producer/Consumer Relationship in Java

Figure 5.5 SharedBuffer class enables threads to modify a shared object without synchronization. (4 of 4)

Sample Output 3:

```
Producer writes 1
Consumer reads 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer read values totaling: 10.
Terminating Consumer.
```

5.2.2 Critical Sections

- Most code is safe to run concurrently
- Sections where shared data is modified must be protected
 - Known as critical sections
 - Only one thread can be in its critical section at once
 - Must be careful to avoid infinite loops and blocking inside a critical section

5.2.3 Mutual Exclusion Primitives

- Indicate when critical data is about to be accessed
 - Mechanisms are normally provided by programming language or libraries
 - Delimit beginning and end of critical section
 - `enterMutualExclusion`
 - `exitMutualExclusion`

5.3 Implementing Mutual Exclusion Primitives

- Common properties of mutual exclusion primitives
 - Each mutual exclusion machine language instruction is executed indivisibly
 - Cannot make assumptions about relative speed of thread execution
 - Thread not in its critical section cannot block other threads from entering their critical sections
 - Thread may not be indefinitely postponed from entering its critical section

5.4.1 Dekker's Algorithm

- First version of Dekker's algorithm
 - Succeeds in enforcing mutual exclusion
 - Uses variable to control which thread can execute
 - Constantly tests whether critical section is available
 - Busy waiting
 - Wastes significant processor time
 - Problem known as lockstep synchronization
 - Each thread can execute only in strict alternation

5.4.1 Dekker's Algorithm

Figure 5.6 Mutual exclusion implementation – version 1 (1 of 2).

```
1  System:  
2  
3  int threadNumber = 1;  
4  
5  startThreads(); // initialize and launch both threads  
6  
7  Thread T1:  
8  
9  void main() {  
10  
11      while ( !done )  
12      {  
13          while ( threadNumber == 2 ); // enterMutualExclusion  
14  
15          // critical section code  
16  
17          threadNumber = 2; // exitMutualExclusion  
18  
19          // code outside critical section  
20  
21      } // end outer while
```

5.4.1 Dekker's Algorithm

Figure 5.6 Mutual exclusion implementation – version 1 (2 of 2).

```
22
23 } // end Thread T1
24
25 Thread T2:
26
27 void main() {
28
29     while ( !done )
30     {
31         while ( threadNumber == 1 ); // enterMutualExclusion
32
33         // critical section code
34
35         threadNumber = 1; // exitMutualExclusion
36
37         // code outside critical section
38
39     } // end outer while
40
41 } // end Thread T2
```

5.4.1 Dekker's Algorithm

- Second version
 - Removes lockstep synchronization
 - Violates mutual exclusion
 - Thread could be preempted while updating flag variable
 - Not an appropriate solution

5.4.1 Dekker's Algorithm

Figure 5.7 Mutual exclusion implementation – version 2 (1 of 3).

```
1 System:
2
3 boolean t1Inside = false;
4 boolean t2Inside = false;
5
6 startThreads(); // initialize and launch both threads
7
8 Thread T1:
9
10 void main() {
11
12     while ( !done )
13     {
14         while ( t2Inside ); // enterMutualExclusion
15
16         t1Inside = true; // enterMutualExclusion
17
18         // critical section code
19
20         t1Inside = false; // exitMutualExclusion
21 }
```



5.4.1 Dekker's Algorithm

Figure 5.7 Mutual exclusion implementation – version 2 (2 of 3).

```
22      // code outside critical section
23
24  } // end outer while
25
26 } // end Thread T1
27
28 Thread T2:
29
30 void main() {
31
32     while ( !done )
33     {
34         while ( t1Inside ); // enterMutualExclusion
35
36         t2Inside = true; // enterMutualExclusion
37
38         // critical section code
```

5.4.1 Dekker's Algorithm

Figure 5.7 Mutual exclusion implementation – version 2 (3 of 3).

```
39      t2Inside = false; // exitMutualExclusion
40
41      // code outside critical section
42
43  } // end outer while
44
45 } // end Thread T2
```

5.4.1 Dekker's Algorithm

- Third version
 - Set critical section flag before entering critical section test
 - Once again guarantees mutual exclusion
 - Introduces possibility of deadlock
 - Both threads could set flag simultaneously
 - Neither would ever be able to break out of loop
 - Not a solution to the mutual exclusion problem

5.4.1 Dekker's Algorithm

Figure 5.8 Mutual exclusion implementation – version 3 (1 of 2).

```
1  System:
2
3  boolean t1WantsToEnter = false;
4  boolean t2WantsToEnter = false;
5
6  startThreads(); // initialize and launch both threads
7
8  Thread T1:
9
10 void main()
11 {
12     while ( !done )
13     {
14         t1WantsToEnter = true; // enterMutualExclusion
15
16         while ( t2WantsToEnter ); // enterMutualExclusion
17
18         // critical section code
19
20         t1WantsToEnter = false; // exitMutualExclusion
21
22         // code outside critical section
23 }
```

5.4.1 Dekker's Algorithm

Figure 5.8 Mutual exclusion implementation – version 3 (2 of 2).

```
24     } // end outer while
25
26 } // end Thread T1
27
28 Thread T2:
29
30 void main()
31 {
32     while ( !done )
33     {
34         t2WantsToEnter = true; // enterMutualExclusion
35
36         while ( t1WantsToEnter ); // enterMutualExclusion
37
38         // critical section code
39
40         t2WantsToEnter = false; // exitMutualExclusion
41
42         // code outside critical section
43
44     } // end outer while
45
46 } // end Thread T2
```

5.4.1 Dekker's Algorithm

- Fourth version
 - Sets flag to false for small periods of time to yield control
 - Solves previous problems, introduces indefinite postponement
 - Both threads could set flags to same values at same time
 - Would require both threads to execute in tandem (unlikely but possible)
 - Unacceptable in mission- or business-critical systems

5.4.1 Dekker's Algorithm

Figure 5.9 Mutual exclusion implementation – version 4 (1 of 4).

```
1 System:
2
3 boolean t1WantsToEnter = false;
4 boolean t2WantsToEnter = false;
5
6 startThreads(); // initialize and launch both threads
7
8 Thread T1:
9
10 void main()
11 {
12     while ( !done )
13     {
14         t1WantsToEnter = true; // enterMutualExclusion
15     }
```

5.4.1 Dekker's Algorithm

Figure 5.9 Mutual exclusion implementation – version 4 (2 of 4).

```
16     while ( t2WantsToEnter ) // enterMutualExclusion
17     {
18         t1WantsToEnter = false; // enterMutualExclusion
19
20         // wait for small, random amount of time
21
22         t1WantsToEnter = true;
23     } // end while
24
25     // critical section code
26
27     t1WantsToEnter = false; // exitMutualExclusion
28
29     // code outside critical section
30
31 } // end outer while
32
```

5.4.1 Dekker's Algorithm

Figure 5.9 Mutual exclusion implementation – version 4 (3 of 4).

```
33 } // end Thread T1
34
35 Thread T2:
36
37 void main()
38 {
39     while ( !done )
40     {
41         t2WantsToEnter = true; // enterMutualExclusion
42
43         while ( t1WantsToEnter ) // enterMutualExclusion
44         {
45             t2WantsToEnter = false; // enterMutualExclusion
46
47             // wait for small, random amount of time
48
49             t2WantsToEnter = true;
50     } // end while
51 }
```



5.4.1 Dekker's Algorithm

Figure 5.9 Mutual exclusion implementation – version 4 (4 of 4).

```
52     // critical section code
53
54     t2WantsToEnter = false; // exitMutualExclusion
55
56     // code outside critical section
57
58 } // end outer while
59
60 } // end Thread T2
```

5.4.1 Dekker's Algorithm

- Dekker's Algorithm
 - Proper solution
 - Uses notion of favored threads to determine entry into critical sections
 - Resolves conflict over which thread should execute first
 - Each thread temporarily unsets critical section request flag
 - Favored status alternates between threads
 - Guarantees mutual exclusion
 - Avoids previous problems of deadlock, indefinite postponement

5.4.1 Dekker's Algorithm

Figure 5.10 Dekker's Algorithm for mutual exclusion. (1 of 4)

```
1  System:
2
3  int favoredThread = 1;
4  boolean t1WantsToEnter = false;
5  boolean t2WantsToEnter = false;
6
7  startThreads(); // initialize and launch both threads
8
9  Thread T1:
10
11 void main()
12 {
13     while ( !done )
14     {
15         t1WantsToEnter = true;
16     }
```

5.4.1 Dekker's Algorithm

Figure 5.10 Dekker's Algorithm for mutual exclusion. (2 of 4)

```
17     while ( t2WantsToEnter )
18     {
19         if ( favoredThread == 2 )
20         {
21             t1WantsToEnter = false;
22             while ( favoredThread == 2 ); // busy wait
23             t1WantsToEnter = true;
24         } // end if
25
26     } // end while
27
28     // critical section code
29
30     favoredThread = 2;
31     t1WantsToEnter = false;
32
33     // code outside critical section
34
35 } // end outer while
36
37 } // end Thread T1
```

5.4.1 Dekker's Algorithm

Figure 5.10 Dekker's Algorithm for mutual exclusion. (3 of 4)

```
38
39 Thread T2:
40
41 void main()
42 {
43     while ( !done )
44     {
45         t2WantsToEnter = true;
46
47         while ( t1WantsToEnter )
48         {
49             if ( favoredThread == 1 )
50             {
51                 t2WantsToEnter = false;
52                 while ( favoredThread == 1 ); // busy wait
53                 t2WantsToEnter = true;
54             } // end if
55
56     } // end while
```

5.4.1 Dekker's Algorithm

Figure 5.10 Dekker's Algorithm for mutual exclusion. (4 of 4)

```
57      // critical section code
58
59      favoredThread = 1;
60      t2WantsToEnter = false;
61
62      // code outside critical section
63
64  } // end outer while
65
66
67 } // end Thread T2
```

5.4.2 Peterson's Algorithm

- Less complicated than Dekker's Algorithm
 - Still uses busy waiting, favored threads
 - Requires fewer steps to perform mutual exclusion primitives
 - Easier to demonstrate its correctness
 - Does not exhibit indefinite postponement or deadlock

5.4.2 Peterson's Algorithm

Figure 5.11 Peterson's Algorithm for mutual exclusion. (1 of 3)

```
1 System:  
2  
3 int favoredThread = 1;  
4 boolean t1WantsToEnter = false;  
5 boolean t2WantsToEnter = false;  
6  
7 startThreads(); // initialize and launch both threads
```

5.4.2 Peterson's Algorithm

Figure 5.11 Peterson's Algorithm for mutual exclusion. (2 of 3)

```
8
9 Thread T1:
10
11 void main()
12 {
13     while ( !done )
14     {
15         t1WantsToEnter = true;
16         favoredThread = 2;
17
18         while ( t2WantsToEnter && favoredThread == 2 );
19
20         // critical section code
21
22         t1WantsToEnter = false;
23
24         // code outside critical section
25
26     } // end while
27
28 } // end Thread T1
29
```

5.4.2 Peterson's Algorithm

Figure 5.11 Peterson's Algorithm for mutual exclusion. (3 of 3)

```
30 Thread T2:
31
32 void main()
33 {
34     while ( !done )
35     {
36         t2WantsToEnter = true;
37         favoredThread = 1;
38
39         while ( t1WantsToEnter && favoredThread == 1 );
40
41         // critical section code
42
43         t2WantsToEnter = false;
44
45         // code outside critical section
46
47     } // end while
48
49 } // end Thread T2
```

5.4.3 N-Thread Mutual Exclusion: Lamport's Bakery Algorithm

- Applicable to any number of threads
 - Creates a queue of waiting threads by distributing numbered “tickets”
 - Each thread executes when its ticket’s number is the lowest of all threads
 - Unlike Dekker’s and Peterson’s Algorithms, the Bakery Algorithm works in multiprocessor systems and for n threads
 - Relatively simple to understand due to its real-world analog

5.4.3 N-Thread Mutual Exclusion: Lamport's Bakery Algorithm

Figure 5.12 Lamport's Bakery Algorithm. (1 of 3)

```
1 System:  
2  
3 // array that records which threads are taking a ticket  
4 boolean choosing[n];  
5  
6 // value of the ticket for each thread initialized to 0  
7 int ticket[n];  
8  
9 startThreads(); // initialize and launch all threads  
10
```

5.4.3 N-Thread Mutual Exclusion: Lamport's Bakery Algorithm

Figure 5.12 Lamport's Bakery Algorithm. (2 of 3)

```
11 Thread Tx
12
13 void main()
14 {
15     x = threadNumber(); // store current thread number
16
17     while ( !done )
18     {
19         // take a ticket
20         choosing[x] = true; // begin ticket selection process
21         ticket[x] = maxValue( ticket ) + 1;
22         choosing[x] = false; // end ticket selection process
23
24         // wait for number to be called by comparing current
25         // ticket value to other thread's ticket value
26         for ( int i = 0; i < n; i++ )
27         {
28             if ( i == x )
29             {
30                 continue; // no need to check own ticket
31             } // end if
32 }
```



5.4.3 N-Thread Mutual Exclusion: Lamport's Bakery Algorithm

Figure 5.12 Lamport's Bakery Algorithm. (3 of 3)

```
33      // busy wait while thread[i] is choosing
34      while ( choosing[i] != false );
35
36      // busy wait until current ticket value is lowest
37      while ( ticket[i] != 0 && ticket[i] < ticket[x] );
38
39      // tie-breaker code favors smaller thread number
40      if ( ticket[i] == ticket[x] && i < x )
41
42          // loop until thread[i] leaves its critical section
43          while ( ticket[i] != 0 ); // busy wait
44      } // end for
45
46      // critical section code
47
48      ticket[x] = 0; // exitMutualExclusion
49
50      // code outside critical section
51
52  } // end while
53
54 } // end Thread TX
```

5.5 Hardware Solutions to the Mutual Exclusion Problem

- Implementing mutual exclusion in hardware
 - Can improve performance
 - Can decrease development time
 - No need to implement complex software mutual exclusion solutions like Lamport's Algorithm

5.5.1 Disabling Interrupts

- Disabling interrupts
 - Works only on uniprocessor systems
 - Prevents the currently executing thread from being preempted
 - Could result in deadlock
 - For example, thread waiting for I/O event in critical section
 - Technique is used rarely

5.5.2 Test-and-Set Instruction

- Use a machine-language instruction to ensure that mutual exclusion primitives are performed indivisibly
 - Such instructions are called atomic
 - Machine-language instructions do not ensure mutual exclusion alone—the software must properly use them
 - For example, programmers must incorporate favored threads to avoid indefinite postponement
 - Used to simplify software algorithms rather than replace them
- Test-and-set instruction
 - `testAndSet(a, b)` copies the value of `b` to `a`, then sets `b` to `true`
 - Example of an atomic read-modify-write (RMW) cycle

5.5.2 Test-and-Set Instruction

Figure 5.13 testAndSet instruction for mutual exclusion. (1 of 3)

```
1  System:  
2  
3  boolean occupied = false;  
4  
5  startThreads(); // initialize and launch both threads  
6  
7  Thread T1:  
8  
9  void main()  
10 {  
11     boolean p1MustWait = true;  
12  
13     while ( !done )  
14     {  
15         while ( p1MustWait )  
16         {  
17             testAndSet( p1MustWait, occupied );  
18         }  
19     }
```

5.5.2 Test-and-Set Instruction

Figure 5.13 testAndSet instruction for mutual exclusion. (2 of 3)

```
20      // critical section code
21
22      p1MustWait = true;
23      occupied = false;
24
25      // code outside critical section
26
27  } // end while
28
29 } // end Thread T1
30
31 Thread T2:
32
```

5.5.2 Test-and-Set Instruction

Figure 5.13 testAndSet instruction for mutual exclusion. (3 of 3)

```
33 void main()
34 {
35     boolean p2MustWait = true;
36
37     while ( !done )
38     {
39         while ( p2MustWait )
40         {
41             testAndSet( p2MustWait, occupied );
42         }
43
44         // critical section code
45
46         p2MustWait = true;
47         occupied = false;
48
49         // code outside critical section
50
51     } // end while
52
53 } // end Thread T2
```

5.5.3 Swap Instruction

- `swap(a, b)` exchanges the values of `a` and `b` atomically
- Similar in functionality to test-and-set
 - `swap` is more commonly implemented on multiple architectures

5.5.3 Swap Instruction

Figure 5.14 swap instruction for mutual exclusion. (1 of 3)

```
1 System:
2
3 boolean occupied = false;
4
5 startThreads(); // initialize and launch both threads
6
7 Thread T1:
8
9 void main()
10 {
11     boolean p1MustWait = true;
12 }
```

5.5.3 Swap Instruction

Figure 5.14 swap instruction for mutual exclusion. (2 of 3)

```
13     while ( !done )
14     {
15         do
16         {
17             swap( p1MustWait, occupied );
18         } while ( p1MustWait );
19
20         // critical section code
21
22         p1MustWait = true;
23         occupied = false;
24
25         // code outside critical section
26
27     } // end while
28
29 } // end Thread T1
30
```

5.5.3 Swap Instruction

Figure 5.14 swap instruction for mutual exclusion. (3 of 3)

```
31 Thread T2:
32
33 void main()
34 {
35     boolean p2MustWait = true;
36
37     while ( !done )
38     {
39         do
40         {
41             swap( p2MustWait, occupied );
42         } while ( p2MustWait );
43
44         // critical section code
45
46         p2MustWait = true;
47         occupied = false;
48
49         // code outside critical section
50
51     } // end while
52
53 } // end Thread T2
```

5.6 Semaphores

- **Semaphores**
 - Software construct that can be used to enforce mutual exclusion
 - Contains a protected variable
 - Can be accessed only via wait and signal commands
 - Also called P and V operations, respectively

5.6.1 Mutual Exclusion with Semaphores

- Binary semaphore: allow only one thread in its critical section at once
 - Wait operation
 - If no threads are waiting, allow thread into its critical section
 - Decrement protected variable (to 0 in this case)
 - Otherwise place in waiting queue
 - Signal operation
 - Indicate that thread is outside its critical section
 - Increment protected variable (from 0 to 1)
 - A waiting thread (if there is one) may now enter

5.6.1 Mutual Exclusion with Semaphores

Figure 5.15 Mutual exclusion with semaphores.

```
1  System:
2
3  // create semaphore and initialize value to 1
4  Semaphore occupied = new Semaphore(1);
5
6  startThreads(); // initialize and launch both threads
7
8  Thread Tx:
9
10 void main()
11 {
12     while ( !done )
13     {
14         P( occupied ); // wait
15
16         // critical section code
17
18         V( occupied ); // signal
19
20         // code outside critical section
21     } // end while
22 } // Thread TX
```

5.6.2 Thread Synchronization with Semaphores

- Semaphores can be used to notify other threads that events have occurred
 - Producer-consumer relationship
 - Producer enters its critical section to produce value
 - Consumer is blocked until producer finishes
 - Consumer enters its critical section to read value
 - Producer cannot update value until it is consumed
 - Semaphores offer a clear, easy-to-implement solution to this problem

5.6.2 Thread Synchronization with Semaphores

Figure 5.16 Producer/consumer relationship implemented with semaphores. (1 of 2)

```
1  System:
2  // semaphores that synchronize access to sharedValue
3  Semaphore valueProduced = new Semaphore(0);
4  Semaphore valueConsumed = new Semaphore(1);
5  int sharedValue; // variable shared by producer and consumer
6
7  startThreads(); // initialize and launch both threads
8
9  Producer Thread:
10 void main()
11 {
12     int nextValueProduced; // variable to store value produced
13
14     while ( !done )
15     {
16         nextValueProduced = generateTheValue(); // produce value
17         P( valueConsumed ); // wait until value is consumed
18         sharedValue = nextValueProduced; // critical section
19         V( valueProduced ); // signal that value has been produced
20
21     } // end while
22
23 }
24 } // end producer thread
```

5.6.2 Thread Synchronization with Semaphores

Figure 5.16 Producer/consumer relationship implemented with semaphores. (2 of 2)

```
25
26 Consumer Thread:
27
28 void main()
29 {
30     int nextValue; // variable to store value consumed
31
32     while ( !done )
33     {
34         P( valueProduced ); // wait until value is produced
35         nextValueConsumed = sharedValue; // critical section
36         V( valueConsumed ); // signal that value has been consumed
37         processTheValue( nextValueConsumed ); // process the value
38
39     } // end while
40
41 } // end consumer thread
```

5.6.3 Counting Semaphores

- Counting semaphores
 - Initialized with values greater than one
 - Can be used to control access to a pool of identical resources
 - Decrement the semaphore's counter when taking resource from pool
 - Increment the semaphore's counter when returning it to pool
 - If no resources are available, thread is blocked until a resource becomes available

5.6.4 Implementing Semaphores

- Semaphores can be implemented at application or kernel level
 - Application level: typically implemented by busy waiting
 - Inefficient
 - Kernel implementations can avoid busy waiting
 - Block waiting threads until they are ready
 - Kernel implementations can disable interrupts
 - Guarantee exclusive semaphore access
 - Must be careful to avoid poor performance and deadlock
 - Implementations for multiprocessor systems must use a more sophisticated approach

Chapter 6 – Concurrent Programming

Outline

- 6.1 **Introduction**
- 6.2 **Monitors**
 - 6.2.1 **Condition Variables**
 - 6.2.2 **Simple Resource Allocation with Monitors**
 - 6.2.3 **Monitor Example: Circular Buffer**
 - 6.2.4 **Monitor Example: Readers and Writers**
- 6.3 **Java Monitors**
- 6.4 **Java Multithreading Case Study, Part III:
Producer/Consumer Relationship in Java**
- 6.5 **Java Multithreading Case Study, Part IV:
Circular Buffer in Java**

Objectives

- After reading this chapter, you should understand:
 - how monitors synchronize access to data.
 - how condition variables are used with monitors.
 - solutions for classic problems in concurrent programming such as readers and writers and circular buffer.
 - Java monitors.
 - remote procedure calls.

6.2.1 Introduction

- Recent interest in concurrent programming languages
 - Naturally express solutions to inherently parallel problems
 - Due to proliferation of multiprocessing systems, distributed systems and massively parallel architectures
 - More complex than standard programs
 - More time required to write, test and debug

6.2 Monitors

- Monitor
 - Contains data and procedures needed to allocate shared resources
 - Accessible only within the monitor
 - No way for threads outside monitor to access monitor data

6.2 Monitors

- Resource allocation using monitors
 - Thread must call monitor entry routine
 - Mutual exclusion is rigidly enforced at monitor boundary
 - A thread that tries to enter monitor when it is in use must wait

6.2 Monitors

- Threads return resources through monitors as well
 - Monitor entry routine calls `signal`
 - Alerts one waiting thread to acquire resource and enter monitor
 - Higher priority given to waiting threads than ones newly arrived
 - Avoids indefinite postponement

6.2.1 Condition Variables

- Before a thread can reenter the monitor, the thread calling `signal` must first exit monitor
 - Signal-and-exit monitor
 - Requires thread to exit the monitor immediately upon signaling
- Signal-and-continue monitor
 - Allows thread inside monitor to signal that the monitor will soon become available
 - Still maintain lock on the monitor until thread exits monitor
 - Thread can exit monitor by waiting on a condition variable or by completing execution of code protected by monitor

6.2.2 Simple Resource Allocation with Monitors

- Thread inside monitor may need to wait outside until another thread performs an action inside monitor
- Monitor associates separate condition variable with distinct situation that might cause thread to wait
 - Every condition variable has an associated queue

6.2.2 Simple Resource Allocation with Monitors

Figure 6.1 Simple resource allocation with a monitor in pseudocode.

```
1 // Fig. 6.1: Resource allocator monitor
2
3 // monitor initialization (performed only once)
4 boolean inUse = false; // simple state variable
5 Condition available; // condition variable
6
7 // request resource
8 monitorEntry void getResource()
9 {
10    if ( inUse ) // is resource in use?
11    {
12        wait( available ); // wait until available is signaled
13    } // end if
14
15    inUse = true; // indicate resource is now in use
16
17 } // end getResource
18
19 // return resource
20 monitorEntry void returnResource()
21 {
22    inUse = false; // indicate resource is not in use
23    signal( available ); // signal a waiting thread to proceed
24
25 } // end returnResource
```

6.2.3 Monitor Example: Circular Buffer

- Circular buffer implementation of the solution to producer/consumer problem
 - Producer deposits data in successive elements of array
 - Consumer removes the elements in the order in which they were deposited (FIFO)
 - Producer can be several items ahead of consumer
 - If the producer fills last element of array, it must “wrap around” and begin depositing data in the first element of array

6.2.3 Monitor Example: Circular Buffer

- Due to the fixed size of a circular buffer
 - Producer will occasionally find all array elements full, in which case the producer must wait until consumer empties an array element
 - Consumer will occasionally find all array elements empty, in which case the consumer must wait until producer deposits data into an array element

6.2.3 Monitor Example: Circular Buffer

Figure 6.2 Monitor pseudocode implementation of a circular buffer. (Part 1 of 2.)

```
1 // Fig. 6.2: Circular buffer monitor
2
3 char circularBuffer[] = new char[ BUFFER_SIZE ]; // buffer
4 int writerPosition = 0; // next slot to write to
5 int readerPosition = 0; // next slot to read from
6 int occupiedSlots = 0; // number of slots with data
7 Condition hasData; // condition variable
8 Condition hasSpace; // condition variable
9
10 // monitor entry called by producer to write data
11 monitorEntry void putChar( char slotData )
12 {
13     // wait on condition variable hasSpace if buffer is full
14     if ( occupiedSlots == BUFFER_SIZE )
15     {
16         wait( hasSpace ); // wait until hasSpace is signaled
17     } // end if
18
19     // write character to buffer
20     circularBuffer[ writerPosition ] = slotData;
21     ++occupiedSlots; // one more slot has data
22     writerPosition = (writerPosition + 1) % BUFFER_SIZE;
23     signal( hasData ); // signal that data is available
24 } // end putChar
25
```

6.2.3 Monitor Example: Circular Buffer

Figure 6.2 Monitor pseudocode implementation of a circular buffer. (Part 2 of 2.)

```
26 // monitor entry called by consumer to read data
27 monitorEntry void getChar( outputParameter slotData )
28 {
29     // wait on condition variable hasData if the buffer is empty
30     if ( occupiedSlots == 0 )
31     {
32         wait( hasData ); // wait until hasData is signaled
33     } // end if
34
35     // read character from buffer into output parameter slotData
36     slotData = circularBuffer[ readPosition ];
37     occupiedSlots--; // one fewer slots has data
38     readerPosition = (readerPosition + 1) % BUFFER_SIZE;
39     signal( hasSpace ); // signal that character has been read
40 } // end getChar
```

6.2.4 Monitor Example: Readers and Writers

- Readers
 - Read data, but do not change contents of data
 - Multiple readers can access the shared data at once
 - When a new reader calls a monitor entry routine, it is allowed to proceed as long as no thread is writing and no writer thread is waiting to write
 - After the reader finishes reading, it calls a monitor entry routine to signal the next waiting reader to proceed, causing a “chain reaction”
- Writers
 - Can modify data
 - Must have exclusive access

6.2.4 Monitor Example: Readers and Writers

Figure 6.3 Monitor pseudocode for solving the readers and writers problem. (Part 1 of 3.)

```
1 // Fig. 6.3: Readers/writers problem
2
3 int readers = 0; // number of readers
4 boolean writeLock = false; // true if a writer is writing
5 Condition canWrite; // condition variable
6 Condition canRead; // condition variable
7
8 // monitor entry called before performing read
9 monitorEntry void beginRead()
10 {
11     // wait outside monitor if writer is currently writing or if
12     // writers are currently waiting to write
13     if ( writeLock || queue( canWrite ) )
14     {
15         wait( canRead ); // wait until reading is allowed
16     } // end if
17
18     ++readers; // there is another reader
19
20     signal( canRead ); // allow waiting readers to proceed
21 } // end beginRead
22
```

6.2.4 Monitor Example: Readers and Writers

Figure 6.3 Monitor pseudocode for solving the readers and writers problem. (Part 2 of 3.)

```
23 // monitor entry called after reading
24 monitorEntry void endRead()
25 {
26     --readers; // there are one fewer readers
27
28     // if no more readers are reading, allow a writer to write
29     if ( readers == 0 )
30     {
31         signal ( canWrite ); // allow a writer to proceed
32     } // end if
33
34 } // end endRead
35
36 // monitor entry called before performing write
37 monitorEntry void beginWrite()
38 {
39     // wait if readers are reading or if a writer is writing
40     if ( readers > 0 || writeLock )
41     {
42         wait( canWrite ); // wait until writing is allowed
43     } // end if
44 }
```

6.2.4 Monitor Example: Readers and Writers

Figure 6.3 Monitor pseudocode for solving the readers and writers problem. (Part 3 of 3.)

```
45     writeLock = true; // lock out all readers and writers
46 } // end beginWrite
47
48 // monitor entry called after performing write
49 monitorEntry void endWrite()
50 {
51     writeLock = false; // release lock
52
53     // if a reader is waiting to enter, signal a reader
54     if ( queue( canRead ) )
55     {
56         signal( canRead ); // cascade in waiting readers
57     } // end if
58     else // signal a writer if no readers are waiting
59     {
60         signal( canWrite ); // one waiting writer can proceed
61     } // end else
62
63 } // end endWrite
```

6.3 Java Monitors

- **Java monitors**
 - Primary mechanism for providing mutual exclusion and synchronization in multithreaded Java applications
 - Signal-and-continue monitors
 - Allow a thread to signal that the monitor will soon become available
 - Maintain a lock on monitor until thread exits monitor
- **Keyword synchronized**
 - Imposes mutual exclusion on an object in Java

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

- Java `wait` method
 - Calling thread releases lock on monitor
 - Waits for condition variable
 - After calling `wait`, thread is placed in wait set
 - Thread remains in wait set until signaled by another thread
- Condition variable is implicit in Java
 - A thread may be signaled, reenter the monitor and find that the condition on which it waited has not been met

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.4 synchronizedBuffer synchronizes access to a shared integer. (Part 1 of 4.)

```
1 // Fig. 6.4: SynchronizedBuffer.java
2 // SynchronizedBuffer synchronizes access to a shared integer.
3
4 public class SynchronizedBuffer implements Buffer
5 {
6     private int buffer = -1; // shared by producer and consumer
7     private int occupiedBuffers = 0; // counts occupied buffers
8
9     // place value into buffer
10    public synchronized void set( int value )
11    {
12        // for display, get name of thread that called this method
13        String name = Thread.currentThread().getName();
14
15        // while no empty buffers, place thread in waiting state
16        while ( occupiedBuffers == 1 )
17        {
18            // output thread and buffer information, then wait
19            try
20            {
21                System.err.println( name + " tries to write." );
22                displayState( "Buffer full. " + name + " waits." );
23                wait(); // wait until buffer is empty
24            } // end try
25        }
```



6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.4 synchronizedBuffer synchronizes access to a shared integer. (Part 2 of 4.)

```
26          // if waiting thread interrupted, print stack trace
27      catch ( InterruptedException exception )
28      {
29          exception.printStackTrace();
30      } // end catch
31
32 } // end while
33
34 buffer = value; // set new buffer value
35
36 // indicate producer cannot store another value
37 // until consumer retrieves current buffer value
38 ++occupiedBuffers;
39
40 displayState( name + " writes " + buffer );
41
42 notify(); // tell waiting thread to enter ready state
43 } // end method set; releases lock on SynchronizedBuffer
44
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.4 SynchronizedBuffer synchronizes access to a shared integer. (Part 3 of 4.)

```
44
45      // return value from buffer
46      public synchronized int get()
47      {
48          // for display, get name of thread that called this method
49          String name = Thread.currentThread().getName();
50
51          // while no data to read, place thread in waiting state
52          while ( occupiedBuffers == 0 )
53          {
54              // output thread and buffer information, then wait
55              try
56              {
57                  System.err.println( name + " tries to read." );
58                  displayState( "Buffer empty. " + name + " waits." );
59
60                  wait(); // wait until buffer contains new values
61              } // end try
62
63              // if waiting thread interrupted, print stack trace
64              catch ( InterruptedException exception )
65              {
66                  exception.printStackTrace();
67              } // end catch
68          } // end while
69
```



6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.4 synchronizedBuffer synchronizes access to a shared integer. (Part 4 of 4.)

```
70      // indicate that producer can store another value
71      // because consumer just retrieved buffer value
72      --occupiedBuffers;
73
74      displayState( name + " reads " + buffer );
75
76      notify(); // tell waiting thread to become ready
77
78      return buffer;
79  } // end method get; releases lock on SynchronizedBuffer
80
81  // display current operation and buffer state
82  public void displayState( String operation )
83  {
84      StringBuffer outputLine = new StringBuffer( operation );
85      outputLine.setLength( 40 );
86      outputLine.append( buffer + "\t\t" + occupiedBuffers );
87      System.err.println( outputLine );
88      System.err.println();
89  } // end method displayState
90
91 } // end class SynchronizedBuffer
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 1 of 8.)

```
1 // Fig. 6.5: SharedBufferTest2.java
2 // SharedBufferTest2 creates producer and consumer threads.
3
4 public class SharedBufferTest2
5 {
6     public static void main( String [] args )
7     {
8         // Create shared object used by threads
9         SynchronizedBuffer sharedLocation = new SynchronizedBuffer();
10
11        // Display column heads for output
12        StringBuffer columnHeads =
13            new StringBuffer( "Operation" );
14        columnHeads.setLength( 40 );
15        columnHeads.append( "Buffer\t\tOccupied Count" );
16        System.err.println( columnHeads );
17        System.err.println();
18        sharedLocation.displayState( "Initial State" );
19
20        // Create producer and consumer objects
21        Producer producer = new Producer( sharedLocation );
22        Consumer consumer = new Consumer( sharedLocation );
23
24        producer.start(); // Start producer thread
```



6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 2 of 8.)

```
25         consumer.start(); // start consumer thread
26
27     } // end main
28
29 } // end class SharedBufferTest2
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 3 of 8.)

Sample Output 1:

Operation	Buffer	Occupied Count
Initial State	-1	0
Consumer tries to read. Buffer empty. Consumer waits.	-1	0
Producer writes 1	1	1
Consumer reads 1	1	0
Consumer tries to read. Buffer empty. Consumer waits.	1	0
Producer writes 2	2	1
Consumer reads 2	2	0
Producer writes 3	3	1

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 4 of 8.)

Consumer reads 3	3	0
Consumer tries to read.		
Buffer empty. Consumer waits.	3	0
Producer writes 4	4	1
Consumer reads 4	4	0
Producer done producing.		
Terminating Producer.		
Consumer read values totaling: 10.		
Terminating Consumer.		

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 5 of 8.)

Sample Output 2:

Operation	Buffer	Occupied Count
Initial State	-1	0
Consumer tries to read. Buffer empty. Consumer waits.	-1	0
Producer writes 1	1	1
Consumer reads 1	1	0
Producer writes 2	2	1
Producer tries to write. Buffer full. Producer waits.	2	1
Consumer reads 2	2	0
Producer writes 3	3	1

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 6 of 8.)

```
Consumer reads 3          3          0
Producer writes 4         4          1
Producer done producing.
Terminating Producer.
Consumer reads 4          4          0
Consumer read values totaling: 10.
Terminating Consumer.
```

Sample Output 3:

Operation	Buffer	Occupied Count
Initial State	-1	0
Producer writes 1	1	1

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 7 of 8.)

Sample Output 3 (Cont.).

Operation	Buffer	Occupied Count
Initial State	-1	0
Producer writes 1	1	1
Consumer reads 1	1	0
Producer writes 2	2	1
Consumer reads 2	2	0
Producer writes 3	3	1
Consumer reads 3	3	0
Producer writes 4	4	1
Producer done producing. Terminating Producer.		

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Figure 6.5 Threads modifying a shared object with synchronization. (Part 8 of 8.)

```
Consumer reads 4          4          0
Consumer read values totaling: 10.
Terminating Consumer.
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

- Threads issue signals via methods `notify` or `notifyAll`
 - `notify` method
 - wakes one thread waiting to enter the monitor
 - `notifyAll` method
 - Wakes all waiting threads

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 SynchronizedBuffer controls access to a shared array of integers. (Part 1 of 7.)

```
1 // Fig. 6.6: CircularBuffer.java
2 // CircularBuffer synchronizes access to an array of
3 // shared buffers.

4
5 public class CircularBuffer implements Buffer
6 {
7     // each array element is a buffer
8     private int buffers[] = { -1, -1, -1 };
9
10    // occupiedBuffers maintains count of occupied buffers
11    private int occupiedBuffers = 0;
12
13    // variables that maintain read and write buffer locations
14    private int readLocation = 0;
15    private int writeLocation = 0;
16
17    // place value into buffer
18    public synchronized void set( int value )
19    {
20        // get name of thread that called this method
21        String name = Thread.currentThread().getName();
22    }
}
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 SynchronizedBuffer controls access to a shared array of integers. (Part 2 of 7.)

```
23      // while buffer full, place thread in waiting state
24      while ( occupiedBuffers == buffers.length )
25      {
26          // output thread and buffer information, then wait
27          try
28          {
29              System.err.println( "\nAll buffers full. " +
30                  name + " waits." );
31              wait(); // wait until space is available
32          } // end try
33
34          // if waiting thread interrupted, print stack trace
35          catch ( InterruptedException exception )
36          {
37              exception.printStackTrace();
38          } // end catch
39
40      } // end while
41
42      // place value in writeLocation of buffers
43      buffers[ writeLocation ] = value;
44
45      // output produced value
46      System.err.println( "\n" + name + " writes " +
47          buffers[ writeLocation ] + " " );
48
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 SynchronizedBuffer controls access to a shared array of integers. (Part 3 of 7.)

```
49         // indicate that one more buffer is occupied
50         ++occupiedBuffers;
51
52         // update writeLocation for future write operation
53         writeLocation = ( writeLocation + 1 ) % buffers.length;
54
55         // display contents of shared buffers
56         System.err.println( createStateOutput() );
57
58         notify(); // return a waiting thread to ready state
59     } // end method set
60
61     // return value from buffer
62     public synchronized int get()
63     {
64         // get name of thread that called this method
65         String name = Thread.currentThread().getName();
66
67         // while buffer is empty, place thread in waiting state
68         while ( occupiedBuffers == 0 )
69         {
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 SynchronizedBuffer controls access to a shared array of integers. (Part 4 of 7.)

```
70      // output thread and buffer information, then wait
71      try
72      {
73          System.err.println( "\nAll buffers empty. " +
74              name + " waits." );
75          wait(); // wait until buffer contains new data
76      } // end try
77
78      // if waiting thread interrupted, print stack trace
79      catch ( InterruptedException exception )
80      {
81          exception.printStackTrace();
82      } // end catch
83
84 } // end while
85
86 // obtain value at current readLocation
87 int readValue = buffers[ readLocation ];
88
89 // output consumed value
90 System.err.println( "\n" + name + " reads " +
91     readValue + " " );
92
93 // decrement occupied buffers value
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 synchronizedBuffer controls access to a shared array of integers. (Part 5 of 7.)

```
94         --occupiedBuffers;
95
96         // update readLocation for future read operation
97         readLocation = ( readLocation + 1 ) % buffers.length;
98
99         // display contents of shared buffers
100        System.err.println( createStateOutput() );
101
102        notify(); // return a waiting thread to ready state
103
104        return readValue;
105    } // end method get
106
107    // create state output
108    public String createStateOutput()
109    {
110        // first line of state information
111        String output = "(buffers occupied: " +
112            occupiedBuffers + ")\\nbuffers: ";
113
114        for ( int i = 0; i < buffers.length; ++i )
115        {
116            output += " " + buffers[ i ] + " ";
117        } // end for
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 `SynchronizedBuffer` controls access to a shared array of integers. (Part 6 of 7.)

```
118
119      // second line of state information
120      output += "\n      ";
121
122      for ( int i = 0; i < buffers.length; ++i )
123      {
124          output += "---- ";
125      } // end for
126
127      // third line of state information
128      output += "\n      ";
129
130      // append readLocation (R) and writeLocation (W)
131      // indicators below appropriate buffer locations
132      for ( int i = 0; i < buffers.length; ++i )
133      {
134          if ( i == writeLocation &&
135              writeLocation == readLocation )
136          {
137              output += " WR ";
138          } // end if
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.6 SynchronizedBuffer controls access to a shared array of integers. (Part 7 of 7.)

```
139         else if ( i == writeLocation )
140         {
141             output += " W ";
142         } // end if
143         else if ( i == readLocation )
144         {
145             output += " R ";
146         } // end if
147         else
148         {
149             output += "     ";
150         } // end else
151
152     } // end for
153
154     output += "\n";
155
156     return output;
157 } // end method createStateOutput
158
159 } // end class CircularBuffer
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `circularBufferTest` instantiates producer and consumer threads. (Part 1 of 9.)

```
1 // Fig. 6.7: CircularBufferTest.java
2 // CircularBufferTest shows two threads manipulating a
3 // circular buffer.
4
5 // set up the producer and consumer threads and start them
6 public class CircularBufferTest
7 {
8     public static void main ( String args[] )
9     {
10         // create shared object for threads; use a reference
11         // to a CircularBuffer rather than a Buffer reference
12         // to invoke CircularBuffer method createStateOutput
13         CircularBuffer sharedLocation = new CircularBuffer();
14
15         // display initial state of buffers in CircularBuffer
16         System.err.println( sharedLocation.createStateOutput() );
17
18         // set up threads
19         Producer producer = new Producer( sharedLocation );
20         Consumer consumer = new Consumer( sharedLocation );
21 }
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `circularBufferTest` instantiates producer and consumer threads. (Part 2 of 9.)

```
22     producer.start(); // start producer thread
23     consumer.start(); // start consumer thread
24 } // end main
25
26 } // end class CircularBufferTest
```

Sample Output:

```
(buffers occupied: 0)
buffers: -1 -1 -1
-----
WR
```

All buffers empty. Consumer waits.

```
Producer writes 11
(buffers occupied: 1)
buffers: 11 -1 -1
-----
R W
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `circularBufferTest` instantiates producer and consumer threads. (Part 3 of 9.)

```
Consumer reads 11
(buffers occupied: 0)
buffers: 11 -1 -1
-----
WR

Producer writes 12
(buffers occupied: 1)
buffers: 11 12 -1
-----
R W

Producer writes 13
(buffers occupied: 2)
buffers: 11 12 13
-----
W R
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `circularBufferTest` instantiates producer and consumer threads. (Part 4 of 9.)

Sample Output (Cont.).

```
Consumer reads 12
(buffers occupied: 1)
buffers: 11 12 13
-----  
W R
```

```
Producer writes 14
(buffers occupied: 2)
buffers: 14 12 13
-----  
W R
```

```
Producer writes 15
(buffers occupied: 3)
buffers: 14 15 13
-----  
WR
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `circularBufferTest` instantiates producer and consumer threads. (Part 5 of 9.)

All buffers full. Producer waits.

Consumer reads 13
(buffers occupied: 2)
buffers: 14 15 13

R W

Producer writes 16
(buffers occupied: 3)
buffers: 14 15 16

WR

All buffers full. Producer waits.

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `circularBufferTest` instantiates producer and consumer threads. (Part 6 of 9.)

Sample Output (Cont.).

```
Consumer reads 14
(buffers occupied: 2)
buffers: 14 15 16
-----
W   R
```

```
Producer writes 17
(buffers occupied: 3)
buffers: 17 15 16
-----
WR
```

```
Consumer reads 15
(buffers occupied: 2)
buffers: 17 15 16
-----
W   R
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `circularBufferTest` instantiates producer and consumer threads. (Part 7 of 9.)

```
Consumer reads 16
(buffers occupied: 1)
buffers: 17 15 16
```

R W

```
Consumer reads 17
(buffers occupied: 0)
buffers: 17 15 16
```

WR

```
Producer writes 18
(buffers occupied: 1)
buffers: 17 18 16
```

R W

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `circularBufferTest` instantiates producer and consumer threads. (Part 8 of 9.)

Sample Output (Cont.):

```
Consumer reads 18
(buffers occupied: 0)
buffers: 17 18 16
-----  
          WR
```

All buffers empty. Consumer waits.

```
Producer writes 19
(buffers occupied: 1)
buffers: 17 18 19
-----  
          W           R
```

```
Consumer reads 19
(buffers occupied: 0)
buffers: 17 18 19
-----  
          WR
```

6.5 Java Multithreading Case Study, Part IV: Circular Buffer in Java

Figure 6.7 `circularBufferTest` instantiates producer and consumer threads. (Part 9 of 9.)

```
Producer writes 20  
(buffers occupied: 1)  
buffers: 20 18 19  
-----  
R W
```

```
Producer done producing.  
Terminating Producer.
```

```
Consumer reads 20  
(buffers occupied: 0)  
buffers: 20 18 19  
-----  
WR
```

```
Consumer read values totaling: 155.  
Terminating Consumer.
```

Chapter 7 – Deadlock and Indefinite Postponement

Outline

- 7.1 **Introduction**
- 7.2 **Examples of Deadlock**
 - 7.2.1 **Traffic Deadlock**
 - 7.2.2 **Simple Resource Deadlock**
 - 7.2.3 **Deadlock in Spooling Systems**
 - 7.2.4 **Example: Dining Philosophers**
- 7.3 **Related Problem: Indefinite Postponement**
- 7.4 **Resource Concepts**
- 7.5 **Four Necessary Conditions for Deadlock**
- 7.6 **Deadlock Solutions**
- 7.7 **Deadlock Prevention**
 - 7.7.1 **Denying the “Wait-For” Condition**
 - 7.7.2 **Denying the “No-Preemption Condition**
 - 7.7.3 **Denying the “Circular-Wait” Condition**

Chapter 7 – Deadlock and Indefinite Postponement

Outline (continued)

- 7.8 **Deadlock Avoidance with Dijkstra's Banker's Algorithm**
 - 7.8.1 **Example of a Safe State**
 - 7.8.2 **Example of an Unsafe State**
 - 7.8.3 **Example of Safe-State-to-Unsafe-State Transition**
 - 7.8.4 **Banker's Algorithm Resource Allocation**
 - 7.8.5 **Weaknesses in the Banker's Algorithm**
- 7.9 **Deadlock Detection**
 - 7.9.1 **Resource-Allocation Graphs**
 - 7.9.2 **Reduction of Resource-Allocation Graphs**
- 7.10 **Deadlock Recovery**
- 7.11 **Deadlock Strategies in Current and Future Systems**

Objectives

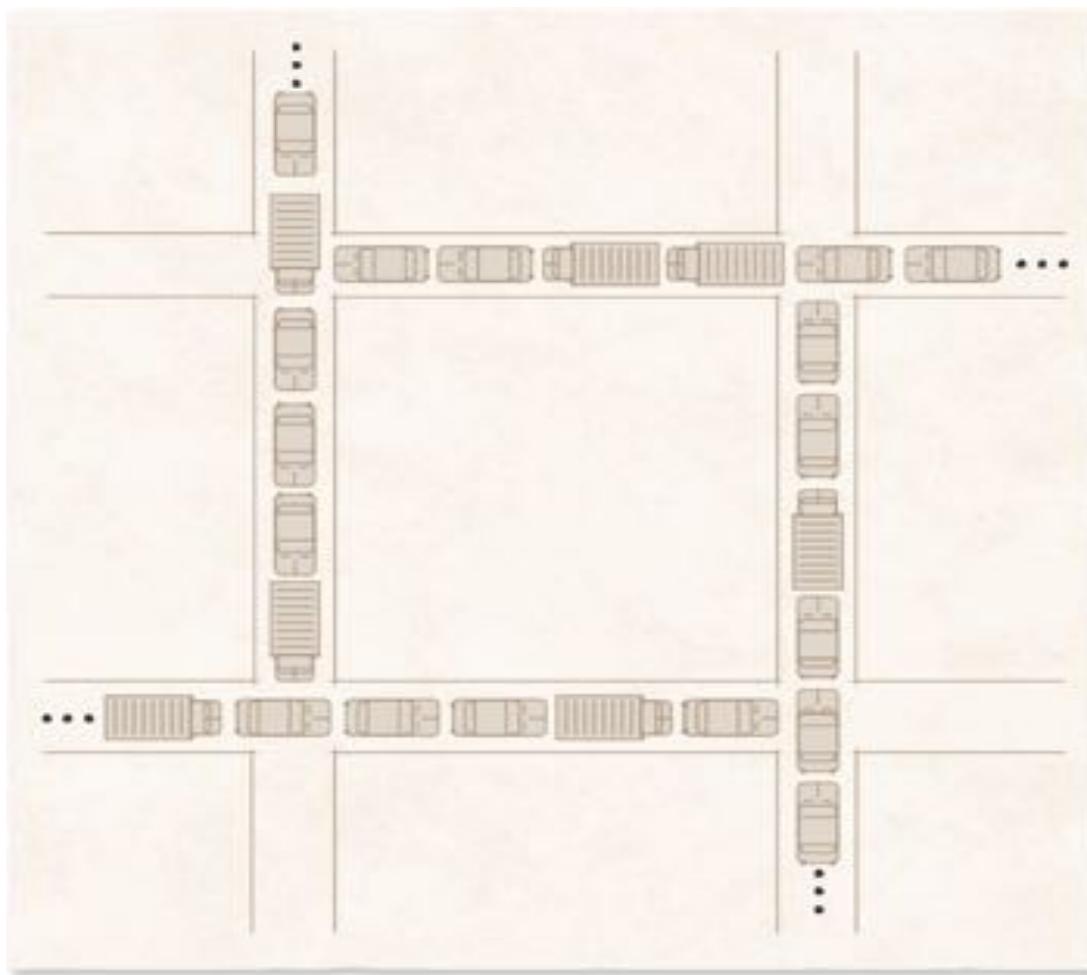
- After reading this chapter, you should understand:
 - the problem of deadlock.
 - the four necessary conditions for deadlock to exist.
 - the problem of indefinite postponement.
 - the notions of deadlock prevention, avoidance, detection and recovery.
 - algorithms for deadlock avoidance and detection.
 - how systems can recover from deadlocks.

7.1 Introduction

- **Deadlock**
 - A process or thread is waiting for a particular event that will not occur
- **System deadlock**
 - One or more processes are deadlocked

7.2.1 Traffic Deadlock

Figure 7.1 Traffic deadlock example.

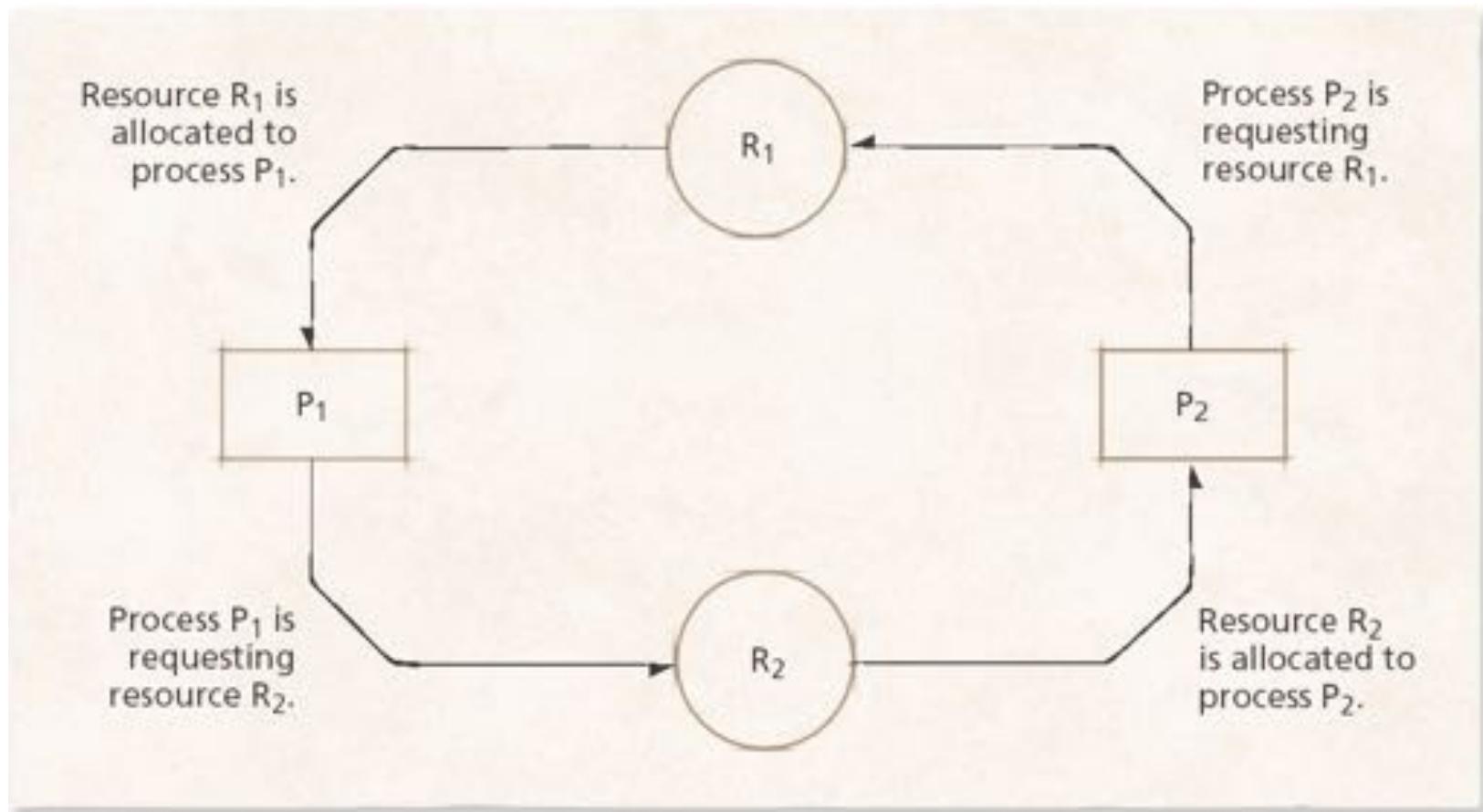


7.2.2 Simple Resource Deadlock

- Most deadlocks develop because of the normal contention for dedicated resources
- Circular wait is characteristic of deadlocked systems

7.2.2 Simple Resource Deadlock

Figure 7.2 Resource deadlock example. This system is deadlocked because each process holds a resource being requested by the other process and neither process is willing to release the resource it holds.



7.2.3 Deadlock in Spooling Systems

- Spooling systems are prone to deadlock
- Common solution
 - Restrain input spoolers so that when the spooling file begins to reach some saturation threshold, the spoolers do not read in more print jobs
- Today's systems
 - Printing begins before the job is completed so that a full spooling file can be emptied even while a job is still executing
 - Same concept has been applied to streaming audio and video

7.2.4 Example: Dining Philosophers

- Problem statement:

Five philosophers sit around a circular table. Each leads a simple life alternating between thinking and eating spaghetti. In front of each philosopher is a dish of spaghetti that is constantly replenished by a dedicated wait staff. There are exactly five forks on the table, one between each adjacent pair of philosophers. Eating spaghetti (in the most proper manner) requires that a philosopher use both adjacent forks (simultaneously). Develop a concurrent program free of deadlock and indefinite postponement that models the activities of the philosophers.

7.2.4 Example: Dining Philosophers

Figure 7.3 Dining philosopher behavior.

```
1 void typicalPhilosopher()
2 {
3     while ( true )
4     {
5         think();
6         eat();
7     } // end while
8
9 } // end typicalPhilosopher
```

7.2.4 Example: Dining Philosophers

- Constraints:
 - To prevent philosophers from starving:
 - Free of deadlock
 - Free of indefinite postponement
 - Enforce mutual exclusion
 - Two philosophers cannot use the same fork at once
- The problems of mutual exclusion, deadlock and indefinite postponement lie in the implementation of method eat.

7.2.4 Example: Dining Philosophers

Figure 7.4 Implementation of method eat.

```
1 void eat()
2 {
3     pickUpLeftFork();
4     pickUpRightFork();
5     eatForSomeTime();
6     putDownRightFork();
7     putDownLeftFork();
8 } // eat
```

7.3 Related Problem: Indefinite Postponement

- **Indefinite postponement**
 - Also called indefinite blocking or starvation
 - Occurs due to biases in a system's resource scheduling policies
- **Aging**
 - Technique that prevents indefinite postponement by increasing process's priority as it waits for resource

7.4 Resource Concepts

- Preemptible resources (e.g. processors and main memory)
 - Can be removed from a process without loss of work
- Nonpreemptible resources (e.g. tape drives and optical scanners)
 - Cannot be removed from the processes to which they are assigned without loss of work
- Reentrant code
 - Cannot be changed while in use
 - May be shared by several processes simultaneously
- Serially reusable code
 - May be changed but is reinitialized each time it is used
 - May be used by only one process at a time

7.5 Four Necessary Conditions for Deadlock

- Mutual exclusion condition
 - Resource may be acquired exclusively by only one process at a time
- Wait-for condition (hold-and-wait condition)
 - Process that has acquired an exclusive resource may hold that resource while the process waits to obtain other resources
- No-preemption condition
 - Once a process has obtained a resource, the system cannot remove it from the process's control until the process has finished using the resource
- Circular-wait condition
 - Two or more processes are locked in a “circular chain” in which each process is waiting for one or more resources that the next process in the chain is holding

7.6 Deadlock Solutions

- Four major areas of interest in deadlock research
 - Deadlock prevention
 - Deadlock avoidance
 - Deadlock detection
 - Deadlock recovery

7.7 Deadlock Prevention

- Deadlock prevention
 - Condition a system to remove any possibility of deadlocks occurring
 - Deadlock cannot occur if any one of the four necessary conditions is denied
 - First condition (mutual exclusion) cannot be broken

7.7.1 Denying the “Wait-For” Condition

- When denying the “wait-for condition”
 - All of the resources a process needs to complete its task must be requested at once
 - This leads to inefficient resource allocation

7.7.2 Denying the “No-Preemption” Condition

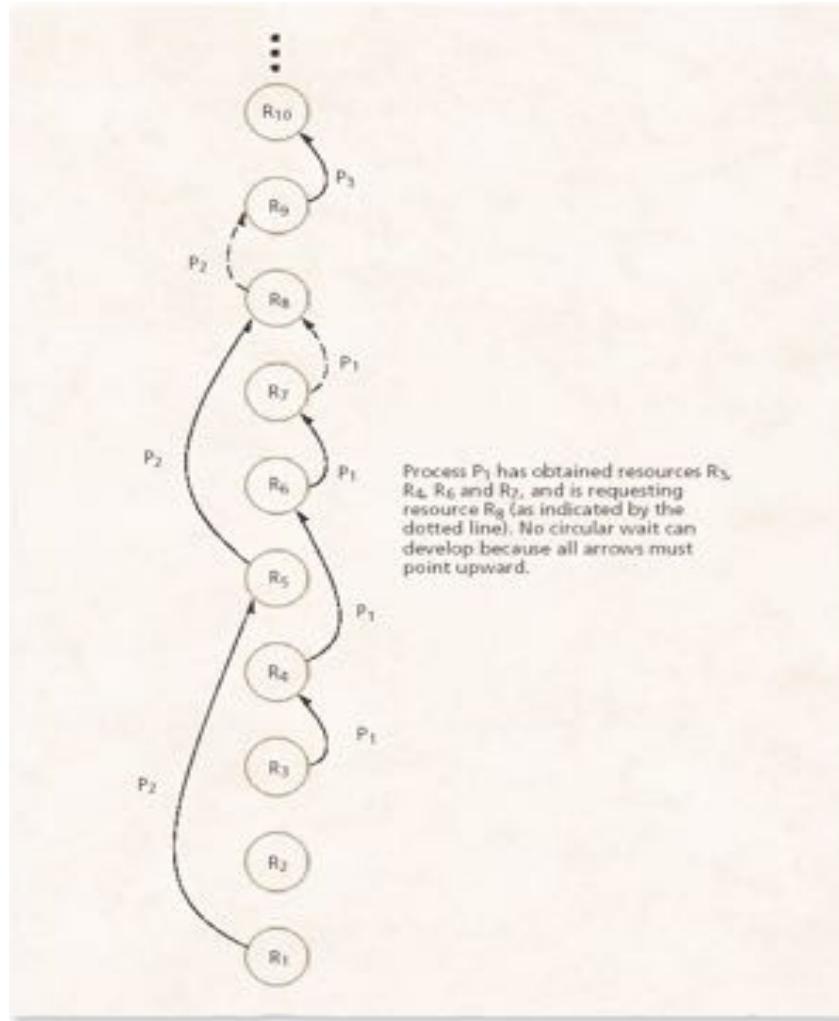
- When denying the “no-preemption” condition
 - Processes may lose work when resources are preempted
 - This can lead to substantial overhead as processes must be restarted

7.7.3 Denying the “Circular-Wait” Condition

- Denying the “circular-wait” condition:
 - Uses a linear ordering of resources to prevent deadlock
 - More efficient resource utilization than the other strategies
- Drawbacks
 - Not as flexible or dynamic as desired
 - Requires the programmer to determine the ordering of resources for each system

7.7.3 Denying the “Circular-Wait” Condition

Figure 7.5 Havender's linear ordering of resources for preventing deadlock.



7.8 Deadlock Avoidance with Dijkstra's Banker's Algorithm

- **Banker's Algorithm**
 - Impose less stringent conditions than in deadlock prevention in an attempt to get better resource utilization
 - Safe state
 - Operating system can guarantee that all current processes can complete their work within a finite time
 - Unsafe state
 - Does not imply that the system is deadlocked, but that the OS cannot guarantee that all current processes can complete their work within a finite time

7.8 Deadlock Avoidance with Dijkstra's Banker's Algorithm

- **Banker's Algorithm (cont.)**
 - Requires that resources be allocated to processes only when the allocations result in safe states.
 - It has a number of weaknesses (such as requiring a fixed number of processes and resources) that prevent it from being implemented in real systems

7.8.2 Example of an Unsafe State

Figure 7.6 Safe state.

Process	$\max(P_i)$ (maximum need)	$\text{loan}(P_i)$ (current loan)	$\text{claim}(P_i)$ (current claim)
P ₁	4	1	3
P ₂	6	4	2
P ₃	8	5	3
Total resources, t, = 12		Available resources, a, = 2	

7.8.2 Example of an Unsafe State

Figure 7.7 Unsafe state.

Process	$\max(P_i)$ (maximum need)	$\text{loan}(P_i)$ (current loan)	$\text{claim}(P_i)$ (current claim)
P ₁	10	8	2
P ₂	5	2	3
P ₃	3	1	2
Total resources, t _r = 12		Available resources, a = 1	

7.8.3 Example of Safe-State-to-Unsafe-State Transition

- **Safe-state-to-unsafe-state transition:**
 - Suppose the current state of a system is safe, as shown in Fig. 7.6.
 - The current value of a is 2.
 - Now suppose that process P_3 requests an additional resource

7.8.3 Example of Safe-State-to-Unsafe-State Transition

Figure 7.8 Safe-state-to-unsafe-state transition.

Process	$\max(P_i)$ (maximum need)	$\text{loan}(P_i)$ (current loan)	$\text{claim}(P_i)$ (current claim)
P ₁	4	1	3
P ₂	6	4	2
P ₃	8	6	2
Total resources, t, = 12		Available resources, a, = 1	

7.8.4 Banker's Algorithm Resource Allocation

- Is the state in the next slide safe?

7.8.4 Banker's Algorithm Resource Allocation

Figure 7.9 State description of three processes.

Process	$\text{max}(P_i)$	$\text{loan}(P_i)$	$\text{claim}(P_i)$
P ₁	5	1	4
P ₂	3	1	2
P ₃	10	5	5
$a = 2$			

7.8.4 Banker's Algorithm Resource Allocation

- Answer:
 - There is no guarantee that all of these processes will finish
 - P_2 will be able to finish by using up the two remaining resources
 - Once P_2 is done, there are only three available resources left
 - This is not enough to satisfy either P_1 's claim of 4 or P_3 's claim of five

7.8.5 Weaknesses in the Banker's Algorithm

- **Weaknesses**
 - Requires there be a fixed number of resource to allocate
 - Requires the population of processes to be fixed
 - Requires the banker to grant all requests within “finite time”
 - Requires that clients repay all loans within “finite time”
 - Requires processes to state maximum needs in advance

7.9 Deadlock Detection

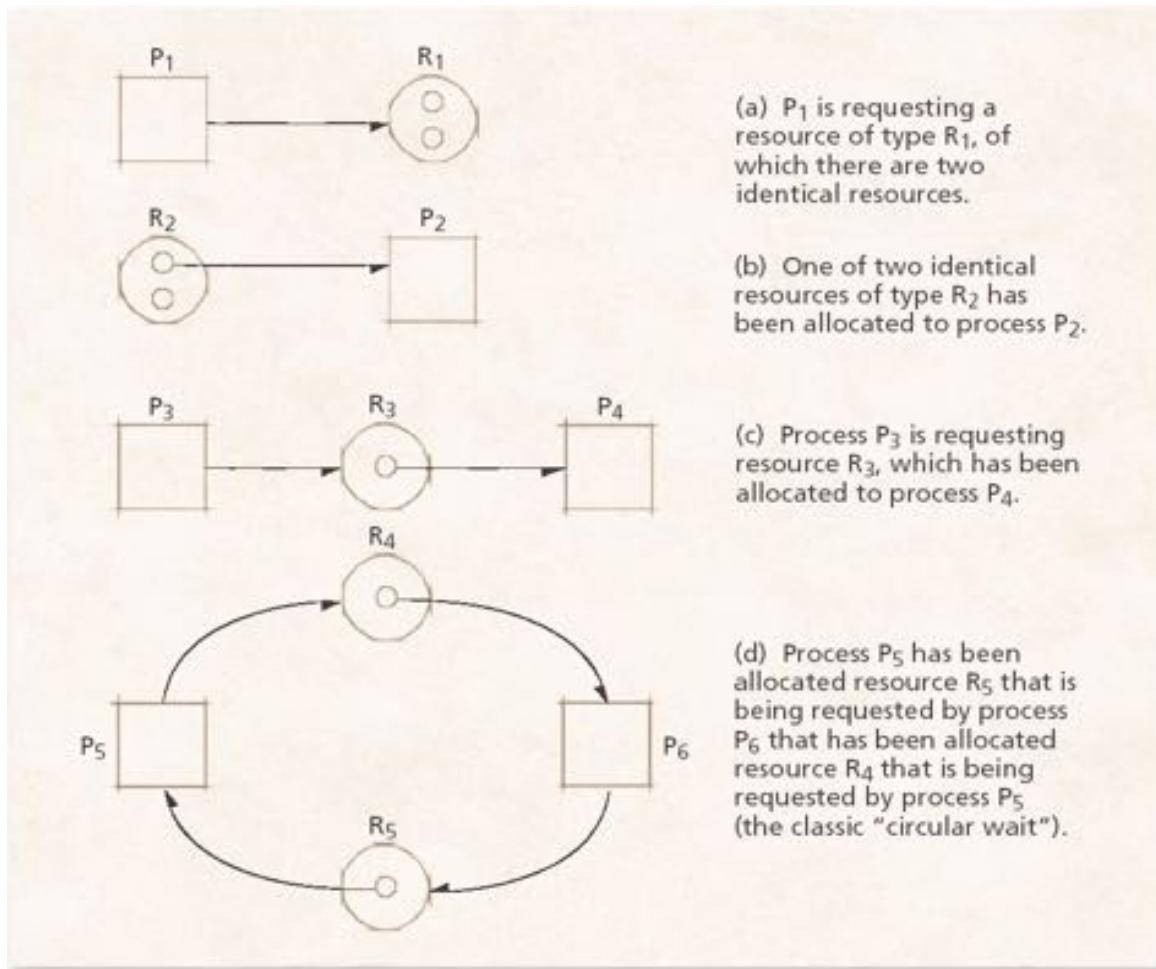
- **Deadlock detection**
 - Used in systems in which deadlocks can occur
 - Determines if deadlock has occurred
 - Identifies those processes and resources involved in the deadlock
 - Deadlock detection algorithms can incur significant runtime overhead

7.9.1 Resource-Allocation Graphs

- Resource-allocation graphs
 - Squares
 - Represent processes
 - Large circles
 - Represent classes of identical resources
 - Small circles drawn inside large circles
 - Indicate separate identical resources of each class

7.9.1 Resource-Allocation Graphs

Figure 7.10 Resource-allocation and request graphs.

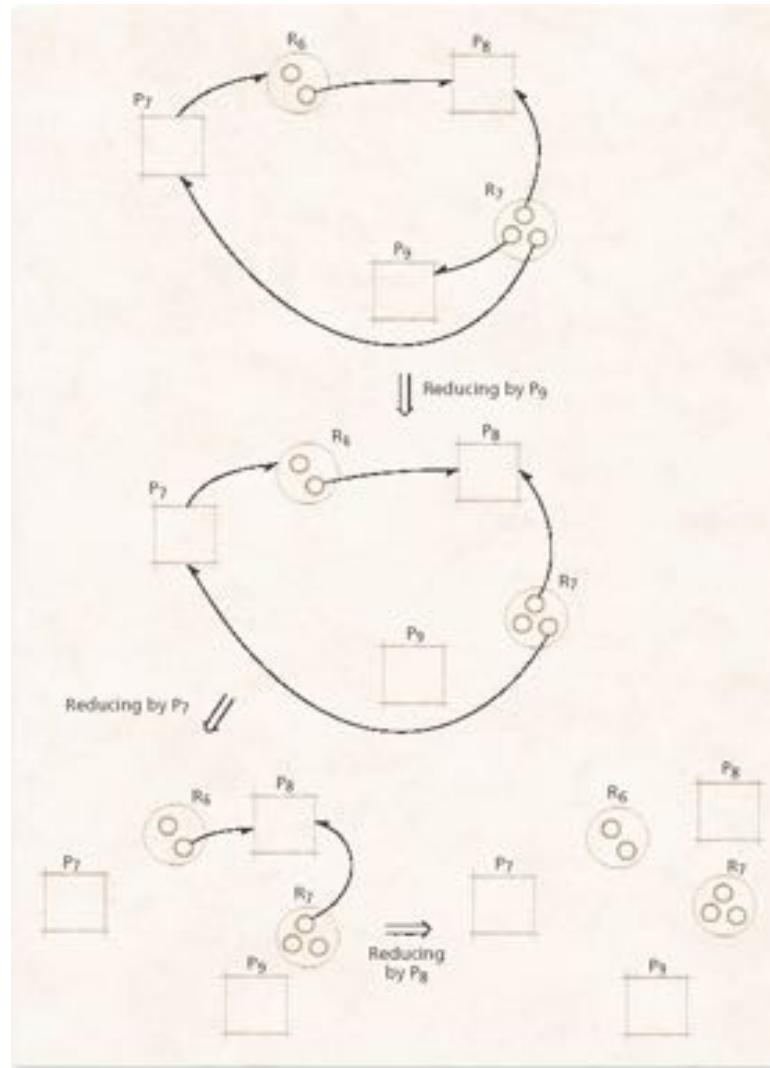


7.9.2 Reduction of Resource-Allocation Graphs

- Graph reductions
 - If a process's resource requests may be granted, the graph may be reduced by that process
 - If a graph can be reduced by all its processes, there is no deadlock
 - If a graph cannot be reduced by all its processes, the irreducible processes constitute the set of deadlocked processes in the graph

7.9.2 Reduction of Resource-Allocation Graphs

Figure 7.11 Graph reductions determining that no deadlock exists.



7.10 Deadlock Recovery

- **Deadlock recovery**
 - Clears deadlocks from system so that deadlocked processes may complete their execution and free their resources
- **Suspend/resume mechanism**
 - Allows system to put a temporary hold on a process
 - Suspended processes can be resumed without loss of work
- **Checkpoint/rollback**
 - Facilitates suspend/resume capabilities
 - Limits the loss of work to the time the last checkpoint was made

7.11 Deadlock Strategies in Current and Future Systems

- Deadlock is viewed as limited annoyance in personal computer systems
 - Some systems implement basic prevention methods suggested by Havender
 - Some others ignore the problem, because checking deadlocks would reduce systems' performance
- Deadlock continues to be an important research area

Chapter 8 – Processor Scheduling

Outline

- 8.1 **Introduction**
- 8.2 **Scheduling Levels**
- 8.3 **Preemptive vs. Nonpreemptive Scheduling**
- 8.4 **Priorities**
- 8.5 **Scheduling Objectives**
- 8.6 **Scheduling Criteria**
- 8.7 **Scheduling Algorithms**
 - 8.7.1 **First-In-First-Out (FIFO) Scheduling**
 - 8.7.2 **Round-Robin (RR) Scheduling**
 - 8.7.3 **Shortest-Process-First (SPF) Scheduling**
 - 8.7.4 **Highest-Response-Ratio-Next (HRRN) Scheduling**
 - 8.7.5 **Shortest-Remaining-Time (SRT) Scheduling**
 - 8.7.6 **Multilevel Feedback Queues**
 - 8.7.7 **Fair Share Scheduling**
- 8.8 **Deadline Scheduling**
- 8.9 **Real-Time Scheduling**
- 8.10 **Java Thread Scheduling**



Objectives

- After reading this chapter, you should understand:
 - the goals of processor scheduling.
 - preemptive vs. nonpreemptive scheduling.
 - the role of priorities in scheduling.
 - scheduling criteria.
 - common scheduling algorithms.
 - the notions of deadline scheduling and real-time scheduling.
 - Java thread scheduling.

8.1 Introduction

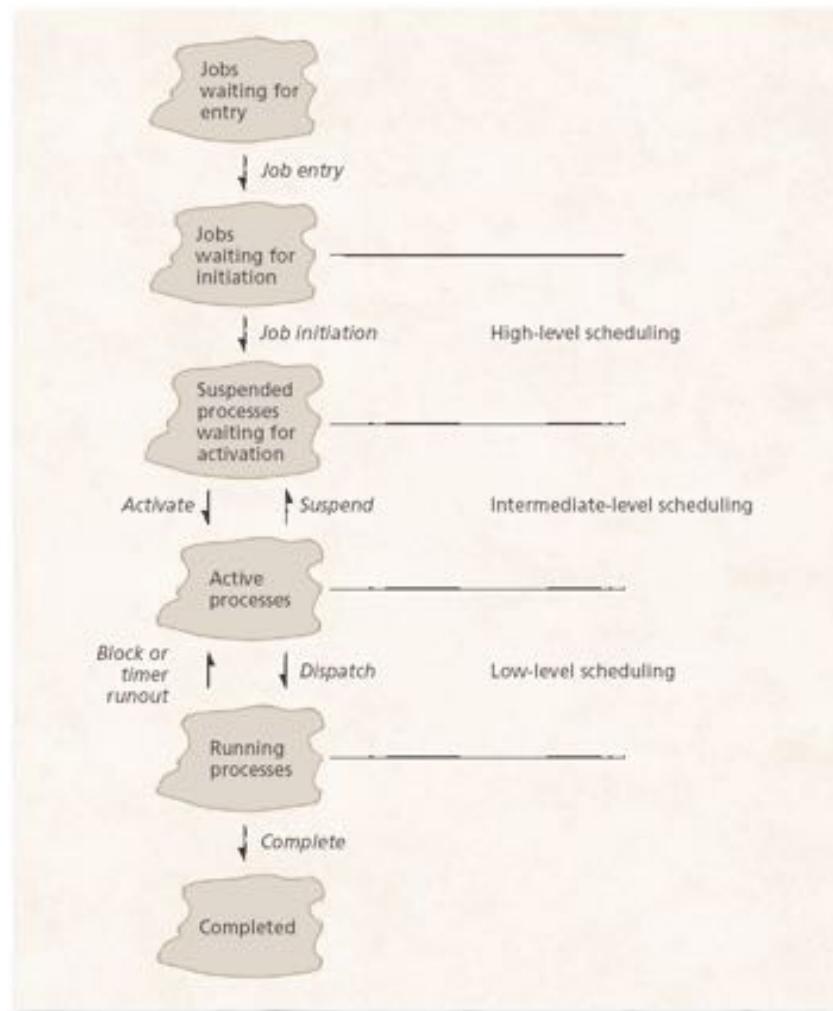
- Processor scheduling policy
 - Decides which process runs at given time
 - Different schedulers will have different goals
 - Maximize throughput
 - Minimize latency
 - Prevent indefinite postponement
 - Complete process by given deadline
 - Maximize processor utilization

8.2 Scheduling Levels

- High-level scheduling
 - Determines which jobs can compete for resources
 - Controls number of processes in system at one time
- Intermediate-level scheduling
 - Determines which processes can compete for processors
 - Responds to fluctuations in system load
- Low-level scheduling
 - Assigns priorities
 - Assigns processors to processes

8.2 Scheduling Levels

Figure 8.1 Scheduling levels.



8.3 Preemptive vs. Nonpreemptive Scheduling

- Preemptive processes
 - Can be removed from their current processor
 - Can lead to improved response times
 - Important for interactive environments
 - Preempted processes remain in memory
- Nonpreemptive processes
 - Run until completion or until they yield control of a processor
 - Unimportant processes can block important ones indefinitely

8.4 Priorities

- Static priorities
 - Priority assigned to a process does not change
 - Easy to implement
 - Low overhead
 - Not responsive to changes in environment
- Dynamic priorities
 - Responsive to change
 - Promote smooth interactivity
 - Incur more overhead than static priorities
 - Justified by increased responsiveness

8.5 Scheduling Objectives

- Different objectives depending on system
 - Maximize throughput
 - Maximize number of interactive processes receiving acceptable response times
 - Minimize resource utilization
 - Avoid indefinite postponement
 - Enforce priorities
 - Minimize overhead
 - Ensure predictability

8.5 Scheduling Objectives

- Several goals common to most schedulers
 - Fairness
 - Predictability
 - Scalability

8.6 Scheduling Criteria

- Processor-bound processes
 - Use all available processor time
- I/O-bound
 - Generates an I/O request quickly and relinquishes processor
- Batch processes
 - Contains work to be performed with no user interaction
- Interactive processes
 - Requires frequent user input

8.7 Scheduling Algorithms

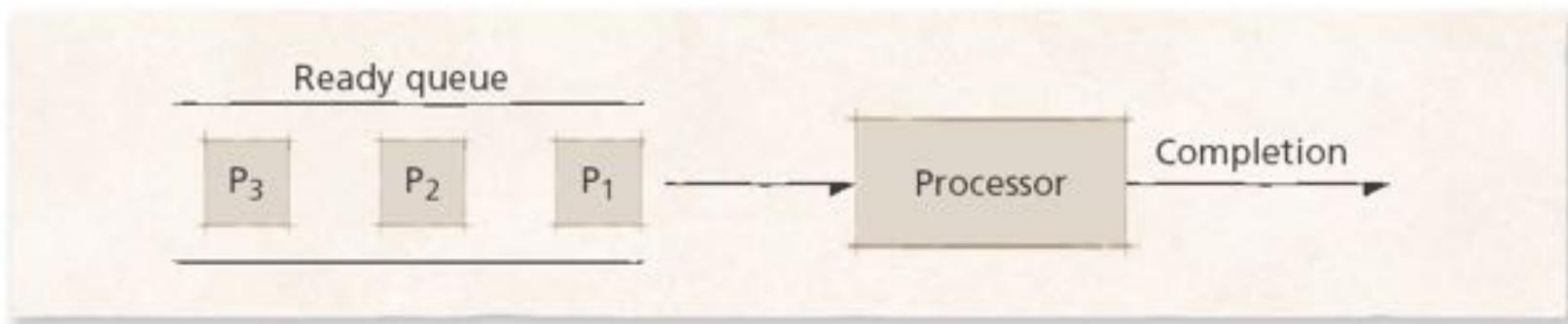
- Scheduling algorithms
 - Decide when and for how long each process runs
 - Make choices about
 - Preemptibility
 - Priority
 - Running time
 - Run-time-to-completion
 - fairness

8.7.1 First-In-First-Out (FIFO) Scheduling

- FIFO scheduling
 - Simplest scheme
 - Processes dispatched according to arrival time
 - Nonpreemptible
 - Rarely used as primary scheduling algorithm

8.7.1 First-In-First-Out (FIFO) Scheduling

Figure 8.2 First-in-first-out scheduling.

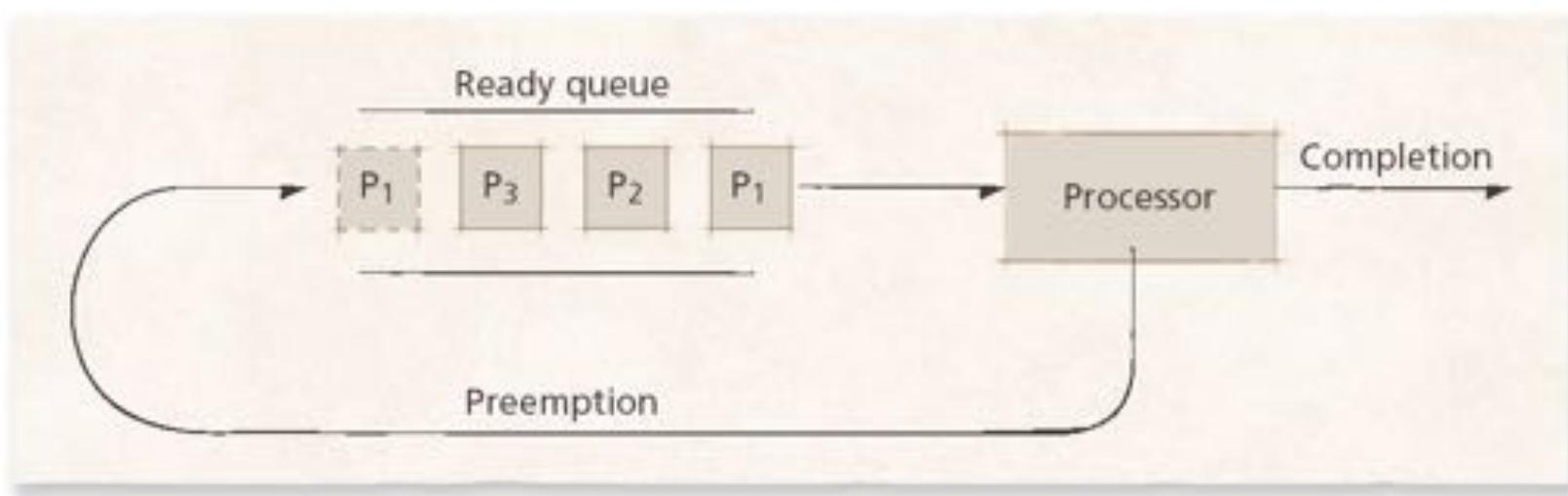


8.7.2 Round-Robin (RR) Scheduling

- Round-robin scheduling
 - Based on FIFO
 - Processes run only for a limited amount of time called a time slice or quantum
 - Preemptible
 - Requires the system to maintain several processes in memory to minimize overhead
 - Often used as part of more complex algorithms

8.7.2 Round-Robin (RR) Scheduling

Figure 8.3 Round-robin scheduling.



8.7.2 Round-Robin (RR) Scheduling

- Selfish round-robin scheduling
 - Increases priority as process ages
 - Two queues
 - Active
 - Holding
 - Favors older processes to avoids unreasonable delays

8.7.2 Round-Robin (RR) Scheduling

- Quantum size
 - Determines response time to interactive requests
 - Very large quantum size
 - Processes run for long periods
 - Degenerates to FIFO
 - Very small quantum size
 - System spends more time context switching than running processes
 - Middle-ground
 - Long enough for interactive processes to issue I/O request
 - Batch processes still get majority of processor time

8.7.3 Shortest-Process-First (SPF) Scheduling

- Scheduler selects process with smallest time to finish
 - Lower average wait time than FIFO
 - Reduces the number of waiting processes
 - Potentially large variance in wait times
 - Nonpreemptive
 - Results in slow response times to arriving interactive requests
 - Relies on estimates of time-to-completion
 - Can be inaccurate or falsified
 - Unsuitable for use in modern interactive systems

8.7.4 Highest-Response-Ratio-Next (HRRN) Scheduling

- HRRN scheduling
 - Improves upon SPF scheduling
 - Still nonpreemptive
 - Considers how long process has been waiting
 - Prevents indefinite postponement

8.7.5 Shortest-Remaining-Time (SRT) Scheduling

- SRT scheduling
 - Preemptive version of SPF
 - Shorter arriving processes preempt a running process
 - Very large variance of response times: long processes wait even longer than under SPF
 - Not always optimal
 - Short incoming process can preempt a running process that is near completion
 - Context-switching overhead can become significant

8.7.6 Multilevel Feedback Queues

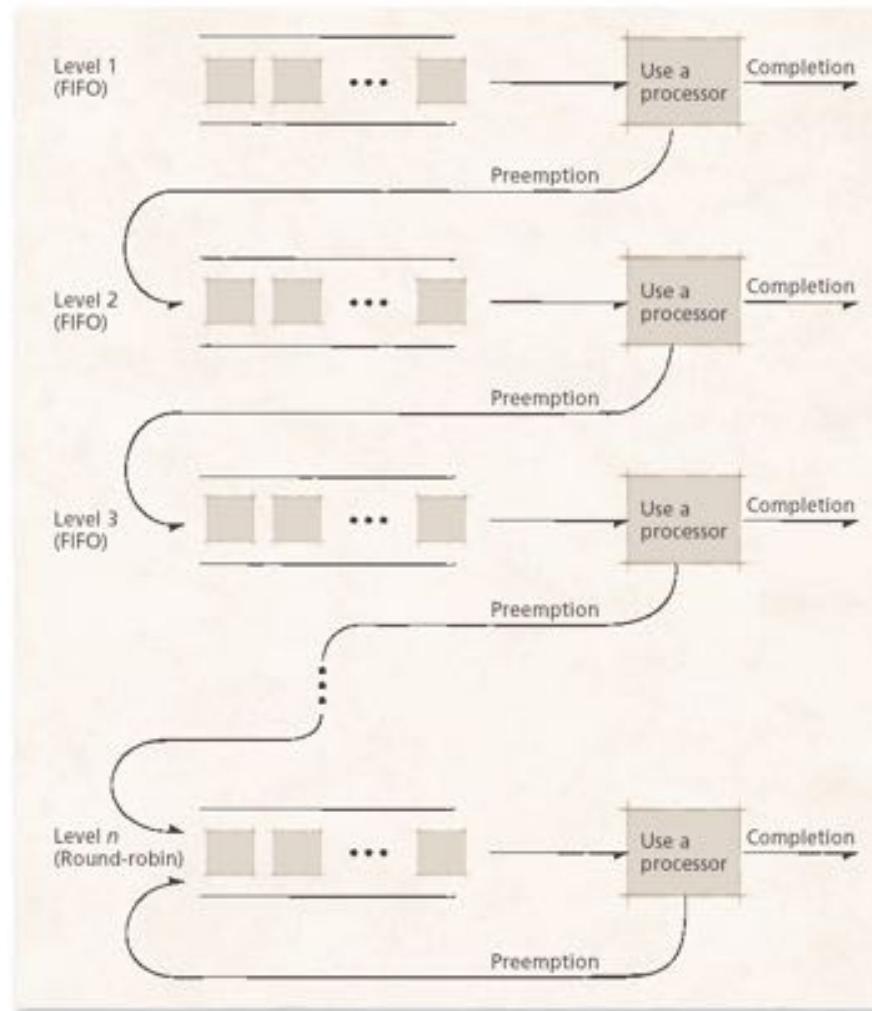
- Different processes have different needs
 - Short I/O-bound interactive processes should generally run before processor-bound batch processes
 - Behavior patterns not immediately obvious to the scheduler
- Multilevel feedback queues
 - Arriving processes enter the highest-level queue and execute with higher priority than processes in lower queues
 - Long processes repeatedly descend into lower levels
 - Gives short processes and I/O-bound processes higher priority
 - Long processes will run when short and I/O-bound processes terminate
 - Processes in each queue are serviced using round-robin
 - Process entering a higher-level queue preempt running processes

8.7.6 Multilevel Feedback Queues

- Algorithm must respond to changes in environment
 - Move processes to different queues as they alternate between interactive and batch behavior
- Example of an adaptive mechanism
 - Adaptive mechanisms incur overhead that often is offset by increased sensitivity to process behavior

8.7.6 Multilevel Feedback Queues

Figure 8.4 Multilevel feedback queues.

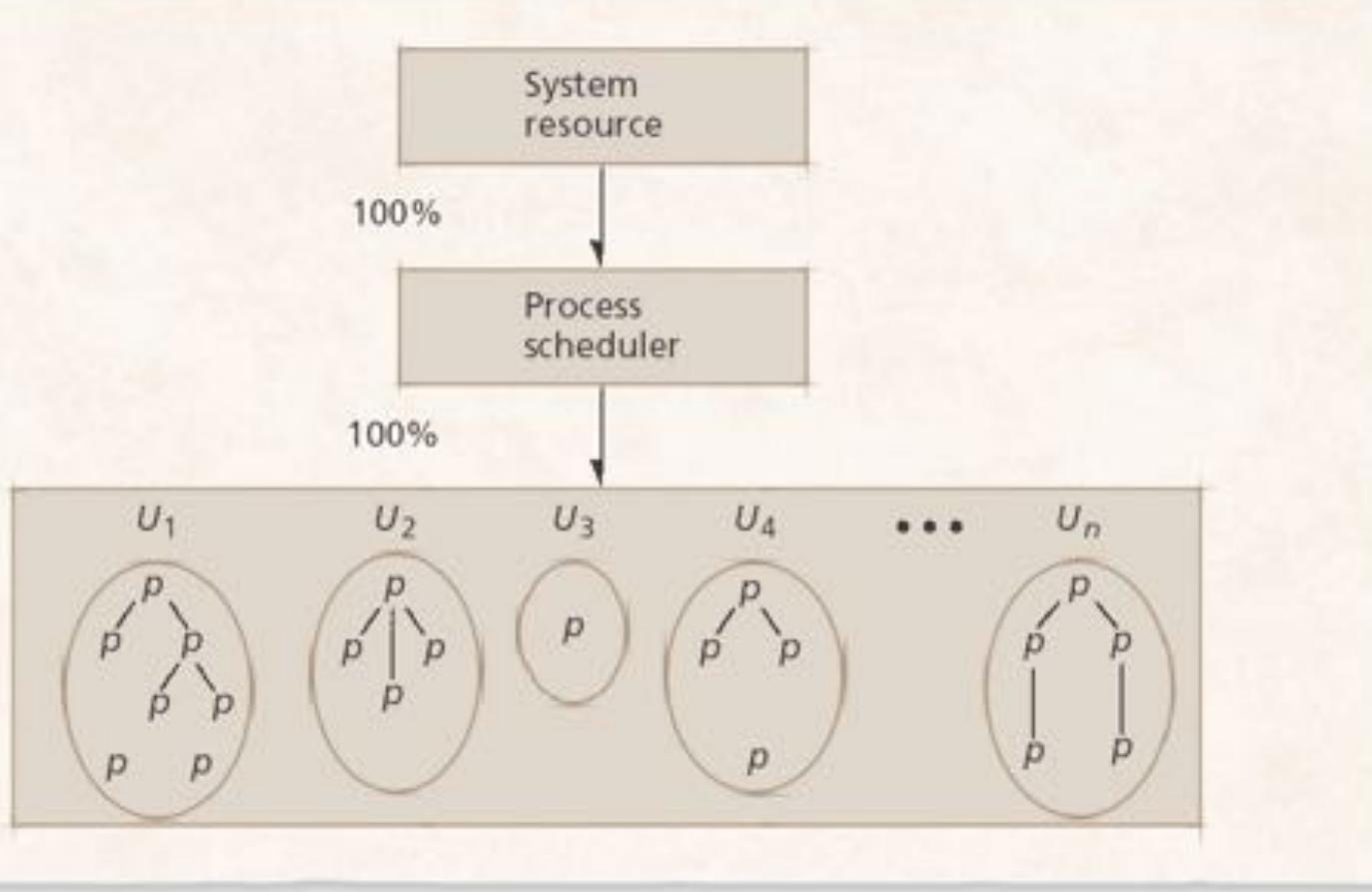


8.7.7 Fair Share Scheduling

- FSS controls users' access to system resources
 - Some user groups more important than others
 - Ensures that less important groups cannot monopolize resources
 - Unused resources distributed according to the proportion of resources each group has been allocated
 - Groups not meeting resource-utilization goals get higher priority

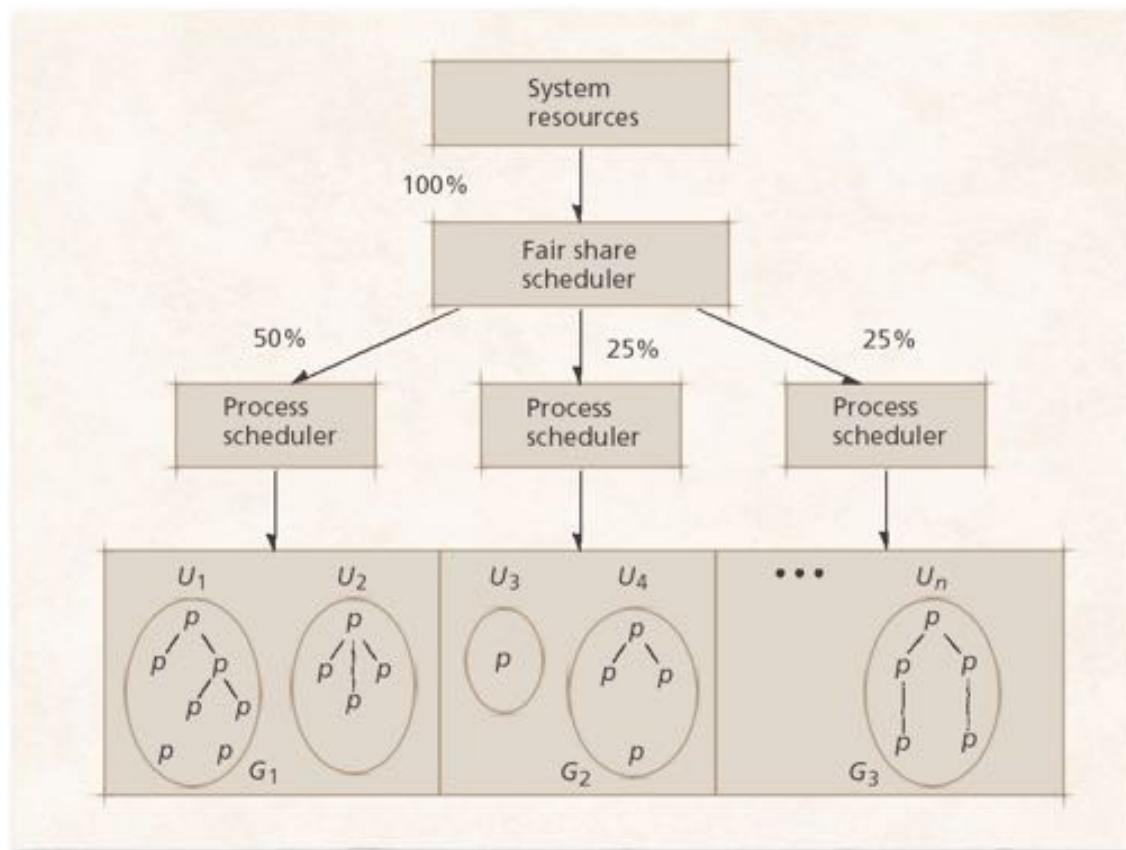
8.7.7 Fair Share Scheduling

Figure 8.5 Standard UNIX process scheduler. The scheduler grants the processor to users, each of whom may have many processes. (Property of AT&T Archives. Reprinted with permission of AT&T.)



8.7.7 Fair Share Scheduling

Figure 8.6 Fair share scheduler. The fair share scheduler divides system resource capacity into portions, which are then allocated by process schedulers assigned to various fair share groups. (Property of AT&T Archives. Reprinted with permission of AT&T.)



8.8 Deadline Scheduling

- **Deadline scheduling**
 - Process must complete by specific time
 - Used when results would be useless if not delivered on-time
 - Difficult to implement
 - Must plan resource requirements in advance
 - Incurs significant overhead
 - Service provided to other processes can degrade

8.9 Real-Time Scheduling

- Real-time scheduling
 - Related to deadline scheduling
 - Processes have timing constraints
 - Also encompasses tasks that execute periodically
- Two categories
 - Soft real-time scheduling
 - Does not guarantee that timing constraints will be met
 - For example, multimedia playback
 - Hard real-time scheduling
 - Timing constraints will always be met
 - Failure to meet deadline might have catastrophic results
 - For example, air traffic control

8.9 Real-Time Scheduling

- Static real-time scheduling
 - Does not adjust priorities over time
 - Low overhead
 - Suitable for systems where conditions rarely change
 - Hard real-time schedulers
 - Rate-monotonic (RM) scheduling
 - Process priority increases monotonically with the frequency with which it must execute
 - Deadline RM scheduling
 - Useful for a process that has a deadline that is not equal to its period

8.9 Real-Time Scheduling

- Dynamic real-time scheduling
 - Adjusts priorities in response to changing conditions
 - Can incur significant overhead, but must ensure that the overhead does not result in increased missed deadlines
 - Priorities are usually based on processes' deadlines
 - Earliest-deadline-first (EDF)
 - Preemptive, always dispatch the process with the earliest deadline
 - Minimum-laxity-first
 - Similar to EDF, but bases priority on laxity, which is based on the process's deadline and its remaining run-time-to-completion

8.10 Java Thread Scheduling

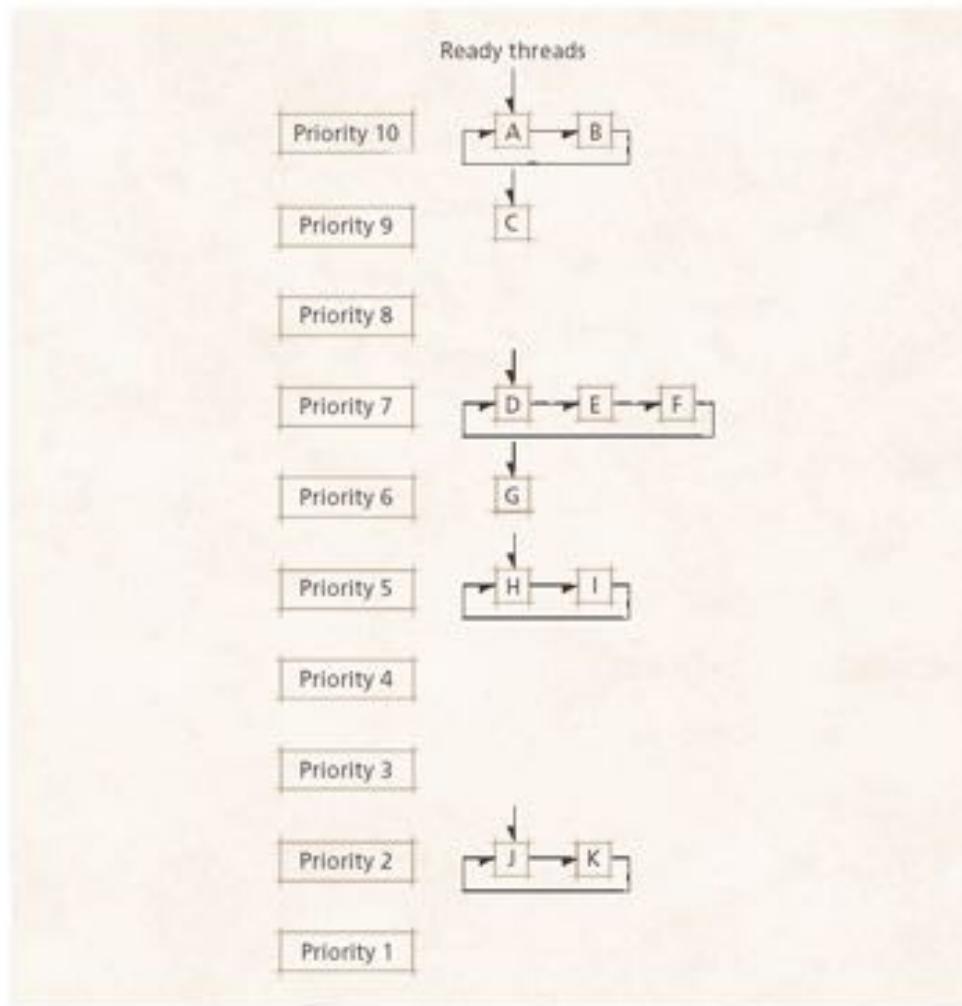
- Operating systems provide varying thread scheduling support
 - User-level threads
 - Implemented by each program independently
 - Operating system unaware of threads
 - Kernel-level threads
 - Implemented at kernel level
 - Scheduler must consider how to allocate processor time to a process's threads

8.10 Java Thread Scheduling

- Java threading scheduler
 - Uses kernel-level threads if available
 - User-mode threads implement timeslicing
 - Each thread is allowed to execute for at most one quantum before preemption
 - Threads can *yield* to others of equal priority
 - Only necessary on nontimesliced systems
 - Threads waiting to run are called waiting, sleeping or blocked

8.10 Java Thread Scheduling

Figure 8.7 Java thread priority scheduling.



Chapter 9 – Real Memory Organization and Management

Outline

- 9.1 **Introduction**
- 9.2 **Memory Organization**
- 9.3 **Memory Management**
- 9.4 **Memory Hierarchy**
- 9.5 **Memory Management Strategies**
- 9.6 **Contiguous vs. Noncontiguous Memory Allocation**
- 9.7 **Single-User Contiguous Memory Allocation**
- 9.7.1 **Overlays**
- 9.7.2 **Protection in a Single-User System**
- 9.7.3 **Single-Stream Batch Processing**
- 9.8 **Fixed-Partition Multiprogramming**
- 9.9 **Variable-Partition Multiprogramming**
- 9.9.1 **Variable-Partition Characteristics**
- 9.9.2 **Memory Placement Strategies**
- 9.10 **Multiprogramming with Memory Swapping**

Objectives

- After reading this chapter, you should understand:
 - the need for real (also called physical) memory management.
 - the memory hierarchy.
 - contiguous and noncontiguous memory allocation.
 - fixed- and variable-partition multiprogramming.
 - memory swapping.
 - memory placement strategies.

9.1 Introduction

- Memory divided into tiers
 - Main memory
 - Relatively expensive
 - Relatively small capacity
 - High-performance
 - Secondary storage
 - Cheap
 - Large capacity
 - Slow
 - Main memory requires careful management

9.2 Memory Organization

- Memory can be organized in different ways
 - One process uses entire memory space
 - Each process gets its own partition in memory
 - Dynamically or statically allocated
- Trend: Application memory requirements tend to increase over time to fill main memory capacities

9.2 Memory Organization

Figure 9.1 Microsoft Windows operating system memory requirements.

<i>Operating System</i>	<i>Release Date</i>	<i>Minimum Memory Requirement</i>	<i>Recommended Memory</i>
Windows 1.0	November 1985	256KB	
Windows 2.03	November 1987	320KB	
Windows 3.0	March 1990	896KB	1MB
Windows 3.1	April 1992	2.6MB	4MB
Windows 95	August 1995	8MB	16MB
Windows NT 4.0	August 1996	32MB	96MB
Windows 98	June 1998	24MB	64MB
Windows ME	September 2000	32MB	128MB
Windows 2000 Professional	February 2000	64MB	128MB
Windows XP Home	October 2001	64MB	128MB
Windows XP Professional	October 2001	128MB	256MB

9.3 Memory Management

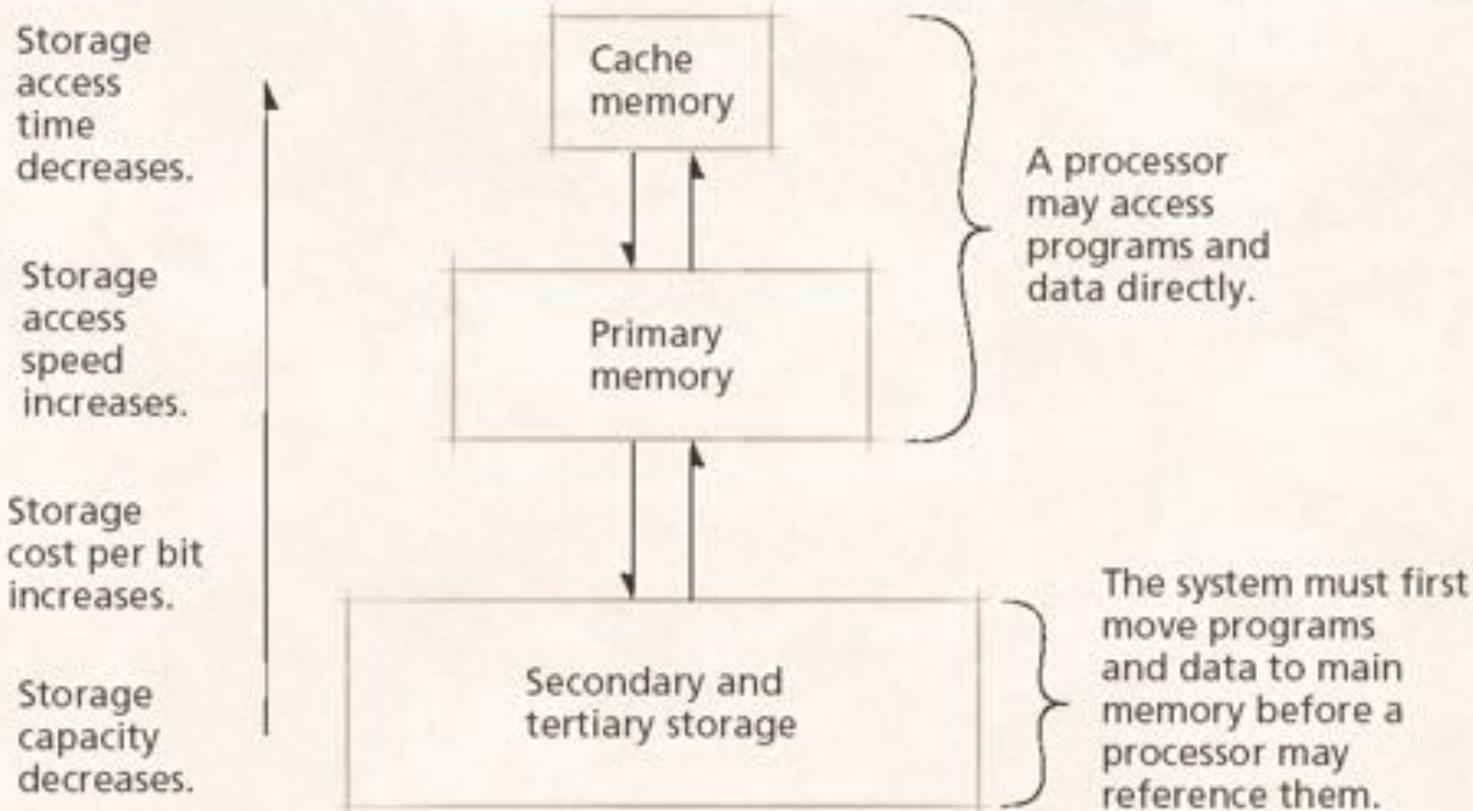
- Strategies for obtaining optimal memory performance
 - Performed by memory manager
 - Which process will stay in memory?
 - How much memory will each process have access to?
 - Where in memory will each process go?

9.4 Memory Hierarchy

- Main memory
 - Should store currently needed program instructions and data only
- Secondary storage
 - Stores data and programs that are not actively needed
- Cache memory
 - Extremely high speed
 - Usually located on processor itself
 - Most-commonly-used data copied to cache for faster access
 - Small amount of cache still effective for boosting performance
 - Due to temporal locality

9.4 Memory Hierarchy

Figure 9.2 Hierarchical memory organization.



9.5 Memory Management Strategies

- Strategies divided into several categories
 - Fetch strategies
 - Demand or anticipatory
 - Decides which piece of data to load next
 - Placement strategies
 - Decides where in main memory to place incoming data
 - Replacement strategies
 - Decides which data to remove from main memory to make more space

9.6 Contiguous vs. Noncontiguous Memory Allocation

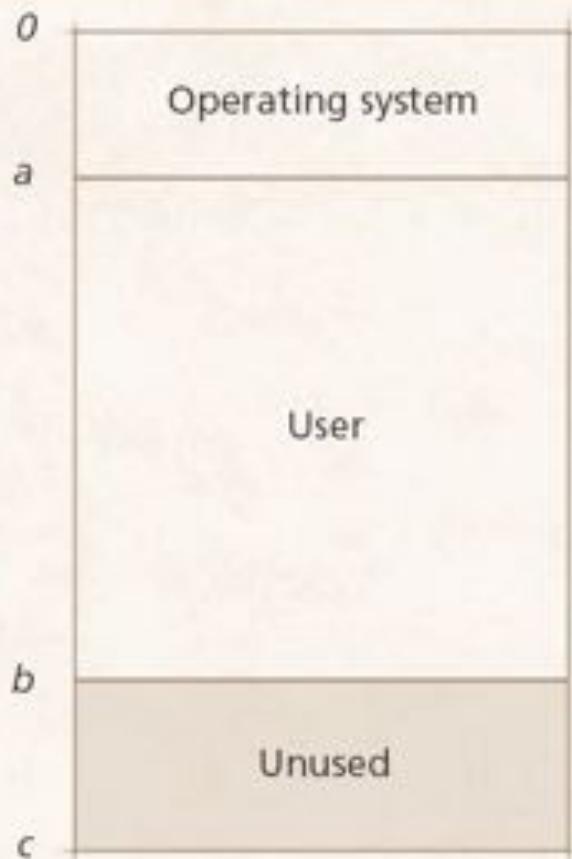
- Ways of organizing programs in memory
 - Contiguous allocation
 - Program must exist as a single block of contiguous addresses
 - Sometimes it is impossible to find a large enough block
 - Low overhead
 - Noncontiguous allocation
 - Program divided into chunks called segments
 - Each segment can be placed in different part of memory
 - Easier to find “holes” in which a segment will fit
 - Increased number of processes that can exist simultaneously in memory offsets the overhead incurred by this technique

9.7 Single-User Contiguous Memory Allocation

- One user had control of entire machine
 - Resources did not need to be shared
 - Originally no operating systems on computer
 - Programmer wrote code to perform resource management
 - Input-Output Control Systems (IOCS)
 - Libraries of prewritten code to manage I/O devices
 - Precursor to operating systems

9.7 Single-User Contiguous Memory Allocation

Figure 9.3 Single-user contiguous memory allocation.

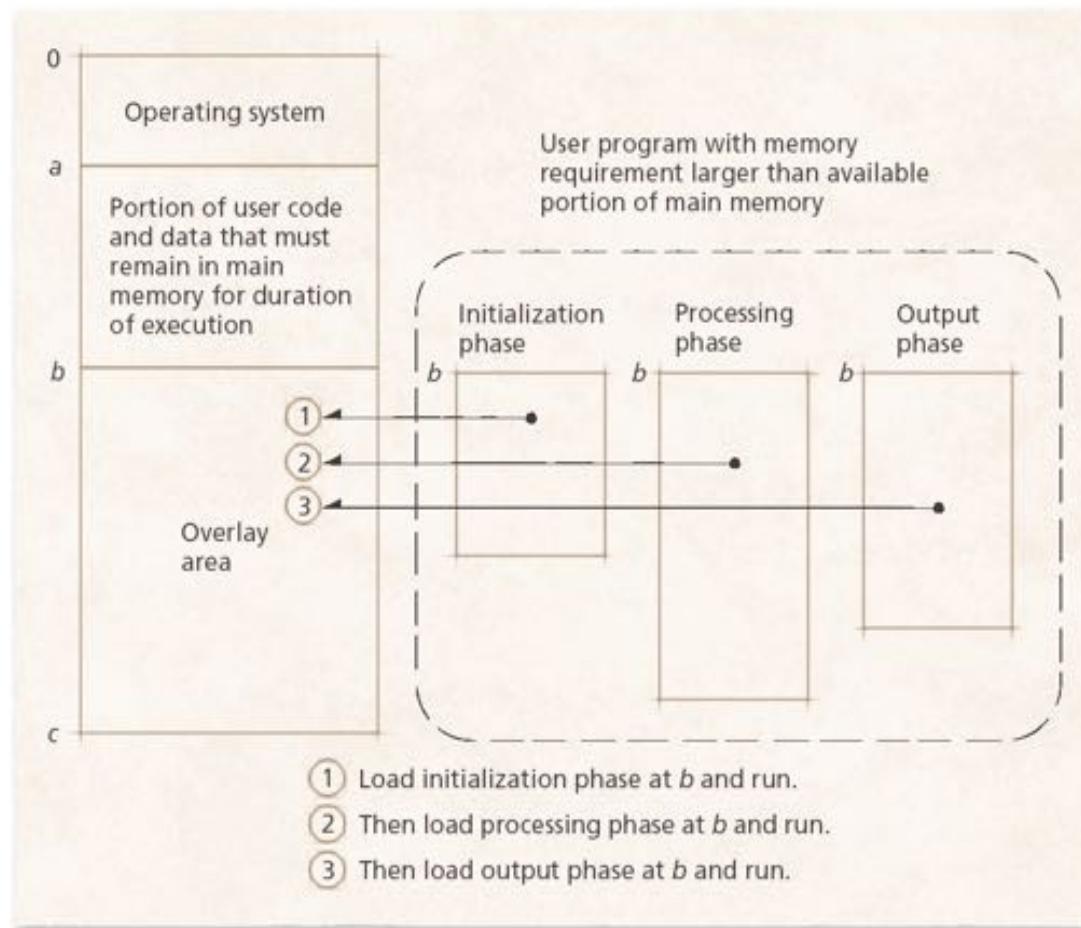


9.7.1 Overlays

- Overlays: Programming technique to overcome contiguous allocation limits
 - Program divided into logical sections
 - Only place currently active section in memory
 - Severe drawbacks
 - Difficult to organize overlays to make efficient use of main memory
 - Complicates modifications to programs
 - Virtual memory accomplishes similar goal
 - Like IOCS, VM shields programmers from complex issues such as memory management

9.7.1 Overlays

Figure 9.4 Overlay structure.

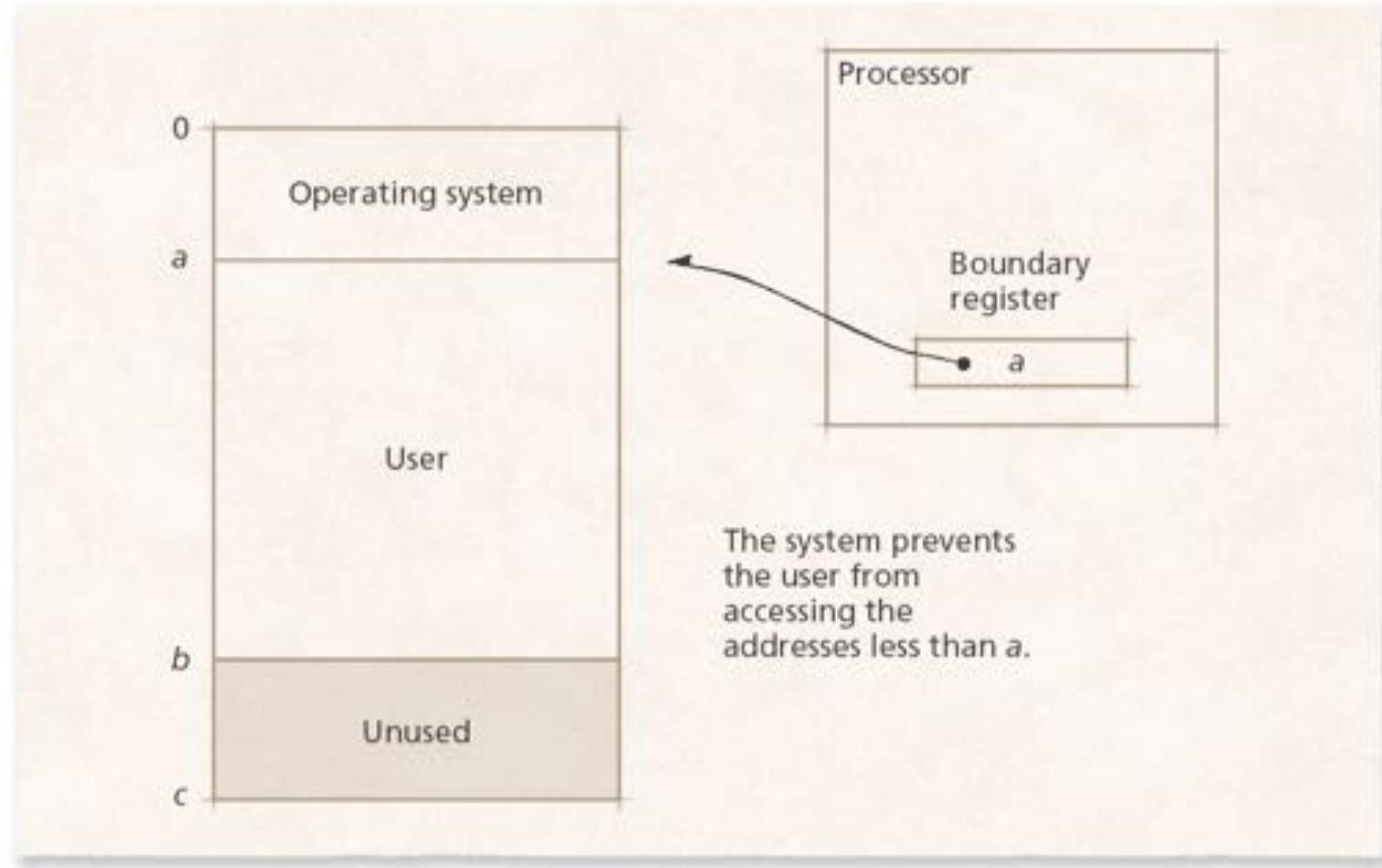


9.7.2 Protection in a Single-User Environment

- Operating system must not be damaged by programs
 - System cannot function if operating system overwritten
 - Boundary register
 - Contains address where program's memory space begins
 - Any memory accesses outside boundary are denied
 - Can only be set by privileged commands
 - Applications can access OS memory to execute OS procedures using system calls, which places the system in executive mode

9.7.2 Protection in a Single-User Environment

Figure 9.5 Memory protection with single-user contiguous memory allocation.



9.7.3 Single-Stream Batch Processing

- Early systems required significant setup time
 - Wasted time and resources
 - Automating setup and teardown improved efficiency
- Batch processing
 - Job stream processor reads job control languages
 - Defines each job and how to set it up

9.8 Fixed-Partition Multiprogramming

- I/O requests can tie up a processor for long periods
 - Multiprogramming is one solution
 - Process not actively using a processor should relinquish it to others
 - Requires several processes to be in memory at once

9.8 Fixed-Partition Multiprogramming

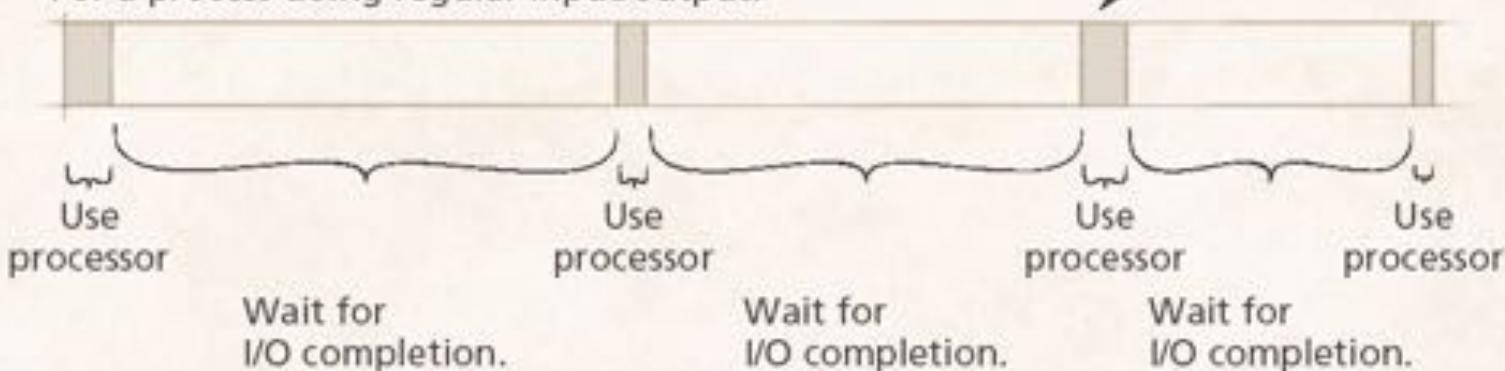
Figure 9.6 Processor utilization on a single-user system. [Note: In many single-user jobs, I/O waits are much longer relative to processor utilization periods indicated in this diagram.]

For a process doing intensive calculation:



Shaded area indicates
"Processor in use."

For a process doing regular input/output:

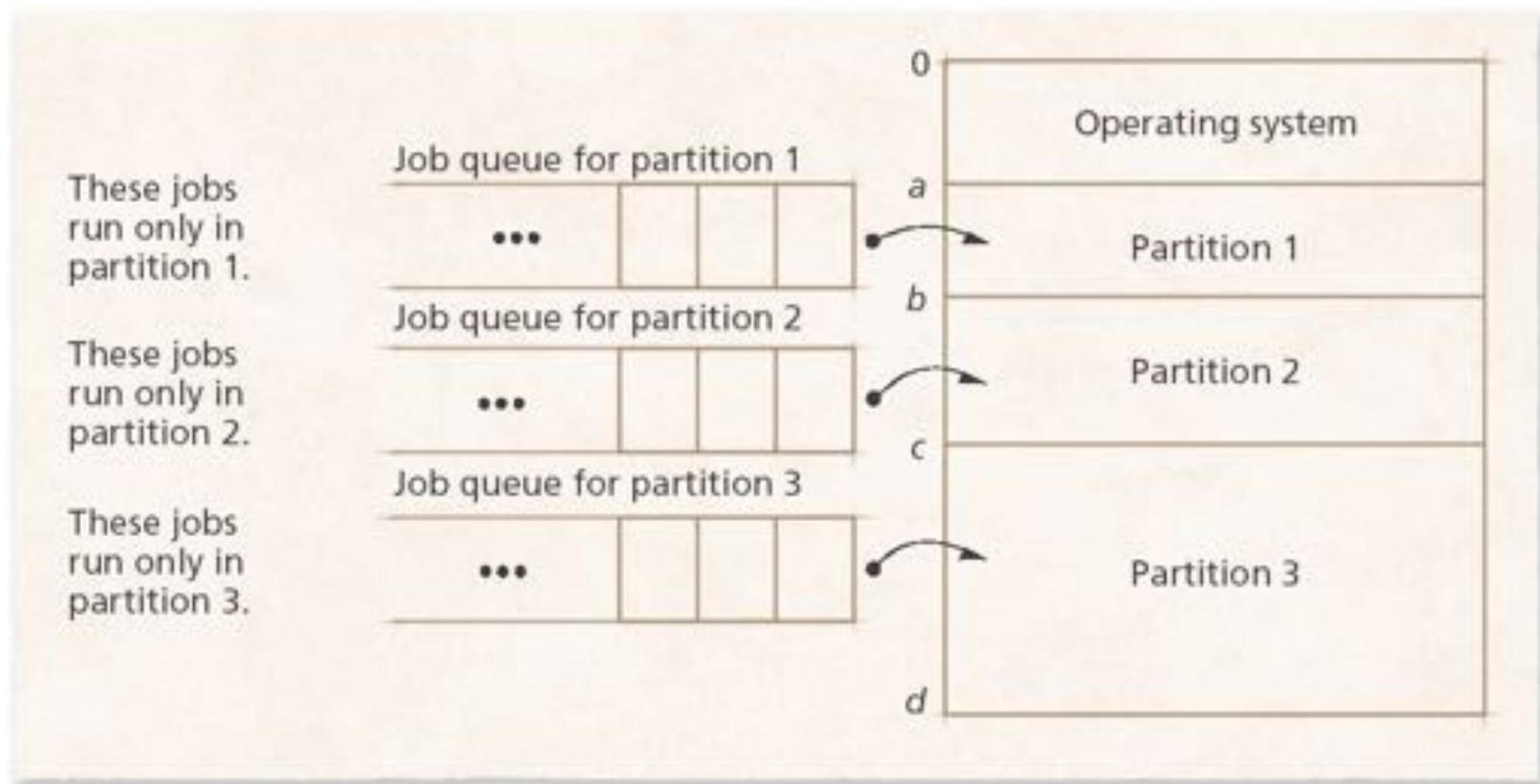


9.8 Fixed-Partition Multiprogramming

- **Fixed-partition multiprogramming**
 - Each active process receives a fixed-size block of memory
 - Processor rapidly switches between each process
 - Multiple boundary registers protect against damage

9.8 Fixed-Partition Multiprogramming

Figure 9.7 Fixed-partition multiprogramming with absolute translation and loading.

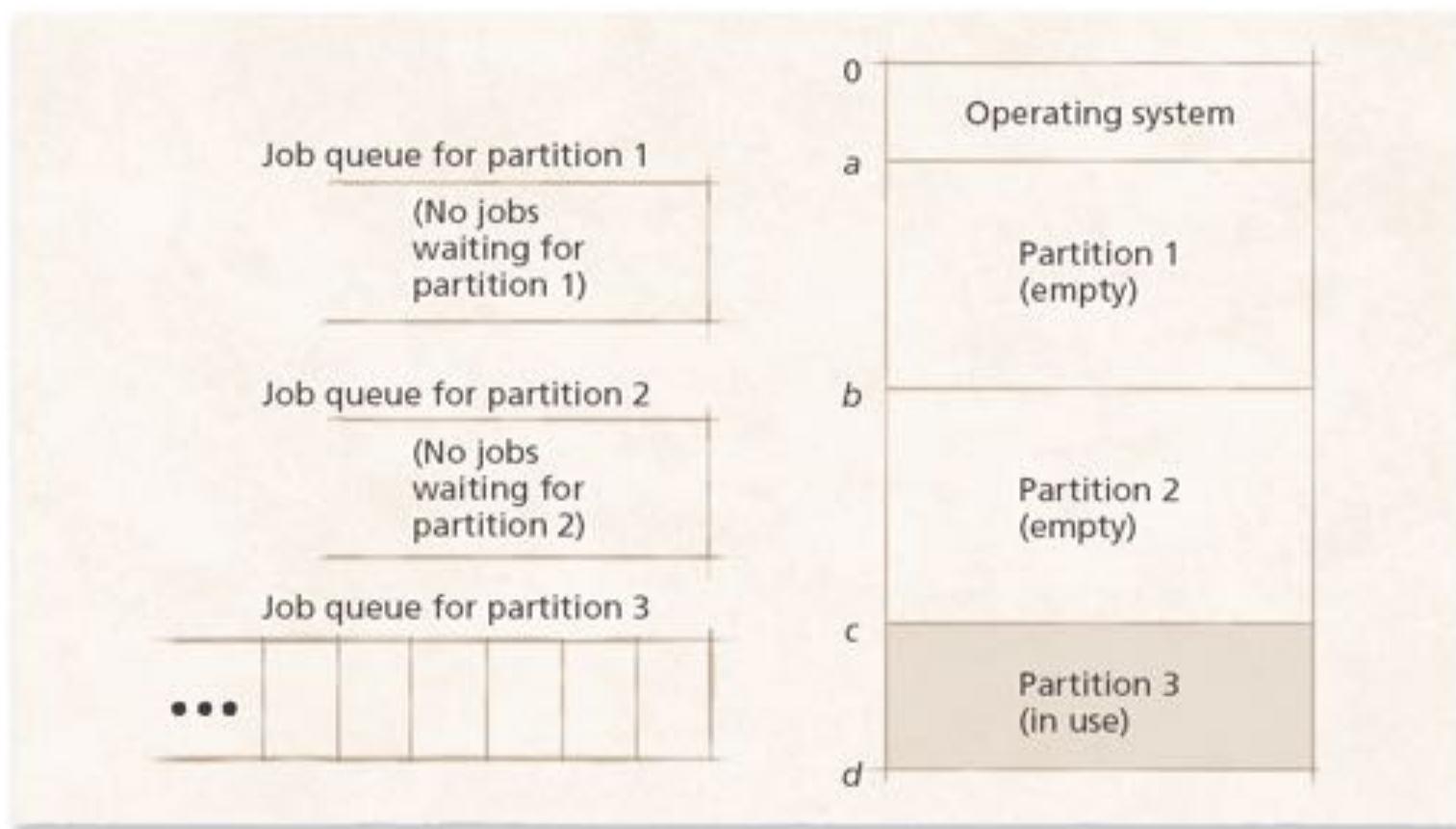


9.8 Fixed-Partition Multiprogramming

- Drawbacks to fixed partitions
 - Early implementations used absolute addresses
 - If the requested partition was full, code could not load
 - Later resolved by relocating compilers

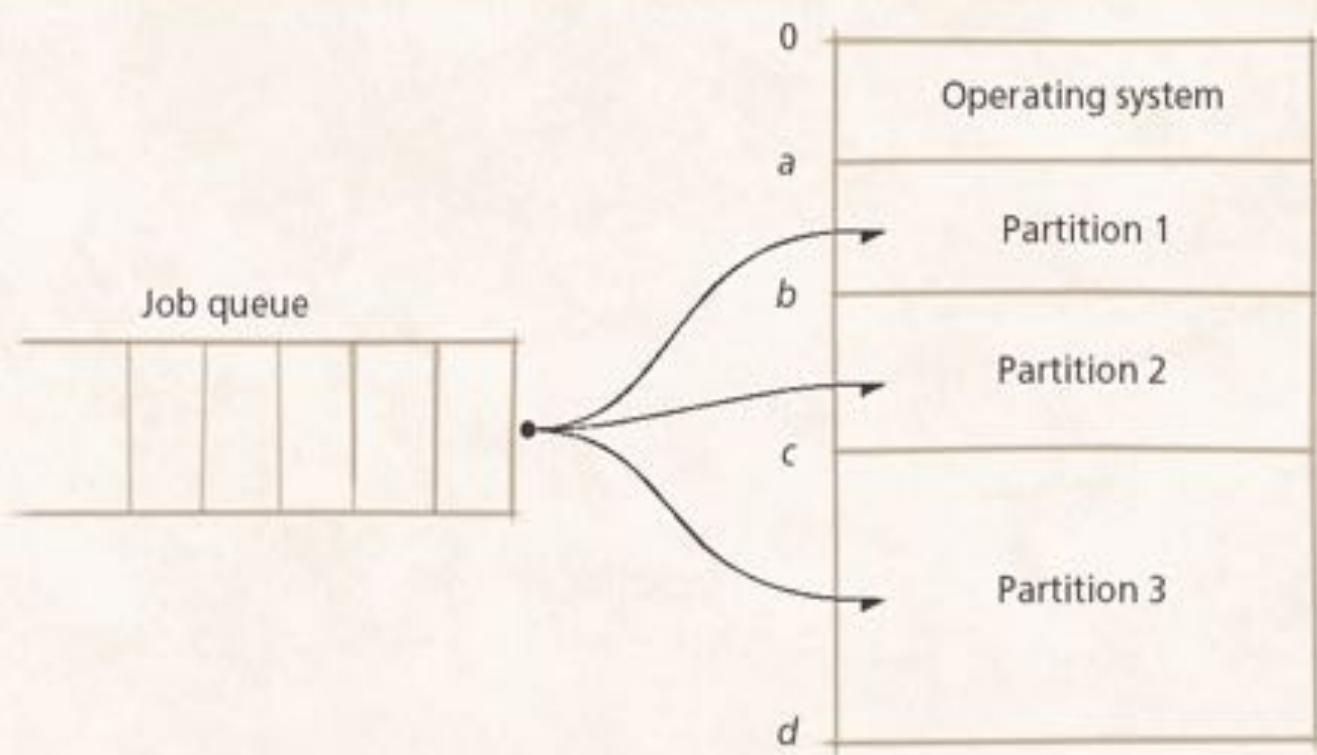
9.8 Fixed-Partition Multiprogramming

Figure 9.8 Memory waste under fixed-partition multiprogramming with absolute translation and loading.



9.8 Fixed-Partition Multiprogramming

Figure 9.8 Fixed-partition multiprogramming with relocatable translation and loading.



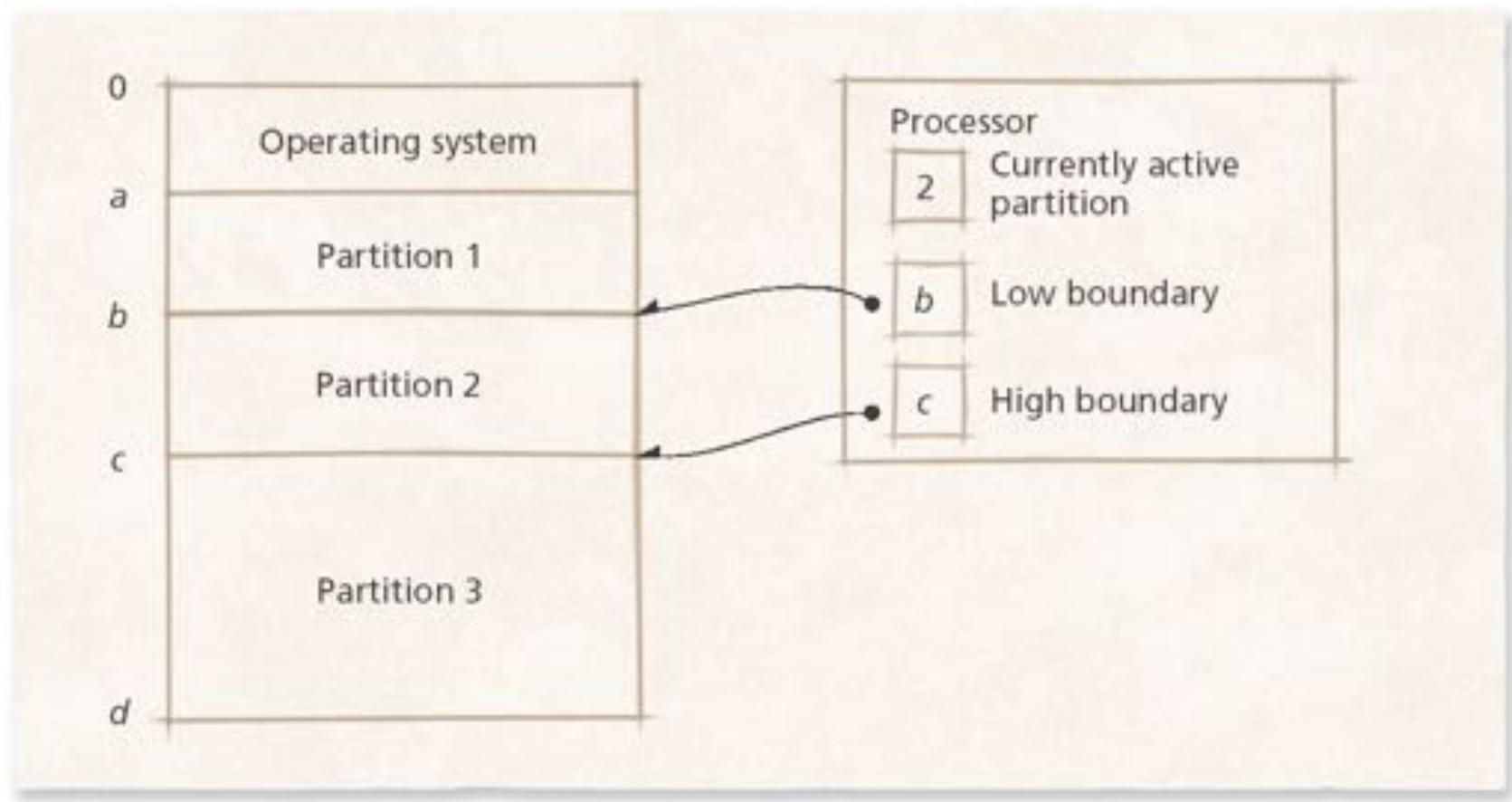
A job may be placed in any available partition in which it fits.

9.8 Fixed-Partition Multiprogramming

- Protection
 - Can be implemented by boundary registers, called base and limit (also called low and high)

9.8 Fixed-Partition Multiprogramming

Figure 9.10 Memory protection in contiguous-allocation multiprogramming systems.

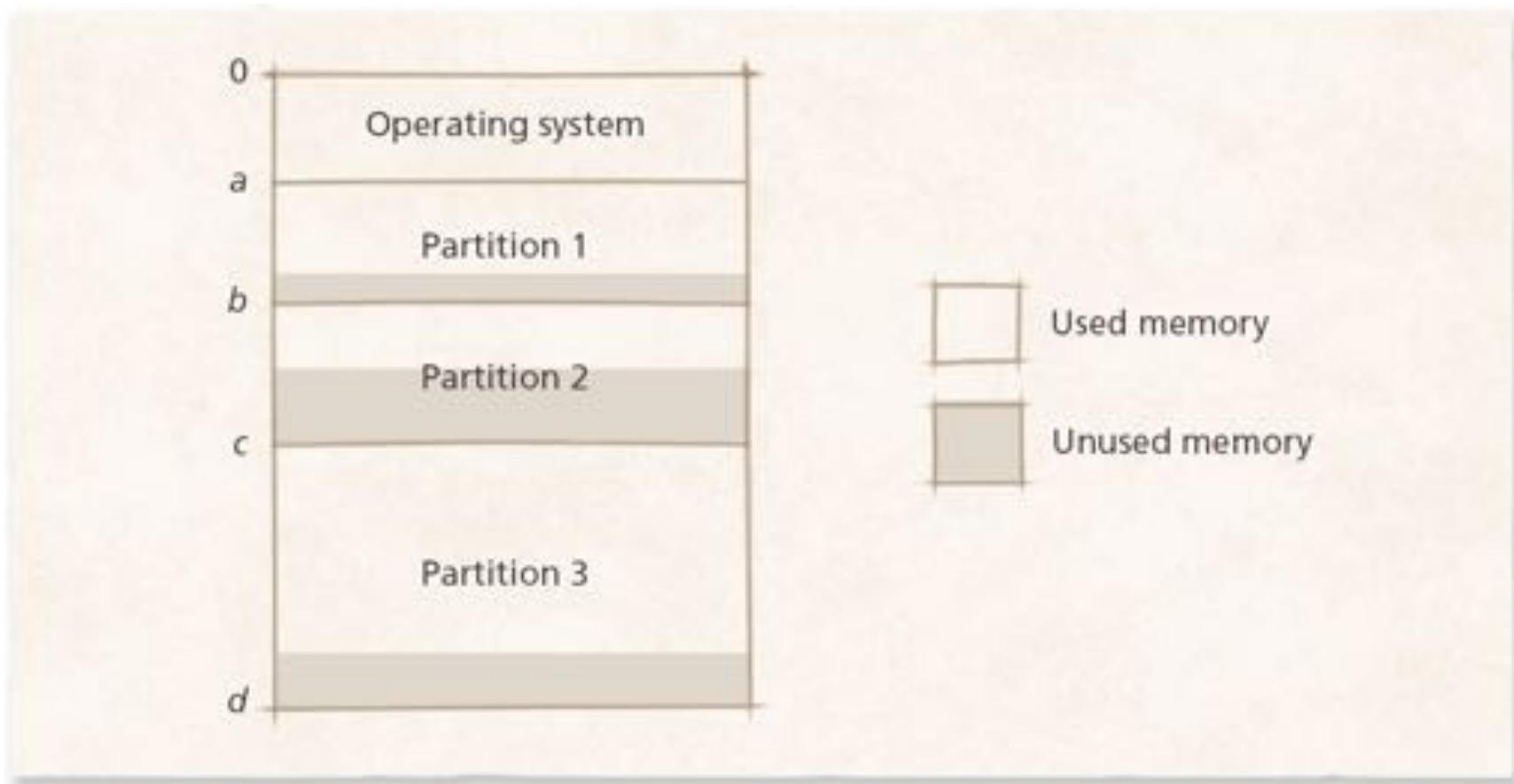


9.8 Fixed-Partition Multiprogramming

- Drawbacks to fixed partitions (Cont.)
 - Internal fragmentation
 - Process does not take up entire partition, wasting memory
 - Incurs more overhead
 - Offset by higher resource utilization

9.8 Fixed-Partition Multiprogramming

Figure 9.11 Internal fragmentation in a fixed-partition multiprogramming system.

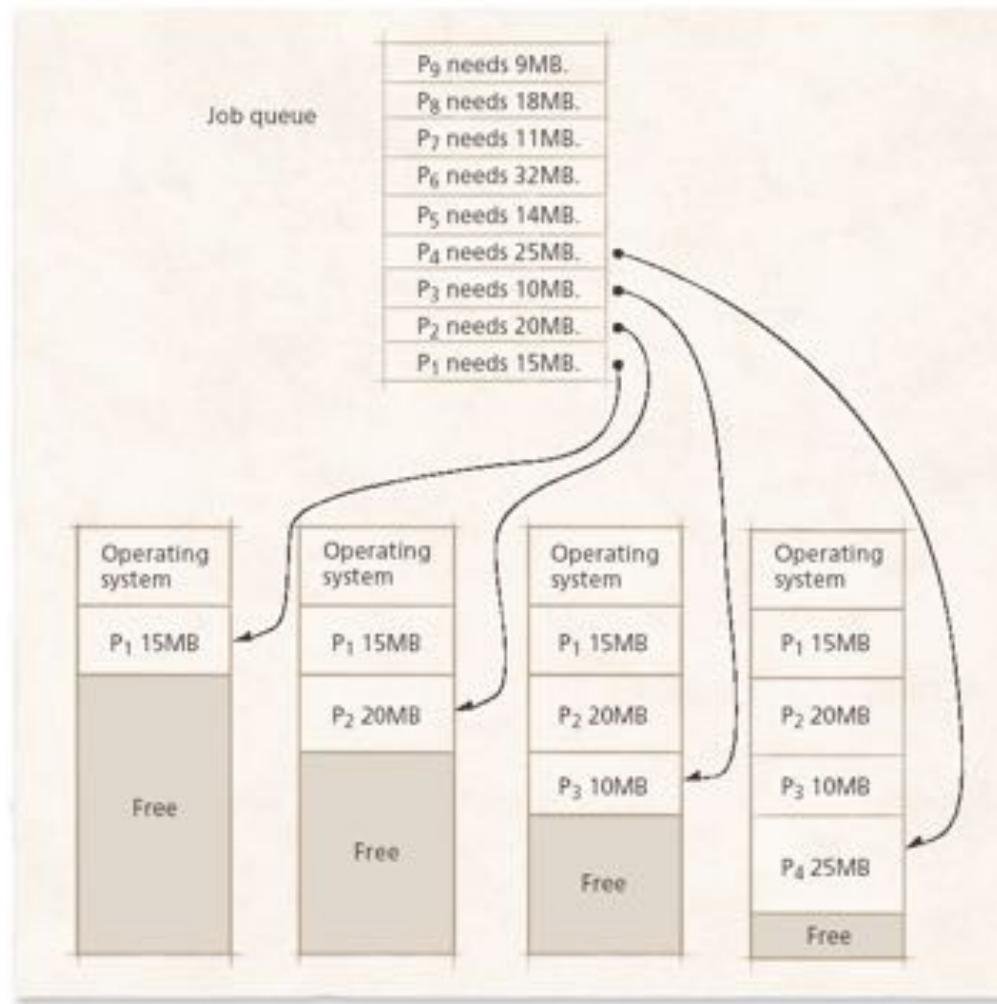


9.9 Variable-Partition Multiprogramming

- System designers found fixed partitions too restrictive
 - Internal fragmentation
 - Potential for processes to be too big to fit anywhere
 - Variable partitions designed as replacement

9.9 Variable-Partition Multiprogramming

Figure 9.12 Initial partition assignments in variable-partition programming.

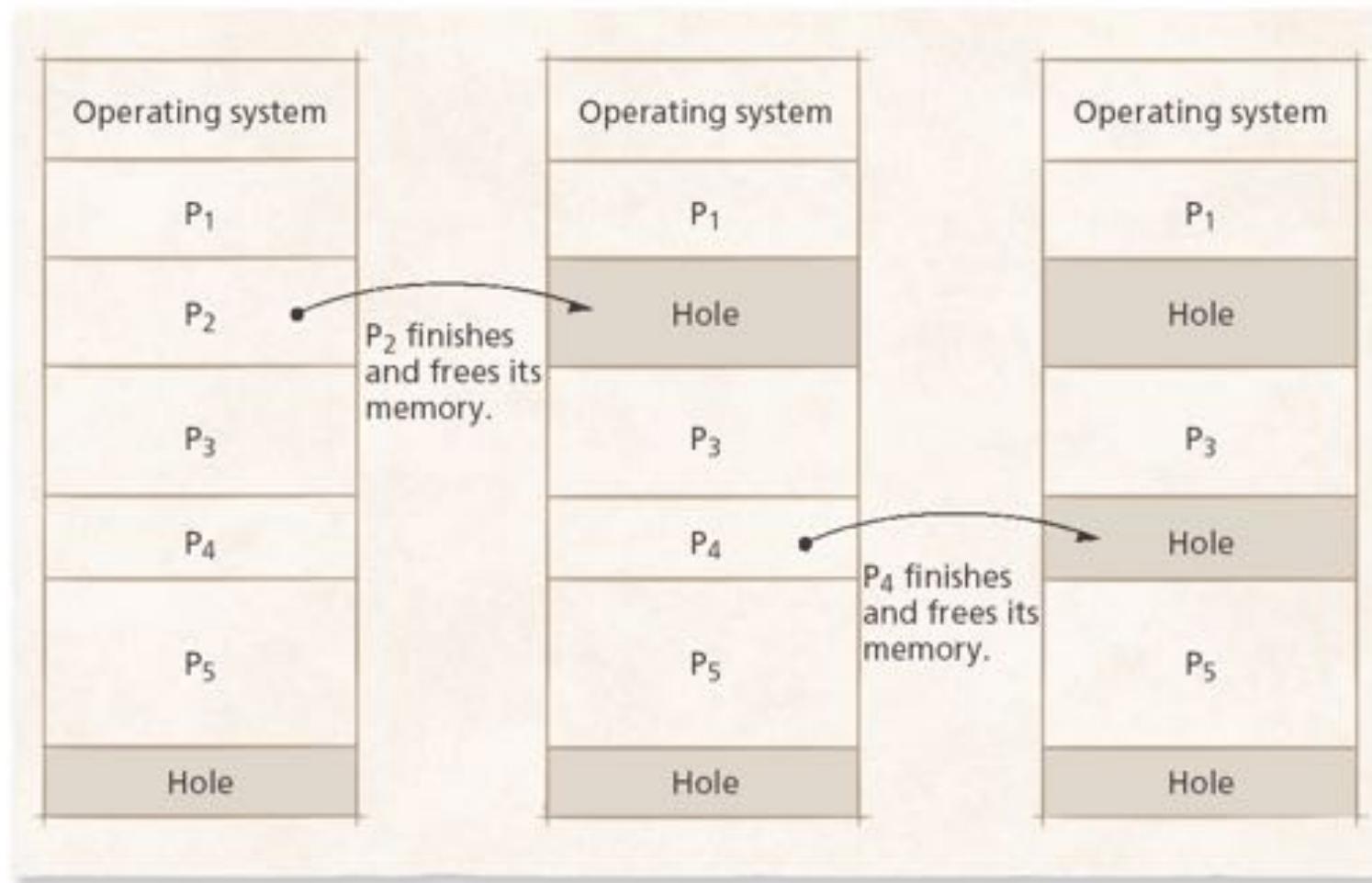


9.9.1 Variable-Partition Characteristics

- Jobs placed where they fit
 - No space wasted initially
 - Internal fragmentation impossible
 - Partitions are exactly the size they need to be
 - External fragmentation can occur when processes removed
 - Leave holes too small for new processes
 - Eventually no holes large enough for new processes

9.9.1 Variable-Partition Characteristics

Figure 9.13 Memory “holes” in variable-partition multiprogramming.

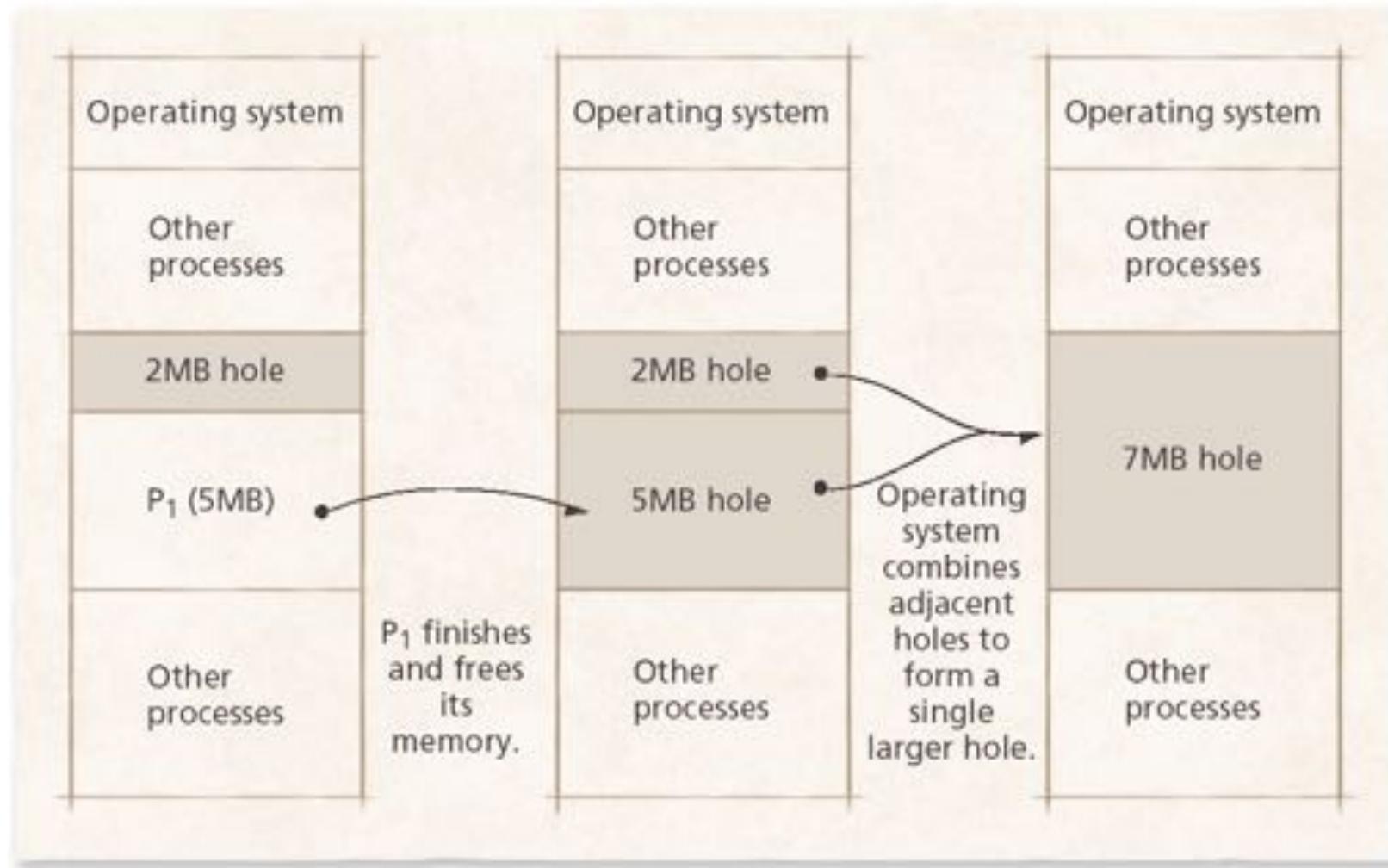


9.9.1 Variable-Partition Characteristics

- Several ways to combat external fragmentation
 - Coalescing
 - Combine adjacent free blocks into one large block
 - Often not enough to reclaim significant amount of memory
 - Compaction
 - Sometimes called garbage collection (not to be confused with GC in object-oriented languages)
 - Rearranges memory into a single contiguous block free space and a single contiguous block of occupied space
 - Makes all free space available
 - Significant overhead

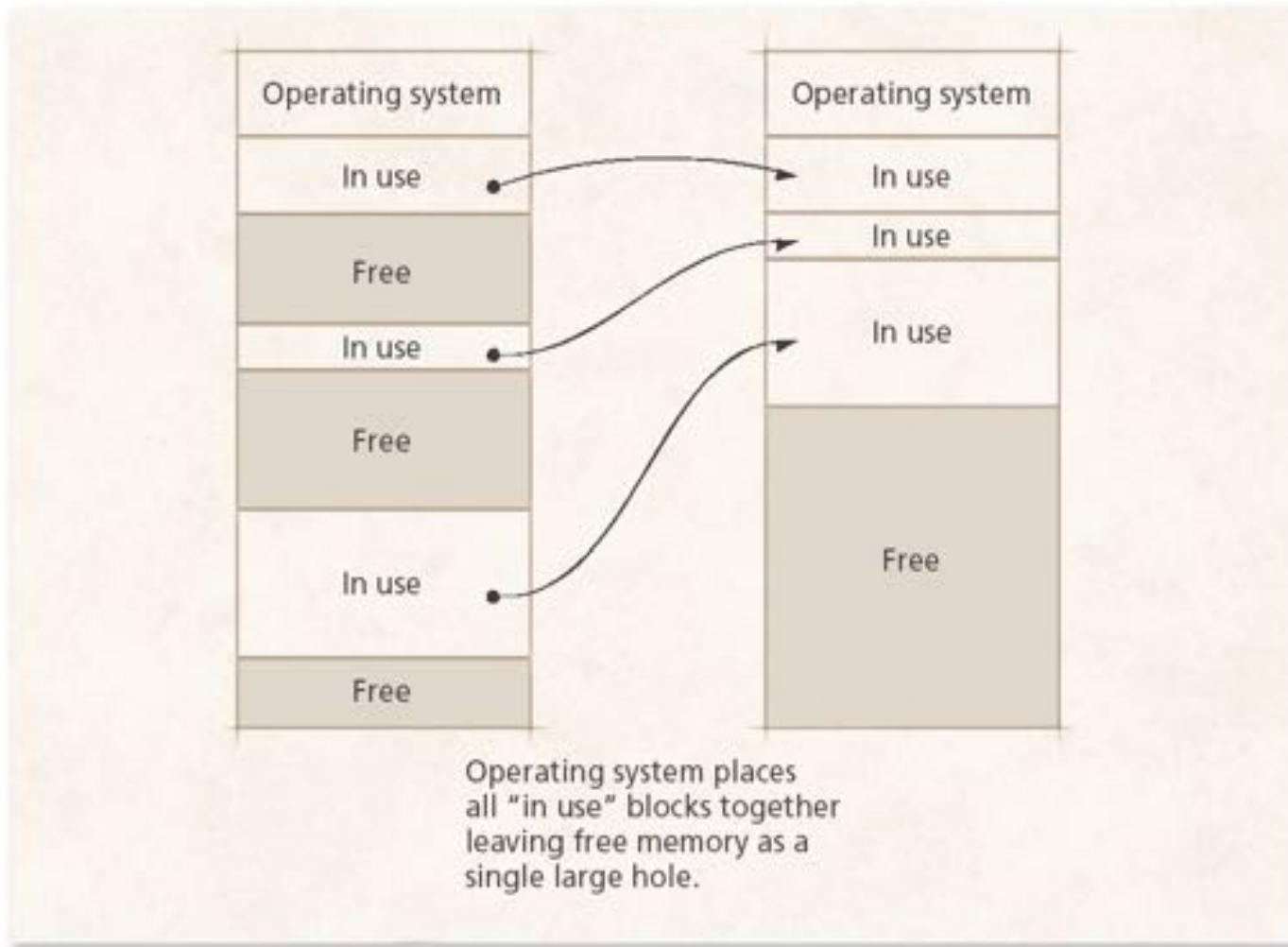
9.9.1 Variable-Partition Characteristics

Figure 9.14 Coalescing memory “holes” in variable-partition multiprogramming.



9.9.1 Variable-Partition Characteristics

Figure 9.15 Memory compaction in variable-partition multiprogramming.



9.9.2 Memory Placement Strategies

- Where to put incoming processes
 - First-fit strategy
 - Process placed in first hole of sufficient size found
 - Simple, low execution-time overhead
 - Best-fit strategy
 - Process placed in hole that leaves least unused space around it
 - More execution-time overhead
 - Worst-fit strategy
 - Process placed in hole that leaves most unused space around it
 - Leaves another large hole, making it more likely that another process can fit in the hole

9.9.2 Memory Placement Strategies

Figure 9.16 First-fit, best-fit and worst-fit memory placement strategies (Part 1 of 3).

(a) First-fit strategy

Place job in first memory hole on free memory list in which it will fit.

Free Memory List (Kept in random order.)

Start
address Length

a 16MB

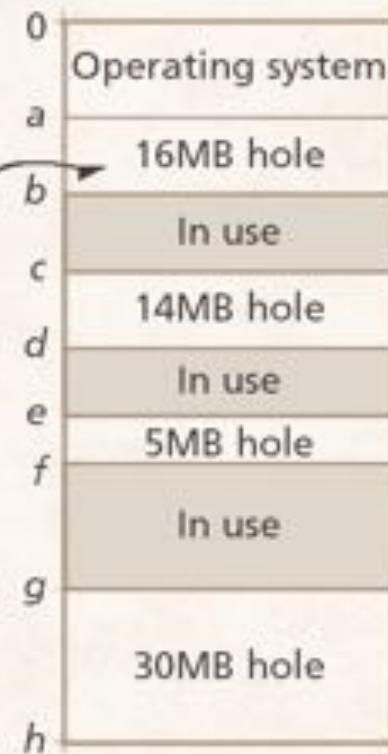
e 5MB

c 14MB

g 30MB

Request for
13MB

⋮



9.9.2 Memory Placement Strategies

Figure 9.16 First-fit, best-fit and worst-fit memory placement strategies (Part 2 of 3).

(b) Best-fit strategy

Place process in the smallest possible hole in which it will fit.

Free Memory List

Start address Length

e 5MB

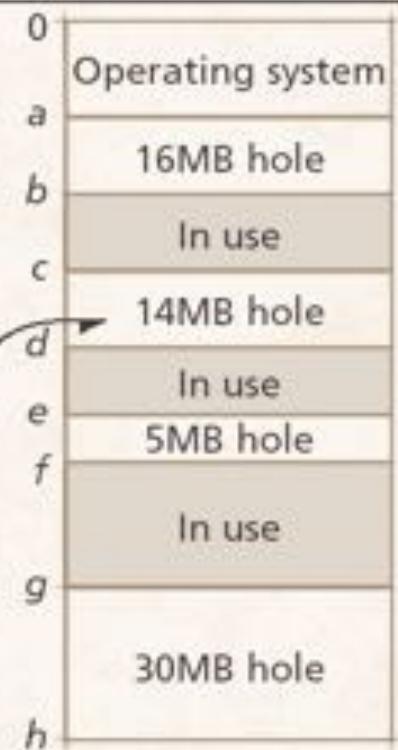
c 14MB

a 16MB

g 30MB

(Kept in ascending order by hole size.)

Request for 13MB	⋮



9.9.2 Memory Placement Strategies

Figure 9.16 First-fit, best-fit and worst-fit memory placement strategies (Part 3 of 3).

(c) Worst-fit strategy

Place process in the largest possible hole in which it will fit.

Free Memory List (Kept in descending order by hole size.)

Start address Length

g 30MB

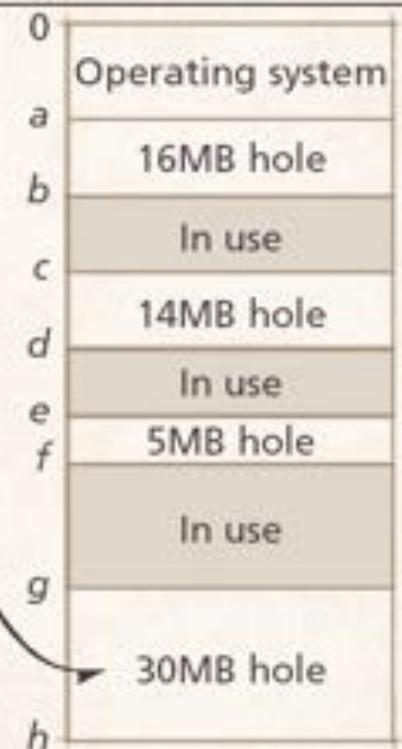
a 16MB

c 14MB

e 5MB

Request for 13MB

⋮

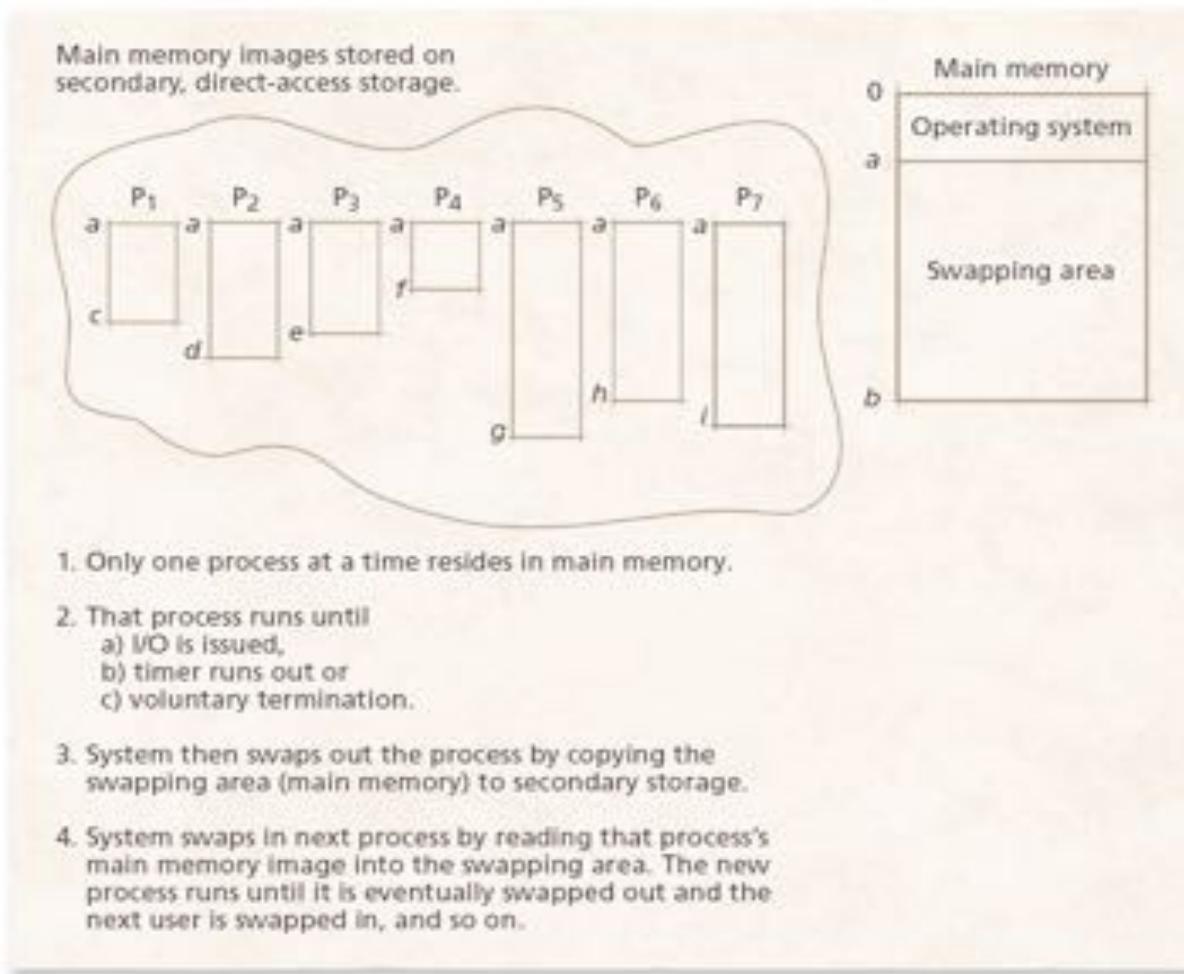


9.10 Multiprogramming with Memory Swapping

- Not necessary to keep inactive processes in memory
 - Swapping
 - Only put currently running process in main memory
 - Others temporarily moved to secondary storage
 - Maximizes available memory
 - Significant overhead when switching processes
 - Better yet: keep several processes in memory at once
 - Less available memory
 - Much faster response times
 - Similar to paging

9.10 Multiprogramming with Memory Swapping

Figure 9.17 Multiprogramming in a swapping system in which only a single process at a time is in main memory.



Chapter 10 – Virtual Memory Organization

Outline

- 10.1 Introduction
- 10.2 Virtual Memory: Basic Concepts
- 10.3 Block Mapping
- 10.4 Paging
 - 10.4.1 Paging Address Translation by Direct Mapping
 - 10.4.2 Paging Address Translation by Associative Mapping
 - 10.4.3 Paging Address Translation with Direct/Associative Mapping
 - 10.4.4 Multilevel Page Tables
 - 10.4.5 Inverted Page Tables
 - 10.4.6 Sharing in a Paging System
- 10.5 Segmentation
 - 10.5.1 Segmentation Address Translation by Direct Mapping
 - 10.5.2 Sharing in a Segmentation System
 - 10.5.3 Protection and Access Control in Segmentation Systems
- 10.6 Segmentation/Paging Systems
 - 10.6.1 Dynamic Address Translation in a Segmentation/Paging System
 - 10.6.2 Sharing and Protection in a Segmentation/Paging System
- 10.7 Case Study: IA-32 Intel Architecture Virtual Memory

Objectives

- After reading this chapter, you should understand:
 - the concept of virtual memory.
 - paged virtual memory systems.
 - segmented virtual memory systems.
 - combined segmentation/paging virtual memory systems.
 - sharing and protection in virtual memory systems.
 - the hardware that makes virtual memory systems feasible.
 - the IA-32 Intel architecture virtual memory implementation.

10.1 Introduction

- **Virtual memory**
 - Solves problem of limited memory space
 - Creates the illusion that more memory exists than is available in system
 - Two types of addresses in virtual memory systems
 - Virtual addresses
 - Referenced by processes
 - Physical addresses
 - Describes locations in main memory
 - Memory management unit (MMU)
 - Translates virtual addresses to physical address

10.1 Introduction

Figure 10.1 Evolution of memory organizations.

Real	Real	Virtual			
Single-user dedicated systems	Real memory multiprogramming systems		Virtual memory multiprogramming systems		
	Fixed-partition multi-programming	Variable-partition multi-programming	Pure paging	Pure segmentation	Combined paging and segmentation
Absolute	Re-locatable				

10.2 Virtual Memory: Basic Concepts

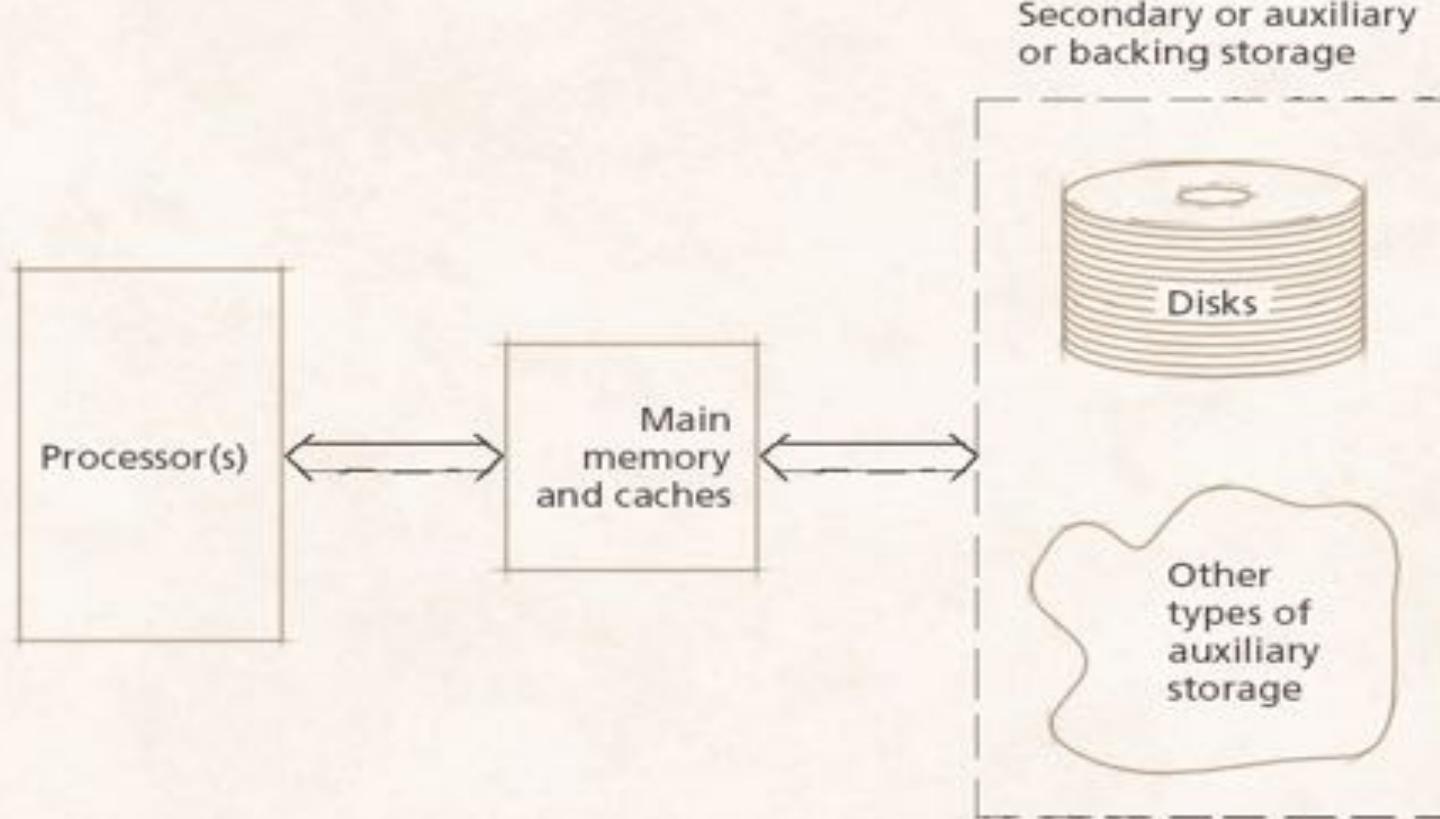
- Virtual address space, V
 - Range of virtual addresses that a process may reference
- Real address space, R
 - Range of physical addresses available on a particular computer system
- Dynamic address translation (DAT) mechanism
 - Converts virtual addresses to physical addresses during program execution

10.2 Virtual Memory: Basic Concepts

- $|V|$ is often much greater than $|R|$
 - OS must store parts of V for each process outside of main memory
 - Two-level storage
 - OS shuttles portions of V between main memory (and caches) and secondary storage

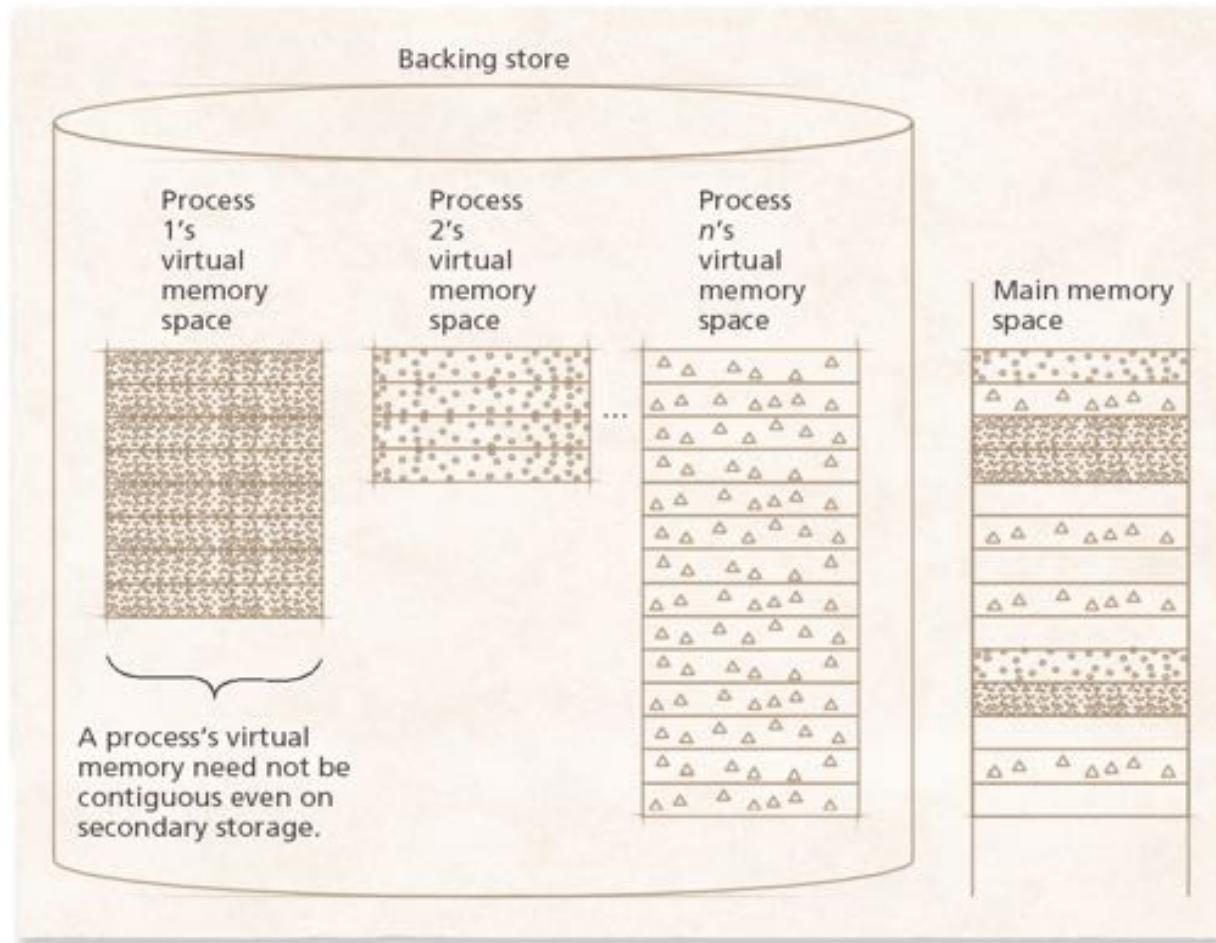
10.2 Virtual Memory: Basic Concepts

Figure 10.2 Two-level storage.



10.2 Virtual Memory: Basic Concepts

Figure 10.3 Pieces of address spaces exist in memory and in virtual storage.

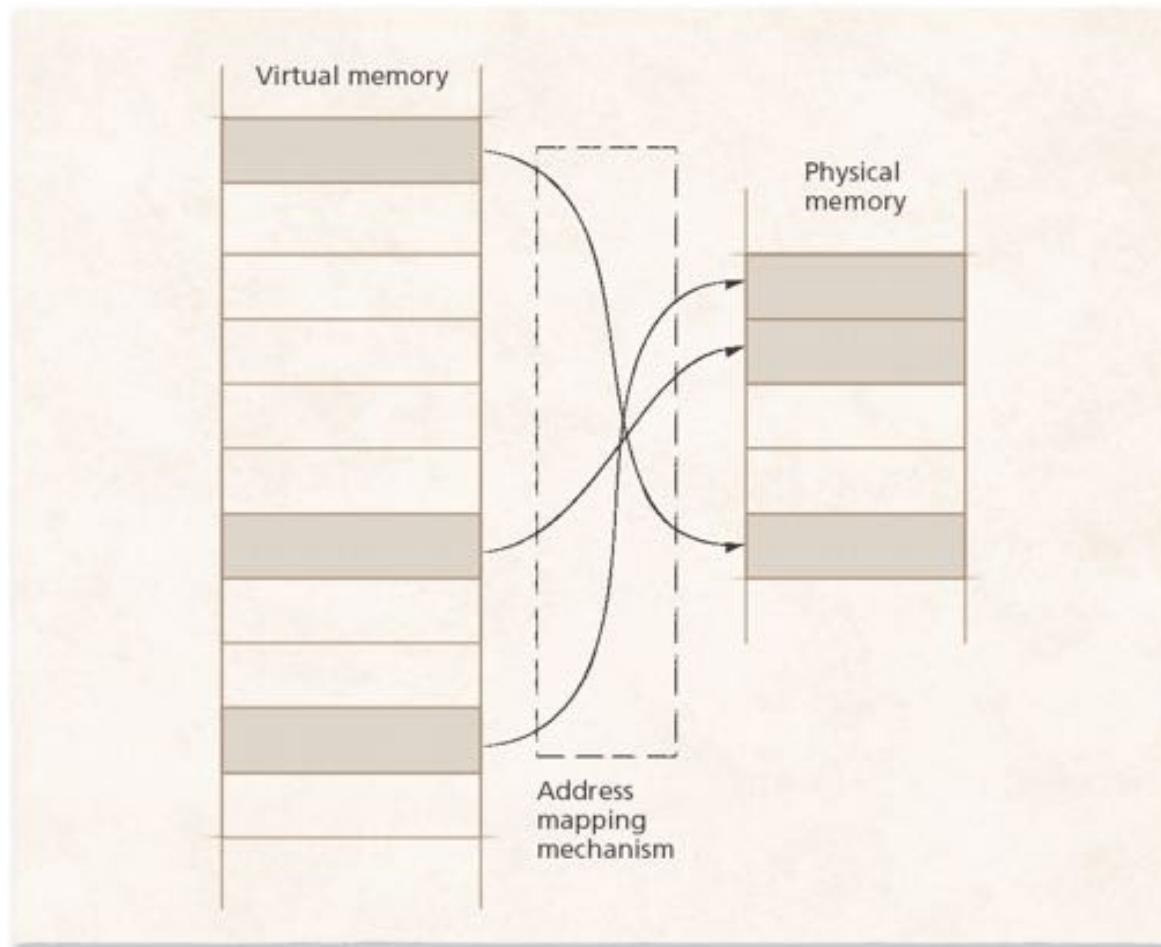


10.3 Block Mapping

- Address translation maps
 - Indicate which regions of a process's virtual address space, V , are currently in main memory and where they are located

10.3 Block Mapping

Figure 10.4 Mapping virtual addresses to real addresses.

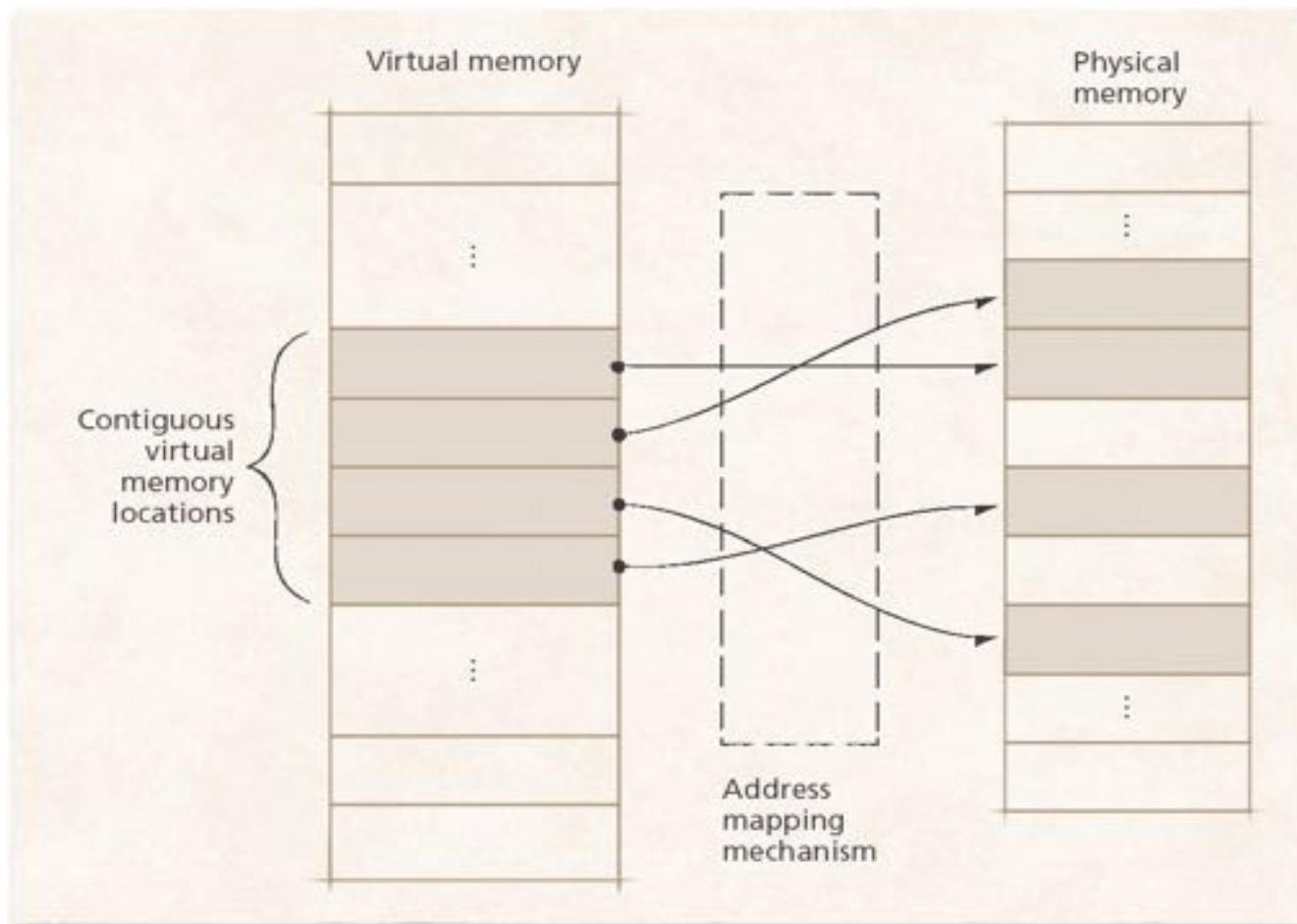


10.3 Block Mapping

- Artificial contiguity
 - Contiguous virtual addresses may not correspond to contiguous real memory addresses

10.3 Block Mapping

Figure 10.5 Artificial contiguity.

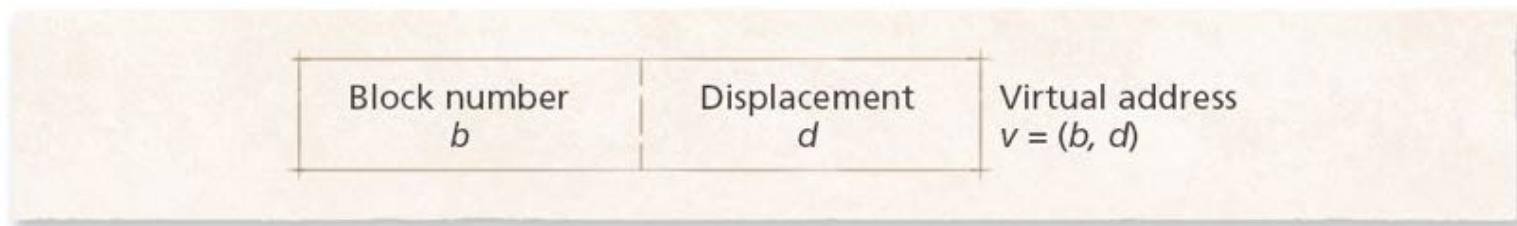


10.3 Block Mapping

- Pages
 - Blocks are fixed size
 - Technique is called paging
- Segments
 - Blocks maybe of different size
 - Technique is called segmentation
- Block mapping
 - System represents addresses as ordered pairs

10.3 Block Mapping

Figure 10.6 Virtual address format in a block mapping system.

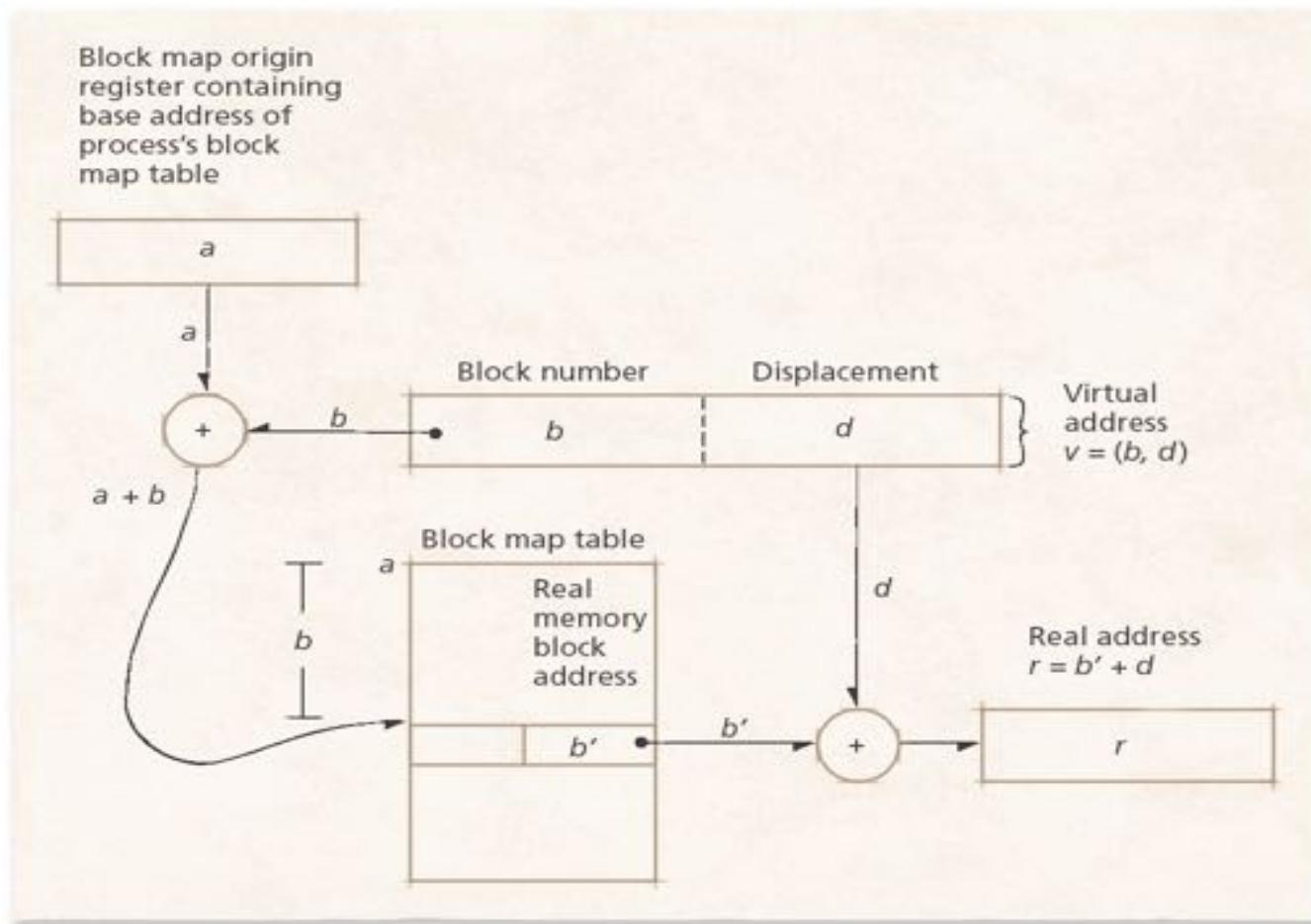


10.3 Block Mapping

- Given a virtual address $v = (b, d)$
 - Block map origin register stored in a
 - Block number, b , is added to a to locate the appropriate entry in the block map table
 - Block map table entry yields the address, b' , of the start of block b in main memory
 - Displacement d is added to b' to form the real address, r

10.3 Block Mapping

Figure 10.7 Virtual address translation with block mapping.



10.4 Paging

- Paging uses fixed-size block mapping
 - Virtual address in paging system is an ordered pair $v = (p, d)$
 - p is the number of the page in virtual memory on which the referenced item resides
 - d is the displacement from the start of page p at which the referenced item is located

10.4 Paging

Figure 10.8 Virtual address format in a pure paging system.

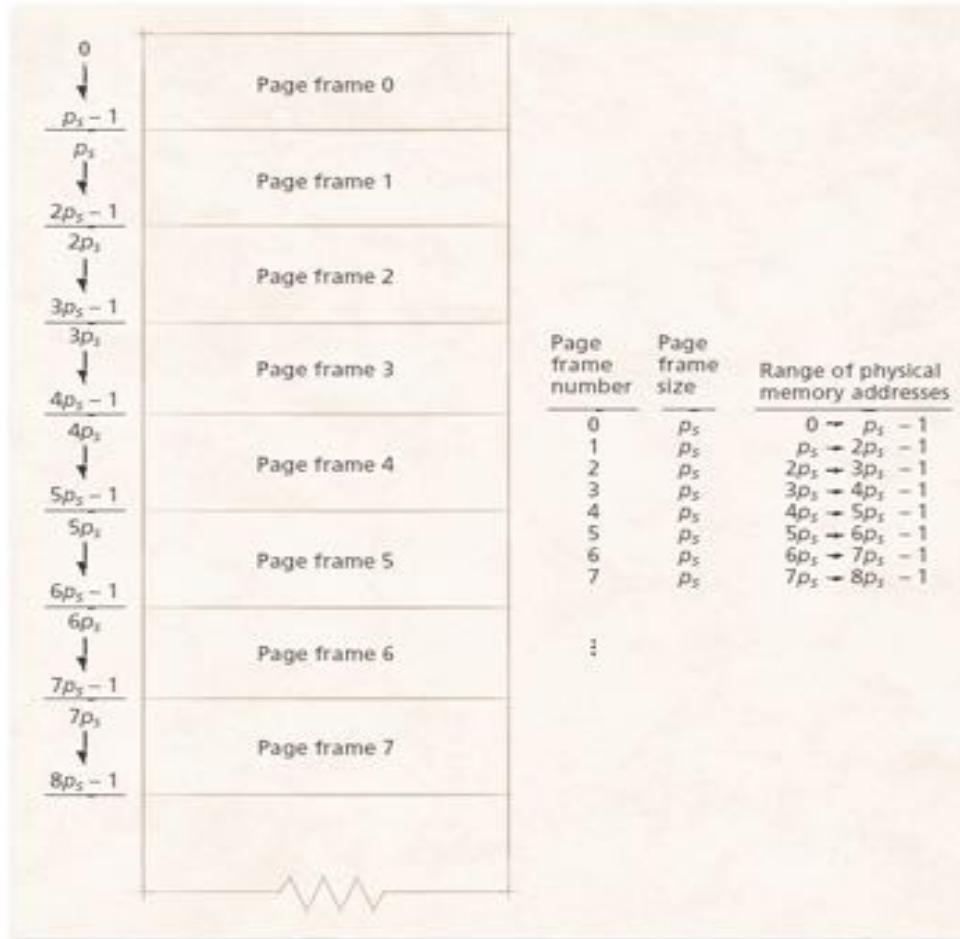
Page number p	Displacement d	Virtual address $v = (p, d)$
--------------------	---------------------	---------------------------------

10.4 Paging

- Page frame
 - Fixed-size block of main memory
 - Begins at a main memory address that is an integral multiple of fixed page size (p_s)

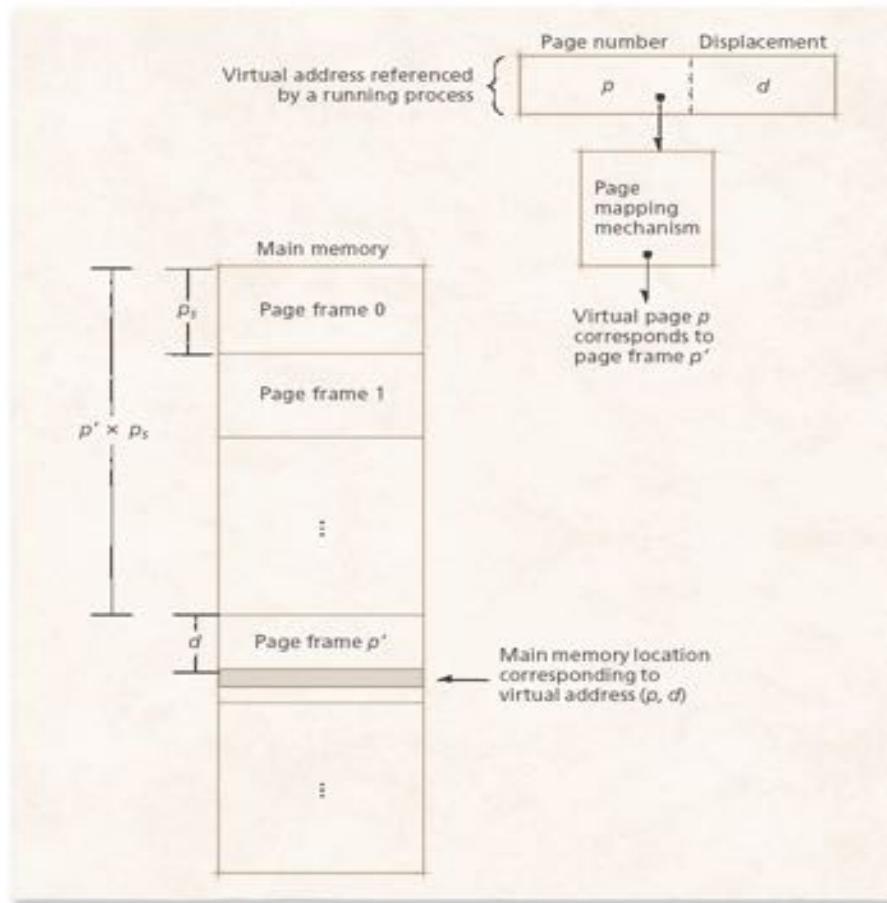
10.4 Paging

Figure 10.9 Main memory divided into page frames.



10.4 Paging

Figure 10.10 Correspondence between virtual memory addresses and physical memory addresses in a pure paging system.



10.4 Paging

- Page table entry (PTE)
 - Indicates that virtual page p corresponds to page frame p'
 - Contains a resident bit to indicate if page is in memory
 - If so, PTE stores the page's frame number
 - Otherwise, PTE stores the location of the page on secondary storage

10.4 Paging

Figure 10.11 Page table entry.

Page resident bit	Secondary storage address (if page is not in main memory)	Page frame number (if page is in main memory)
-------------------	--	--

r	s	p'
-----	-----	------

$r = 0$ if page is not in main memory

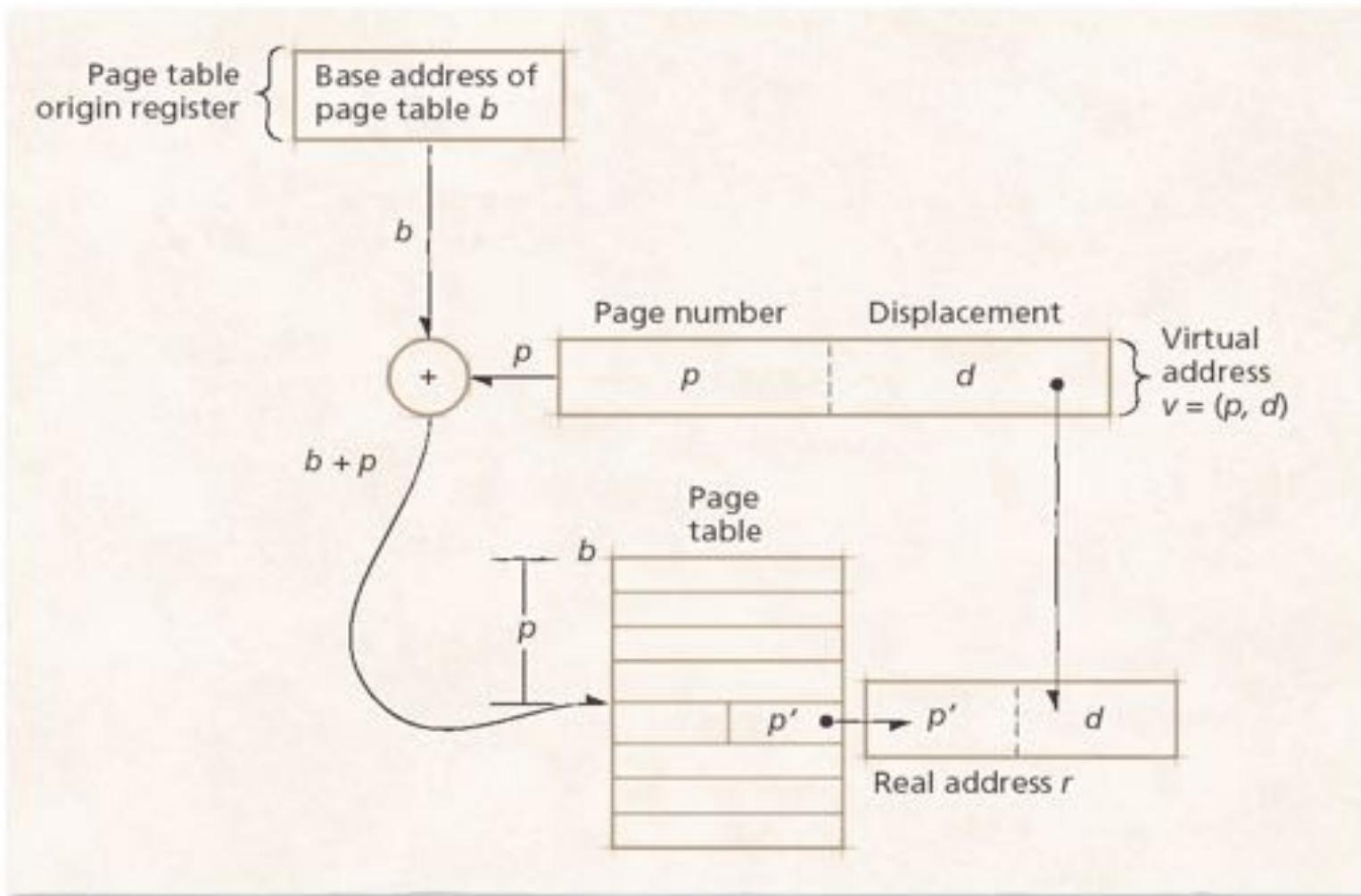
$r = 1$ if page is in main memory

10.4.1 Paging Address Translation by Direct Mapping

- Direct mapping
 - Dynamic address translation under paging is similar to block address translation
 - Process references virtual address $v = (p, d)$
 - DAT adds the process's page table base address, b , to referenced page number, p
 - $b + p$ forms the main memory address of the PTE for page p
 - System concatenates p' with displacement, d , to form real address, r

10.4.1 Paging Address Translation by Direct Mapping

Figure 10.12 Paging address translation by direct mapping.

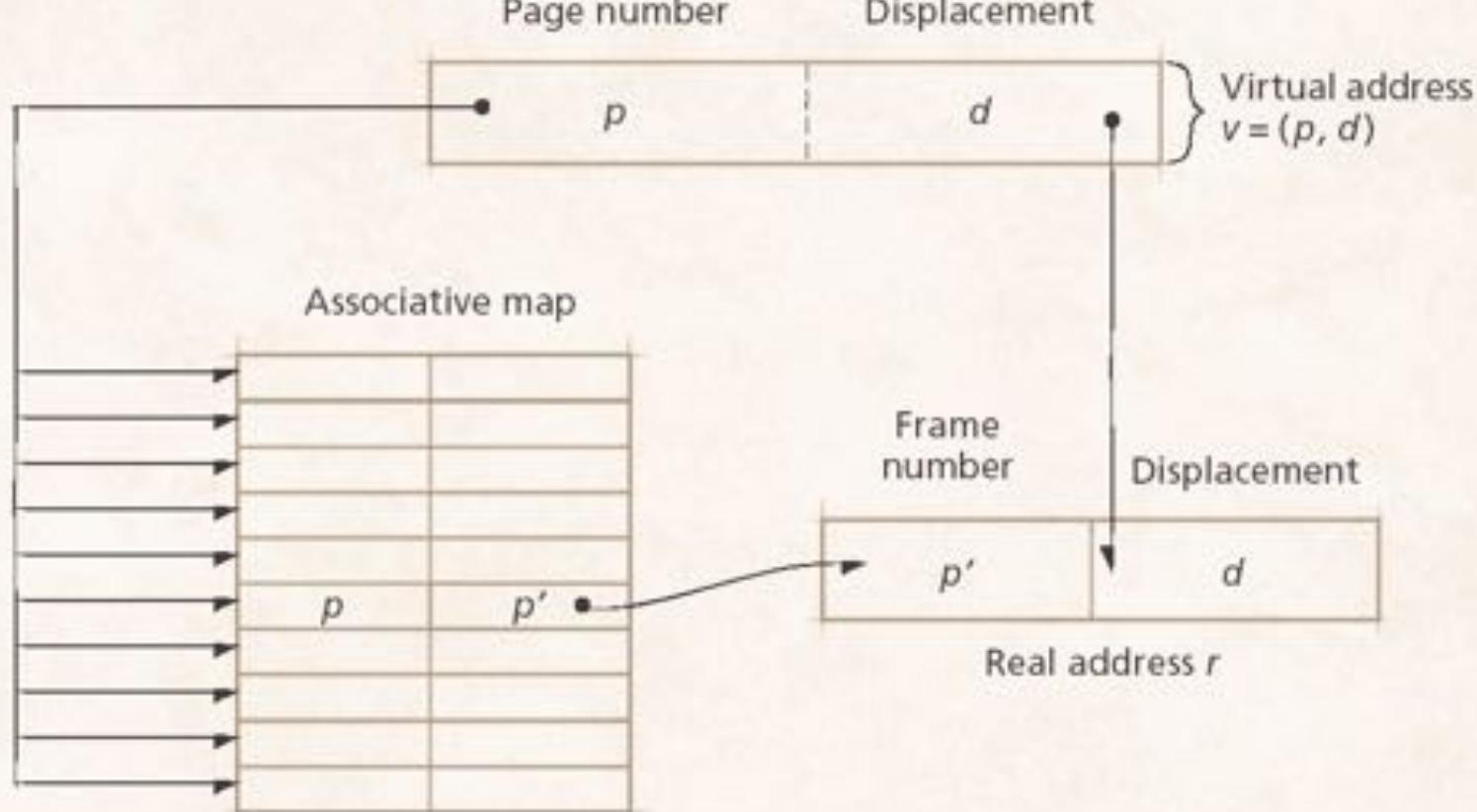


10.4.2 Paging Address Translation by Associative Mapping

- Maintaining entire page table in cache memory is often not viable
 - Due to cost of high-speed, location-addressed cache memory and relatively large size of programs
- Increase performance of dynamic address translation
 - Place entire page table into content-addressed associative memory
 - Every entry in associative memory is searched simultaneously
 - Content-addressed cache memory is also prohibitively expensive

10.4.2 Paging Address Translation by Associative Mapping

Figure 10.13 Paging address translation with pure associative mapping.

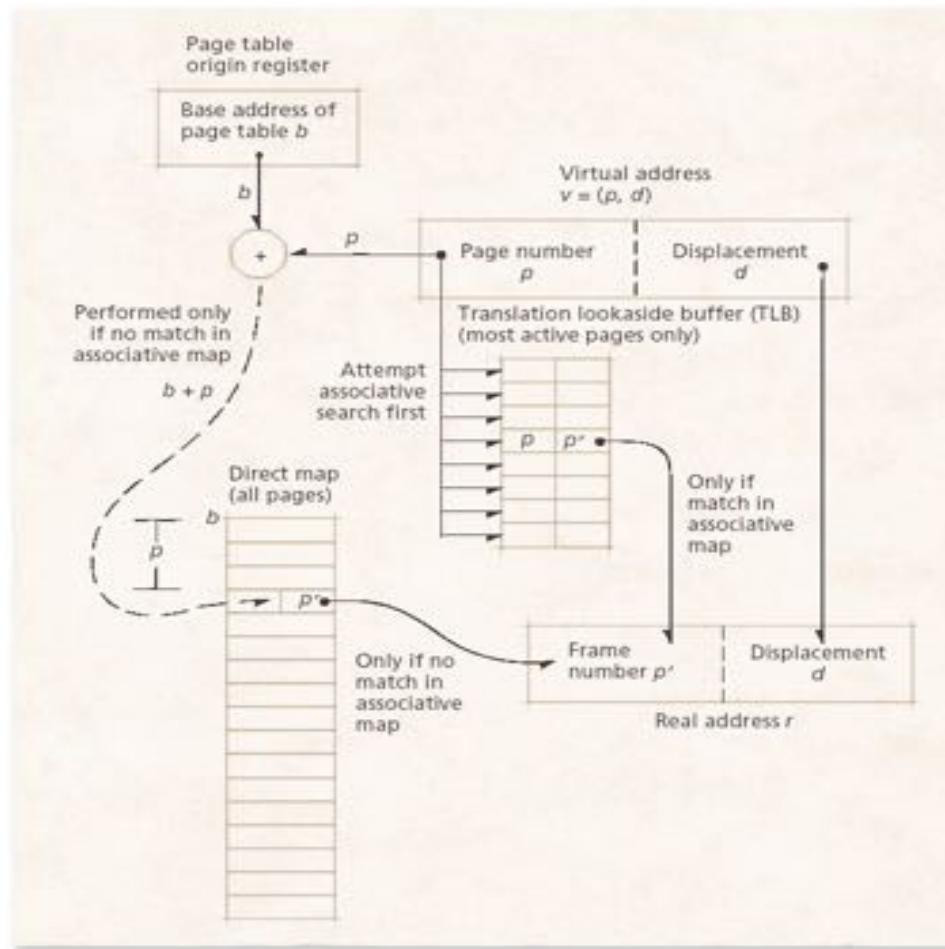


10.4.3 Paging Address Translation with Direct/Associative Mapping

- Compromise between cost and performance
 - Most PTEs are stored in direct-mapped tables in main memory
 - Most-recently-used PTEs are stored in high-speed set-associative cache memory called a Translation Lookaside Buffer (TLB)
 - If PTE is not found in TLB, the DAT mechanism searches the table in main memory
 - Can yield high performance with relatively small TLB due to locality
 - A page referenced by a process recently is likely to be referenced again soon

10.4.3 Paging Address Translation with Direct/Associative Mapping

Figure 10.14 Paging address translation with combined associative/direct mapping.

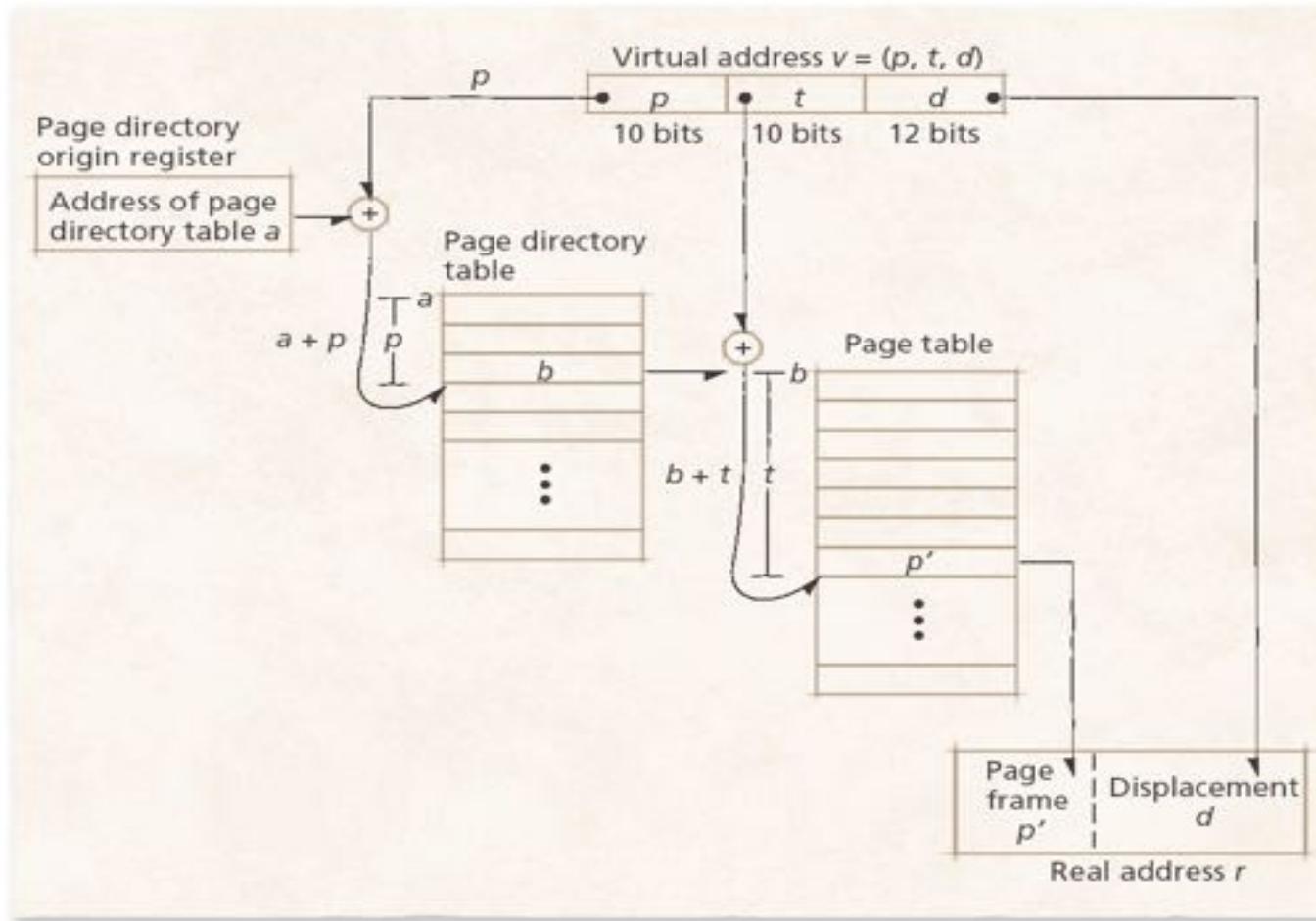


10.4.4 Multilevel Page Tables

- Multilevel page tables
 - System can store in discontiguous locations in main memory those portions of process's page table that the process is using
 - Hierarchy of page tables
 - Each level containing a table that stores pointers to tables in the level below
 - Bottom-most level comprised of tables containing address translations
 - Can reduce memory overhead compared to direct-mapping system

10.4.4 Multilevel Page Tables

Figure 10.15 Multilevel page address translation.

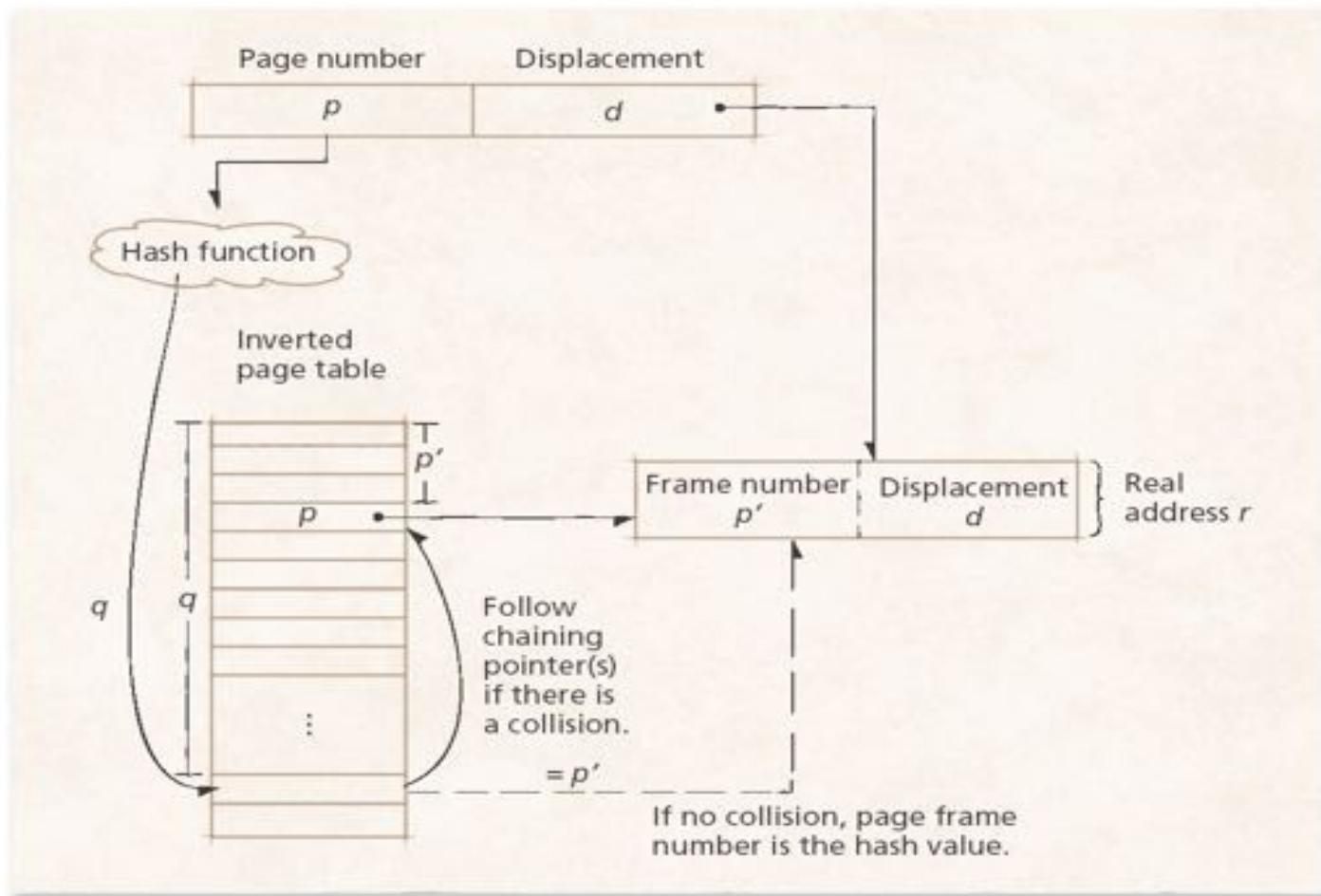


10.4.5 Inverted Page Tables

- Inverted page tables
 - One inverted page table stores one PTE in memory for each page frame in the system
 - Inverted relative to traditional page tables
 - Uses hash functions to map virtual page to inverted page table entry

10.4.5 Inverted Page Tables

Figure 10.16 Page address translation using inverted page tables.

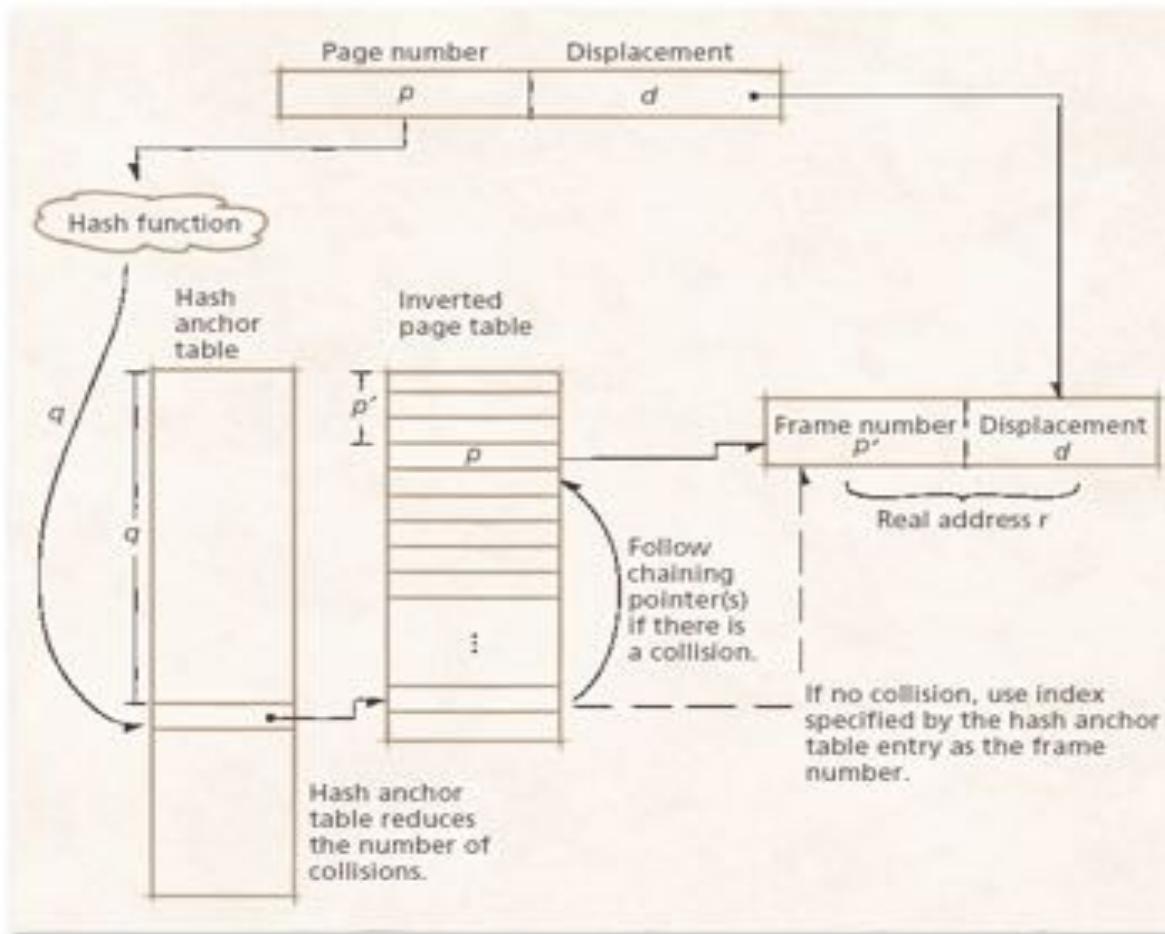


10.4.5 Inverted Page Tables

- Hashing can lead to collisions which increase address translation time by increasing the number of times memory must be accessed
- Collisions can be reduced by increasing the range of the hash function
 - Cannot increase the size of the inverted page table because it must store exactly one PTE for each page frame
 - Hash anchor table (HAT) increases the range of the hash function by adding another level of indirection
 - Size must be carefully chosen to balance table fragmentation and performance

10.4.5 Inverted Page Tables

Figure 10.17 Inverted page table using a hash anchor table.

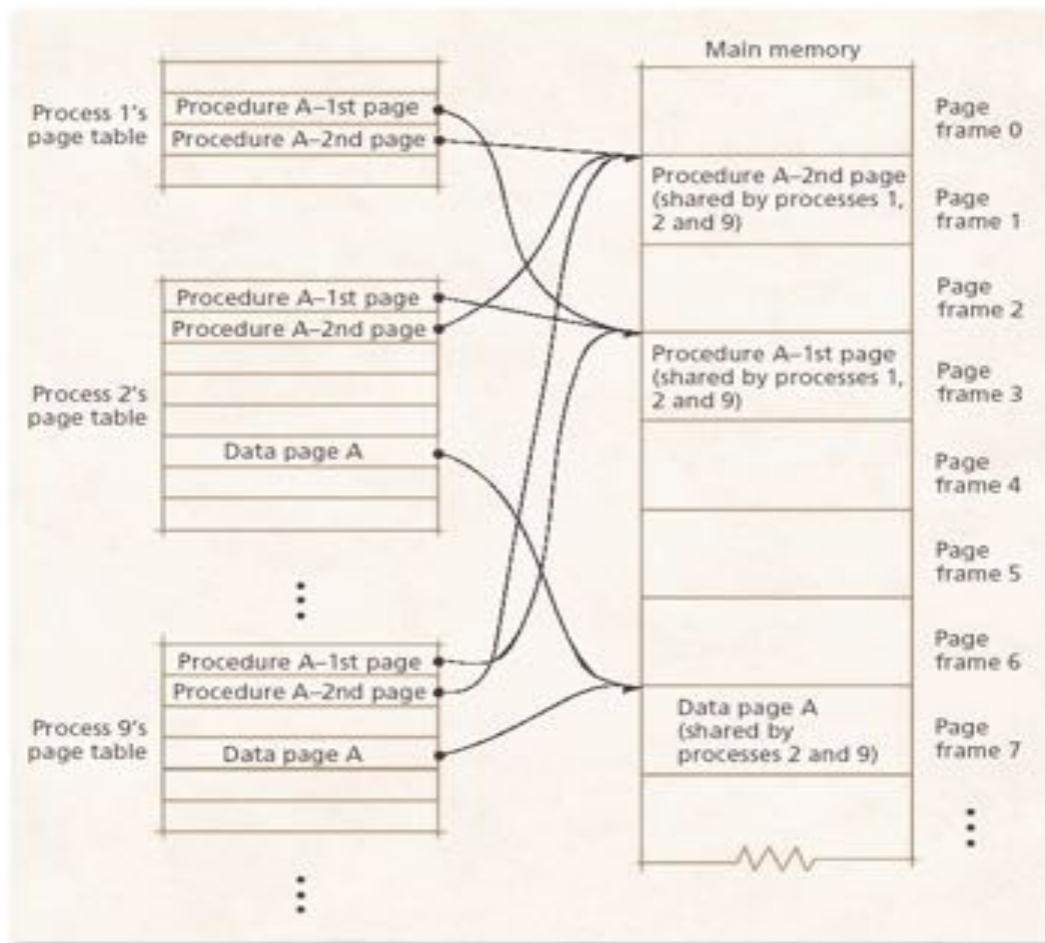


10.4.6 Sharing in a Paging System

- Sharing in multiprogramming systems
 - Reduces memory consumed by programs that use common data and/or instructions
 - Requires system identify each page as sharable or nonsharable

10.4.6 Sharing in a Paging System

Figure 10.18 Sharing in a pure paging system.

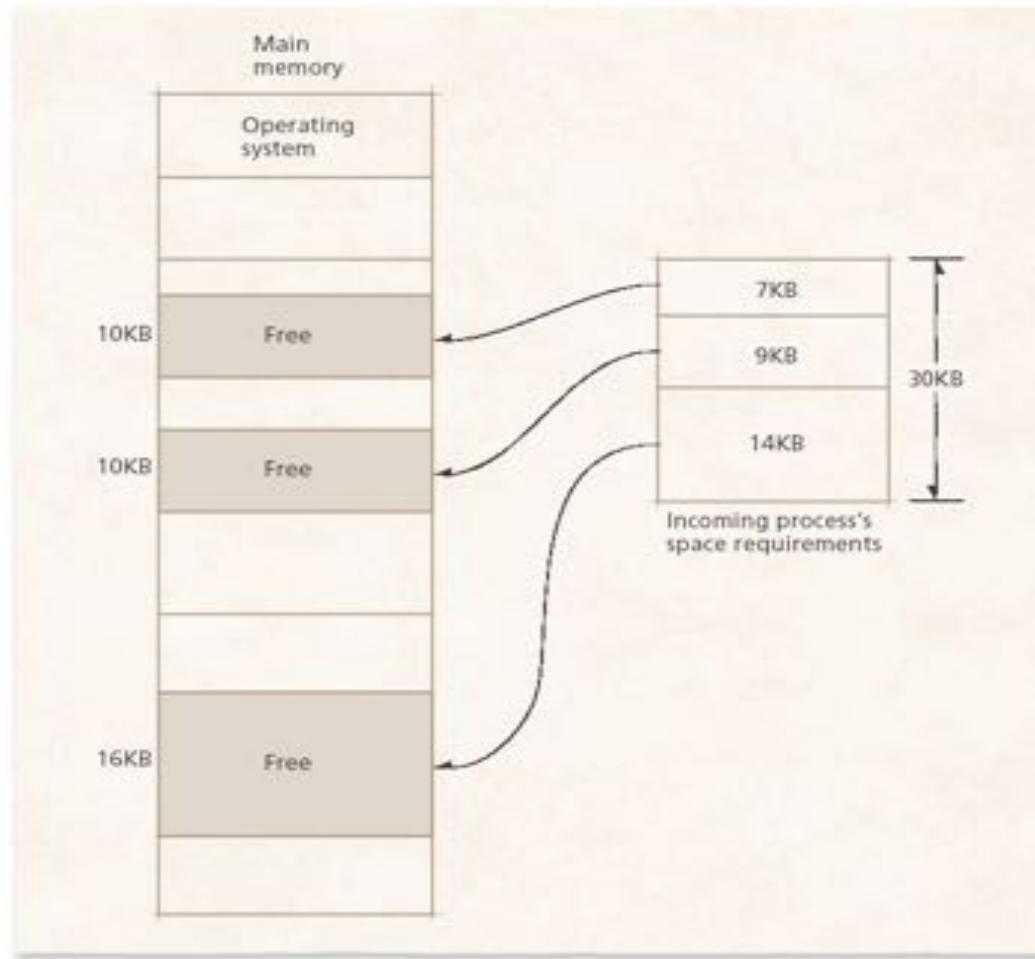


10.5 Segmentation

- Segment
 - Block of program's data and/or instructions
 - Contains meaningful portion of the program (e.g. procedure, array, stack)
 - Consists of contiguous locations
 - Segments need not be the same size nor must they be adjacent to one another in main memory
- A process may execute while its current instructions and referenced data are in segments in main memory

10.5 Segmentation

Figure 10.19 Noncontiguous memory allocation in a real memory segmentation system.



10.5 Segmentation

- Process references a virtual memory address $v = (s, d)$
 - s is the segment number in virtual memory
 - d is the displacement within segment s at which the referenced item is located

10.5 Segmentation

Figure 10.20 Virtual address format in a pure segmentation system.

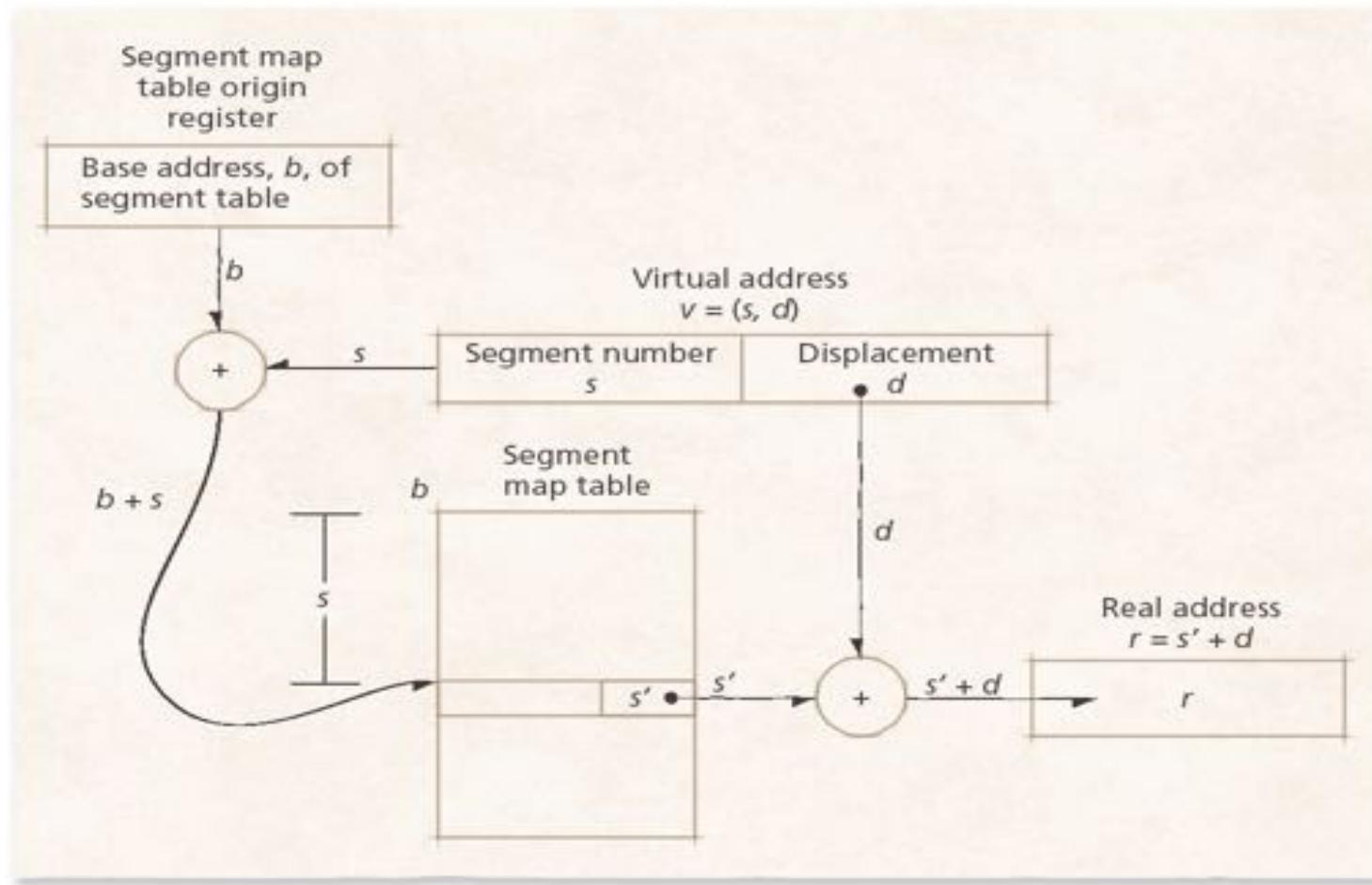
Segment number s	Displacement d	Virtual address $v = (s, d)$
-----------------------	---------------------	---------------------------------

10.5.1 Segmentation Address Translation by Direct Mapping

- Process references a virtual memory address $v = (s, d)$
 - DAT adds the process's segment map table base address, b , to referenced segment number, s
 - $b + s$ forms the main memory address of the segment map table entry for segment s
 - System adds s' to the displacement, d , to form real address, r

10.5.1 Segmentation Address Translation by Direct Mapping

Figure 10.21 Virtual address translation in a pure segmentation system.



10.5.1 Segmentation Address Translation by Direct Mapping

- Segment map table entry
 - Indicates that segment s starts at real memory address s'
 - Contains a resident bit to indicate if segment is in memory
 - If so, it stores the segment base address
 - Otherwise, it stores the location of the segment on secondary storage
 - Also contains a length field that indicates the size of the segment
 - Can be used to prevent a process from referencing addresses outside the segment

10.5.1 Segmentation Address Translation by Direct Mapping

Figure 10.22 Segment map table entry.

Segment resident bit	Secondary storage address (if not in main memory)	Segment length	Protection bits	Base address of segment (if in main memory)
r	a	l		s'

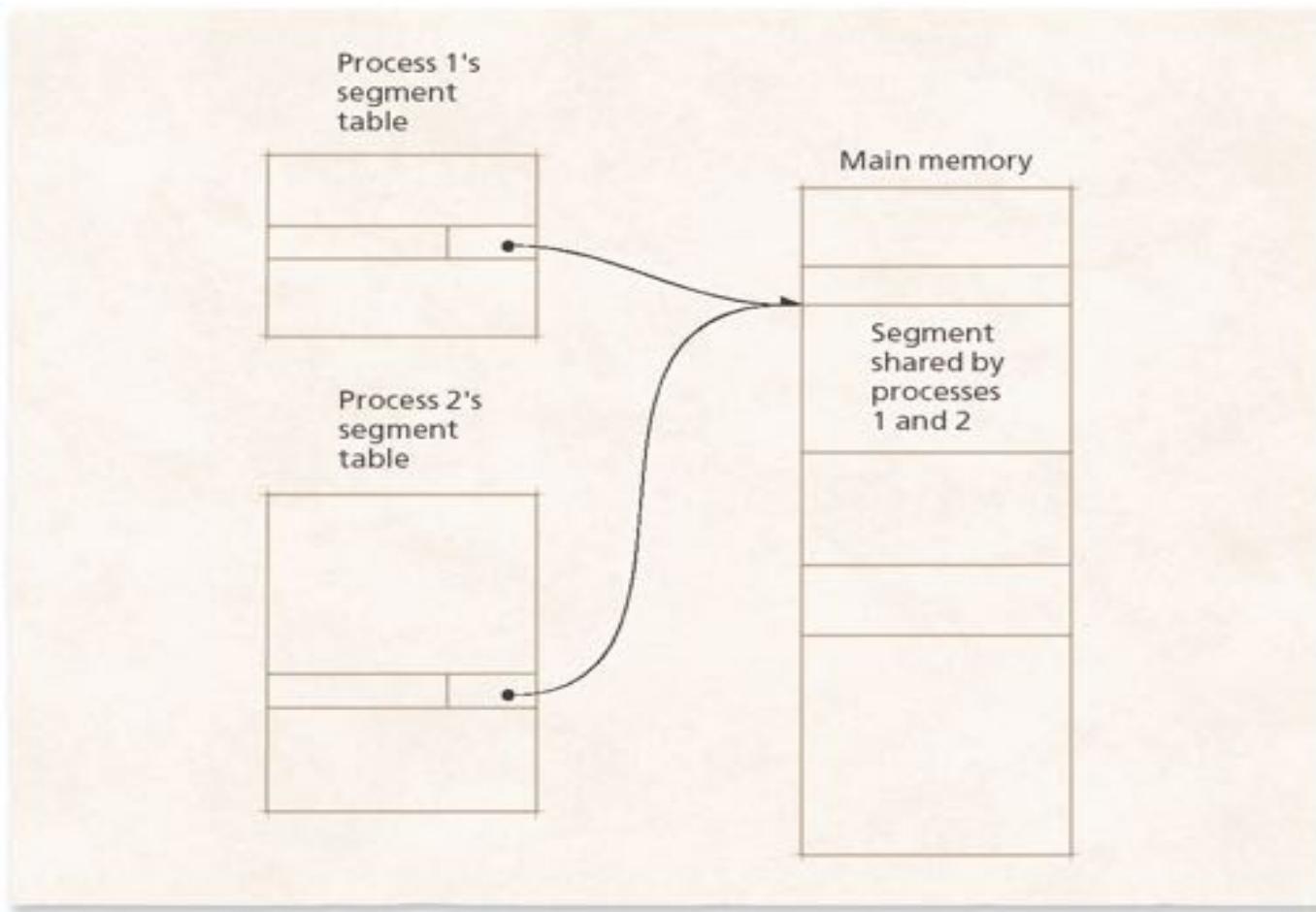
$r = 0$ if segment is not in main memory
 $r = 1$ if segment is in main memory

10.5.2 Sharing in a Segmentation System

- Sharing segments can incur less overhead than sharing in direct-mapped pure paging system
 - Potentially fewer map table entries need to be shared

10.5.2 Sharing in a Segmentation System

Figure 10.23 Sharing in a pure segmentation system.

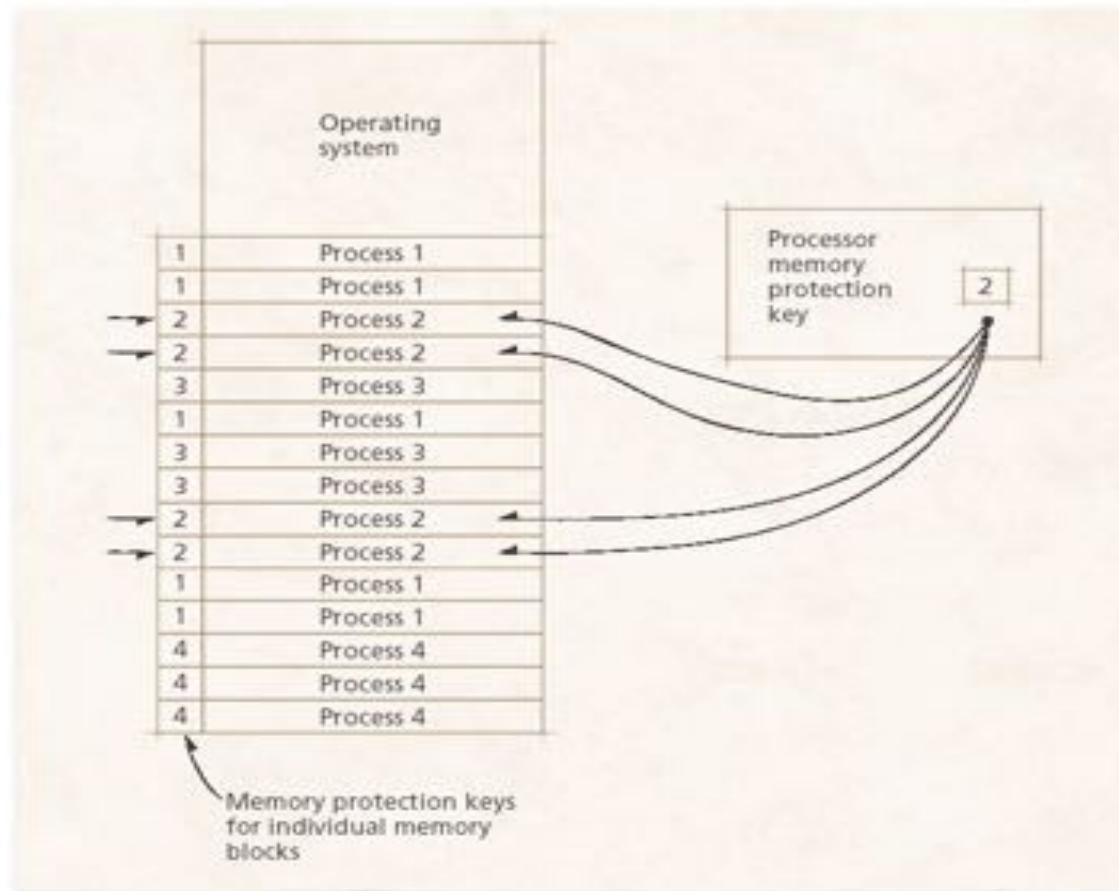


10.5.3 Protection and Access Control in Segmentation Systems

- One scheme for implementing memory protection in segmentation systems is memory protection keys
- Protection key
 - Associated with process
- If protection key for the processor and the requested block are the same, the process can access the segment

10.5.3 Protection and Access Control in Segmentation Systems

Figure 10.24 Memory protection with keys in noncontiguous memory allocation multiprogramming systems.



10.5.3 Protection and Access Control in Segmentation Systems

- A more common scheme is to use protection bits that specify whether a process can read, write, execute code or append to a segment

10.5.3 Protection and Access Control in Segmentation Systems

Figure 10.25 Access control types.

Type of access	Abbreviation	Description
Read	R	This segment may be read.
Write	W	This segment may be modified.
Execute	E	This segment may be executed.
Append	A	This segment may have information added to its end.

10.5.3 Protection and Access Control in Segmentation Systems

Figure 10.26 Combining read, write and execute access to yield useful access control modes.

Mode	Read	Write	Execute	Description	Application
Mode 0	No	No	No	No access permitted	Security.
Mode 1	No	No	Yes	Execute only	A segment made available to processes that cannot modify it or copy it, but that can run it.
Mode 2	No	Yes	No	Write only	These possibilities are not useful, because granting write access without read access is impractical.
Mode 3	No	Yes	Yes	Write/execute but cannot be read	
Mode 4	Yes	No	No	Read only	Information retrieval.
Mode 5	Yes	No	Yes	Read/execute	A program can be copied or executed but cannot be modified.
Mode 6	Yes	Yes	No	Read/write but no execution	Protects data from an erroneous attempt to execute it.
Mode 7	Yes	Yes	Yes	Unrestricted access	This access is granted to trusted users.

10.5.3 Protection and Access Control in Segmentation Systems

- Protection bits are added to the segment map table entry and checked when a process references an address
 - If the segment is not in memory, a segment-missing fault is generated
 - If $d > l$, a segment-overflow exception is generated
 - If the operation (e.g., read, write, execute, append) is not allowed, a segment-protection exception is generated

10.5.3 Protection and Access Control in Segmentation Systems

Figure 10.27 Segment map table entry with protection bits.

Segment resident bit	Secondary storage address (if not in main memory)	Segment length	Protection bits				Base address of segment (if segment is in memory)
<i>r</i>	<i>a</i>	<i>l</i>	<i>R</i>	<i>W</i>	<i>E</i>	<i>A</i>	<i>s'</i>

Protection bits: (1=yes, 0=no)

R—Read access
W—Write access
E—Execute access
A—Append access

10.6 Segmentation/Paging Systems

- Segments occupy one or more pages
- All pages of segment need not be in main memory at once
- Pages contiguous in virtual memory need not be contiguous in main memory
- Virtual memory address implemented as ordered triple $v = (s, p, d)$
 - s is segment number
 - p is page number within segment
 - d is displacement within page at which desired item located

10.6.1 Dynamic Address Translation in a Segmentation/Paging System

Figure 10.28 Virtual address format in a segmentation/paging system.

Segment number s	Page number p	Displacement d	Virtual address $v = (s, p, d)$
-----------------------	--------------------	---------------------	------------------------------------

10.6 Segmentation/Paging Systems

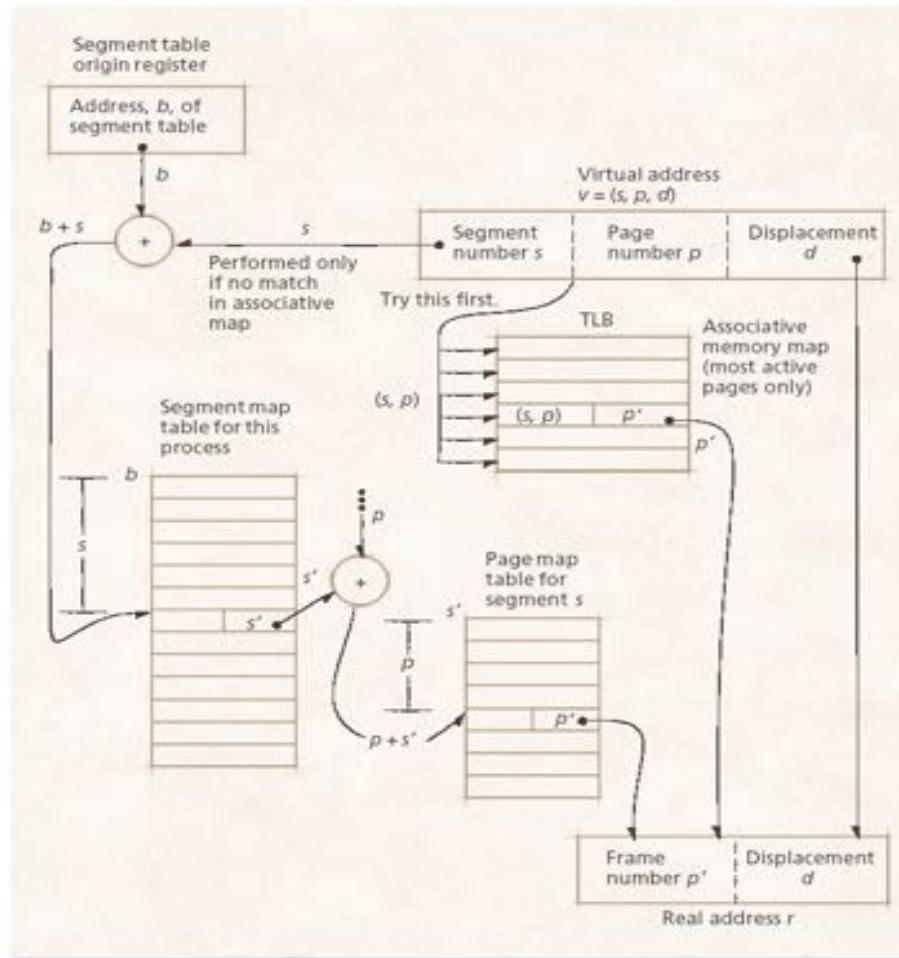
- Process references virtual memory address

$$v = (s, p, d)$$

- DAT adds the process's segment map table base address, b , to referenced segment number, s
- $b + s$ forms the main memory address of the segment map table entry for segment s
- Segment map table entry stores the base address of the page table, s'
- Referenced page number, p , is added to s' to locate the PTE for page p , which stores page frame number p'
- System concatenates p' with displacement, d , to form real address, r

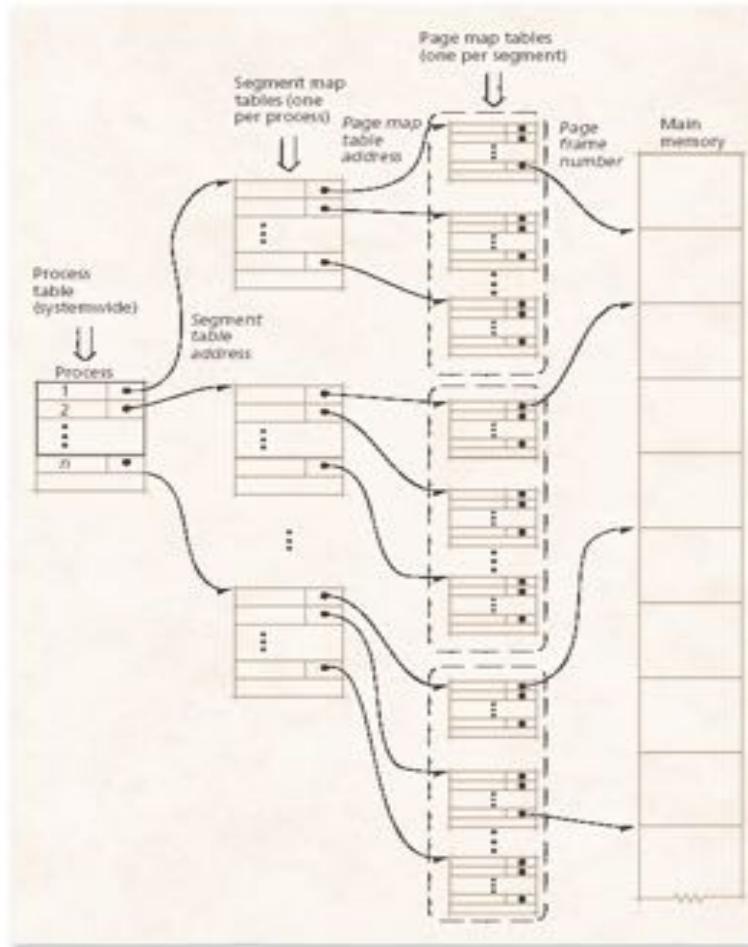
10.6.1 Dynamic Address Translation in a Segmentation/Paging System

Figure 10.29 Virtual address translation with combined associative/direct mapping in a segmentation/paging system.



10.6.1 Dynamic Address Translation in a Segmentation/Paging System

Figure 10.30 Table structure for a segmentation/paging system.

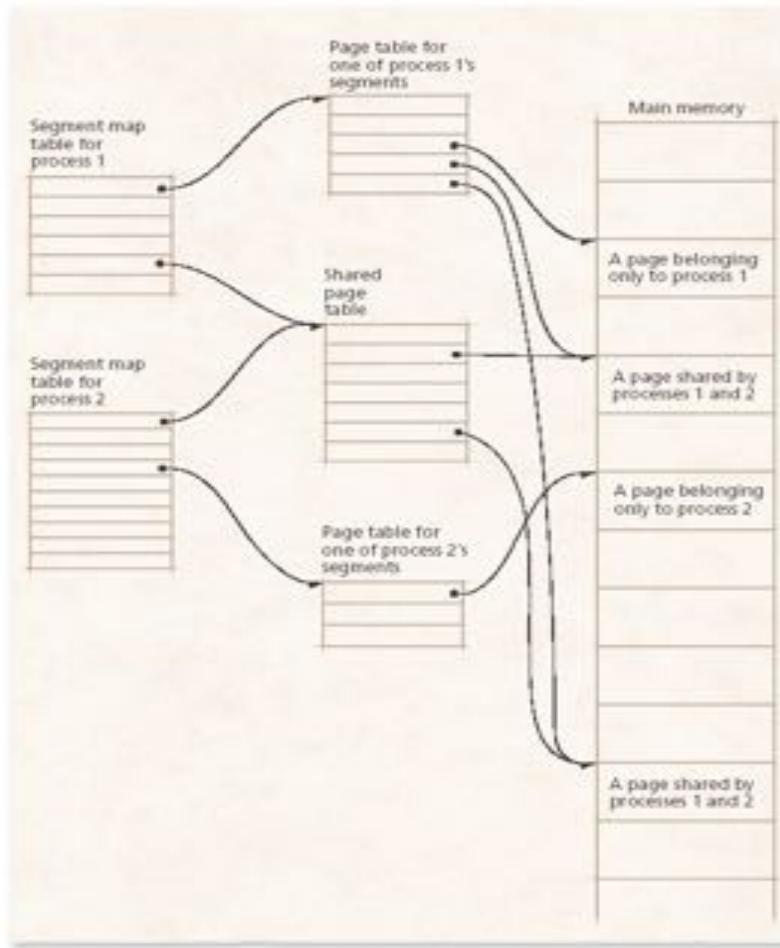


10.6.2 Sharing and Protection in a Segmentation/Paging System

- Segmentation/paging systems
 - Two processes share memory when each process has a segment map table entry that points to the same page table
- Sharing requires careful management by the operating system

10.7 Case Study: IA-32 Intel Architecture Virtual Memory

Figure 10.31 Two processes sharing a segment in a segmentation/paging system.



10.7 Case Study: IA-32 Intel Architecture Virtual Memory

- IA-32 Intel architecture supports either pure segmentation or segmentation/paging virtual memory
- Logical address space
 - Set of addresses contained in each segment
- Segments
 - Placed in any available location in system's linear address space
- Segment address translation
 - Performed by direct mapping that uses high-speed processor registers to store segment map table origin registers in global descriptor table register or in local descriptor table register
- Paging
 - Supports multilevel page tables and multiple page sizes

Chapter 11 – Virtual Memory Management

Outline

- 11.1 Introduction
- 11.2 Locality
- 11.3 Demand Paging
- 11.4 Anticipatory Paging
- 11.5 Page Replacement
- 11.6 Page Replacement Strategies
 - 11.6.1 Random Page Replacement
 - 11.6.2 First-In-First-Out (FIFO) Page Replacement
 - 11.6.3 FIFO Anomaly
 - 11.6.4 Least-Recently-Used (LRU) Page Replacement
 - 11.6.5 Least-Frequently-Used (LFU) Page Replacement
 - 11.6.6 Not-Used-Recently (NUR) Page Replacement
 - 11.6.7 Modifications to FIFO: Second-Chance and Clock Page Replacement
 - 11.6.8 Far Page Replacement
- 11.7 Working Set Model
- 11.8 Page-Fault-Frequency (PFF) Page Replacement



Chapter 11 – Virtual Memory Management

Outline (continued)

- 11.9 Page Release
- 11.10 Page Size
- 11.11 Program Behavior under Paging
- 11.12 Global vs. Local Page Replacement
- 11.13 Case Study: Linux Page Replacement

Objectives

- After reading this chapter, you should understand:
 - the benefits and drawbacks of demand and anticipatory paging.
 - the challenges of page replacement.
 - several popular page-replacement strategies and how they compare to optimal page replacement.
 - the impact of page size on virtual memory performance.
 - program behavior under paging.

11.1 Introduction

- Replacement strategy
 - Technique a system employs to select pages for replacement when memory is full
 - Determines where in main memory to place an incoming page or segment
- Fetch strategy
 - Determines when pages or segments should be loaded into main memory
 - Anticipatory fetch strategies
 - Use heuristics to predict which pages a process will soon reference and load those pages or segments

11.2 Locality

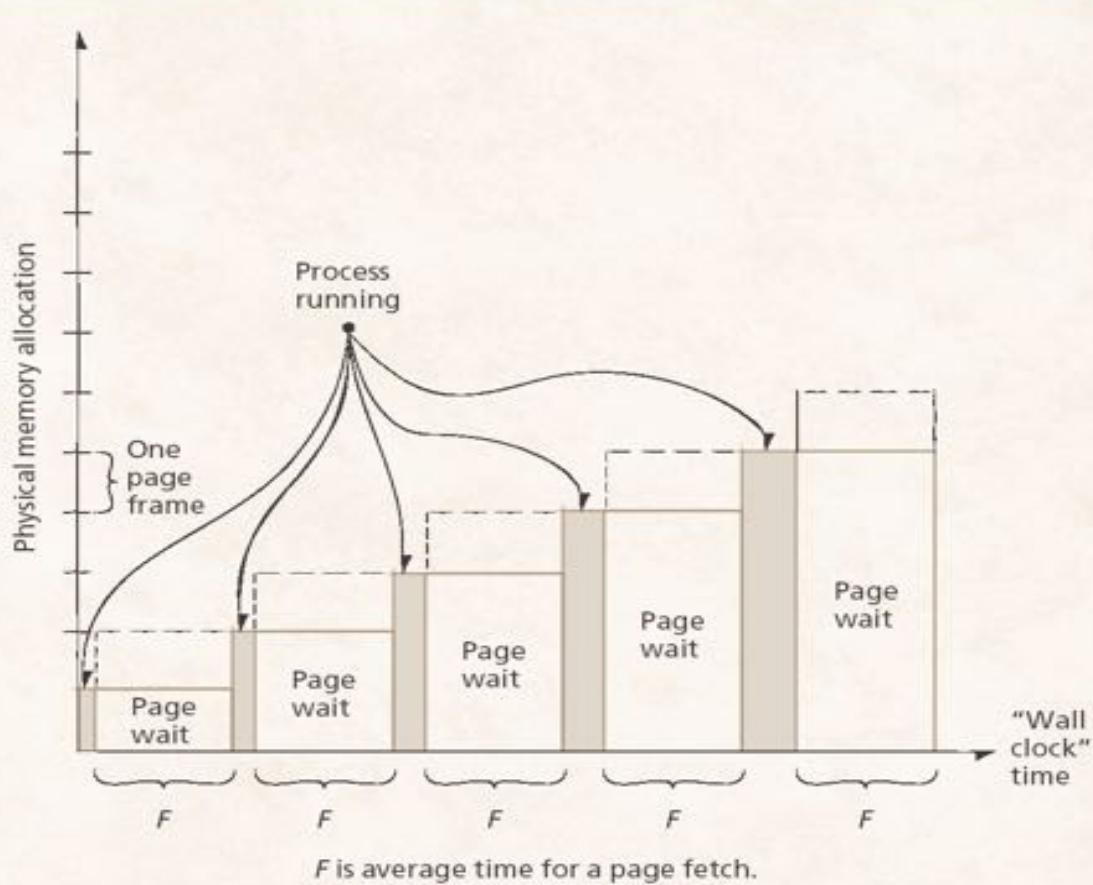
- Process tends to reference memory in highly localized patterns
 - In paging systems, processes tend to favor certain subsets of their pages, and these pages tend to be adjacent to one another in process's virtual address space

11.3 Demand Paging

- Demand paging
 - When a process first executes, the system loads into main memory the page that contains its first instruction
 - After that, the system loads a page from secondary storage to main memory only when the process explicitly references that page
 - Requires a process to accumulate pages one at a time

11.3 Demand Paging

Figure 11.1 Space-time product under demand paging.



11.4 Anticipatory Paging

- Anticipatory paging
 - Operating system attempts to predict the pages a process will need and preloads these pages when memory space is available
 - Anticipatory paging strategies
 - Must be carefully designed so that overhead incurred by the strategy does not reduce system performance

11.5 Page Replacement

- When a process generates a page fault, the memory manager must locate referenced page in secondary storage, load it into page frame in main memory and update corresponding page table entry
- Modified (dirty) bit
 - Set to 1 if page has been modified; 0 otherwise
 - Help systems quickly determine which pages have been modified
- Optimal page replacement strategy (OPT or MIN)
 - Obtains optimal performance, replaces the page that will not be referenced again until furthest into the future

10.6 Page-Replacement Strategies

- A page-replacement strategy is characterized by
 - Heuristic it uses to select a page for replacement
 - The overhead it incurs

10.6.1 Random Page Replacement

- Random page replacement
 - Low-overhead page-replacement strategy that does not discriminate against particular processes
 - Each page in main memory has an equal likelihood of being selected for replacement
 - Could easily select as the next page to replace the page that will be referenced next

10.6.2 First-In-First-Out (FIFO) Page Replacement

- FIFO page replacement
 - Replace page that has been in the system the longest
 - Likely to replace heavily used pages
 - Can be implemented with relatively low overhead
 - Impractical for most systems

11.6.2 First-In-First-Out (FIFO) Page Replacement

Figure 11.2 First-in-first-out (FIFO) page replacement.

Page reference	Result	FIFO page replacement with three pages available		
A	Fault	A	-	-
B	Fault	B	A	-
C	Fault	C	B	A
A	No fault	C	B	A
D	Fault	D	C	B
A	Fault	A	D	C
		:	:	:

A is replaced
B is replaced

11.6.3 FIFO Anomaly

- Belady's (or FIFO) Anomaly
 - Certain page reference patterns actually cause more page faults when number of page frames allocated to a process is increased

11.6.3 FIFO Anomaly

Figure 11.3 FIFO anomaly—page faults can increase with page frame allocation.

Page reference	Result	FIFO page replacement with three pages available			FIFO page replacement with four pages available			
		A	-	-	Fault	A	-	-
A	Fault				Fault	A	-	-
B	Fault	B	A	-	Fault	B	A	-
C	Fault	C	B	A	Fault	C	B	A
D	Fault	D	C	B	Fault	D	C	B
A	Fault	A	D	C	No fault	D	C	B
B	Fault	B	A	D	No fault	D	C	A
E	Fault	E	B	A	Fault	E	D	C
A	No fault	E	B	A	Fault	A	E	D
B	No fault	E	B	A	Fault	B	A	E
C	Fault	C	E	B	Fault	C	B	A
D	Fault	D	C	E	Fault	D	C	B
E	No fault	D	C	E	Fault	E	D	C

Three "no faults"

Two "no faults"

11.6.4 Least-Recently-Used (LRU) Page Replacement

- LRU page replacement
 - Exploits temporal locality by replacing the page that has spent the longest time in memory without being referenced
 - Can provide better performance than FIFO
 - Increased system overhead
 - LRU can perform poorly if the least-recently used page is the next page to be referenced by a program that is iterating inside a loop that references several pages

11.6.4 Least-Recently-Used (LRU) Page Replacement

Figure 11.4 Least-recently-used (LRU) page-replacement strategy.

Page reference	Result	LRU page replacement with three pages available		
A	Fault	A	-	-
B	Fault	B	A	-
C	Fault	C	B	A
B	No fault	B	C	A
B	No fault	B	C	A
A	No fault	A	B	C
D	Fault	D	A	B
A	No fault	A	D	B
B	No fault	B	A	D
F	Fault	F	B	A
B	No fault	B	F	A

11.6.5 Least-Frequently-Used (LFU) Page Replacement

- LFU page replacement
 - Replaces page that is least intensively referenced
 - Based on the heuristic that a page not referenced often is not likely to be referenced in the future
 - Could easily select wrong page for replacement
 - A page that was referenced heavily in the past may never be referenced again, but will stay in memory while newer, active pages are replaced

11.6.6 Not-Used-Recently (NUR) Page Replacement

- NUR page replacement
 - Approximates LRU with little overhead by using referenced bit and modified bit to determine which page has not been used recently and can be replaced quickly
 - Can be implemented on machines that lack hardware referenced bit and/or modified bit

11.6.6 Not-Used-Recently (NUR) Page Replacement

Figure 11.5 Page types under NUR.

<i>Group</i>	<i>Referenced</i>	<i>Modified</i>	<i>Description</i>
Group 1	0	0	Best choice to replace
Group 2	0	1	[Seems unrealistic]
Group 3	1	0	
Group 4	1	1	Worst choice to replace

11.6.7 Modification to FIFO: Second-Chance and Clock Page Replacement

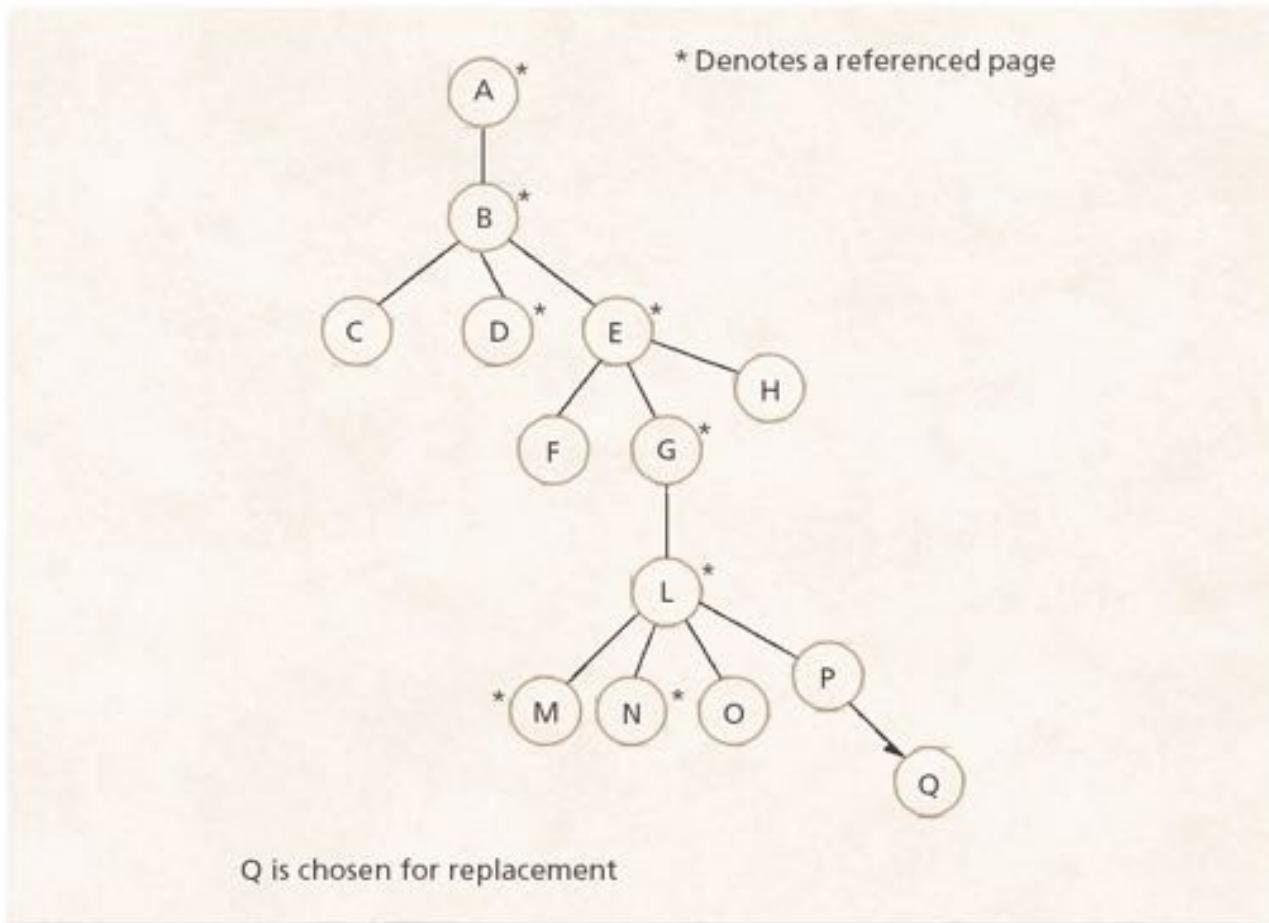
- **Second chance page replacement**
 - Examines referenced bit of the oldest page
 - If it's off
 - The strategy selects that page for replacement
 - If it's on
 - The strategy turns off the bit and moves the page to tail of FIFO queue
 - Ensures that active pages are the least likely to be replaced
- **Clock page replacement**
 - Similar to second chance, but arranges the pages in circular list instead of linear list

11.6.8 Far Page Replacement

- Far page replacement
 - Creates an access graph that characterizes a process's reference patterns
 - Replace the unreferenced page that is furthest away from any referenced page in the access graph
 - Performs at near-optimal levels
 - Has not been implemented in real systems
 - Access graph is complex to search and manage without hardware support

11.6.8 For Page Replacement

Figure 11.6 Far page-replacement-strategy access graph.

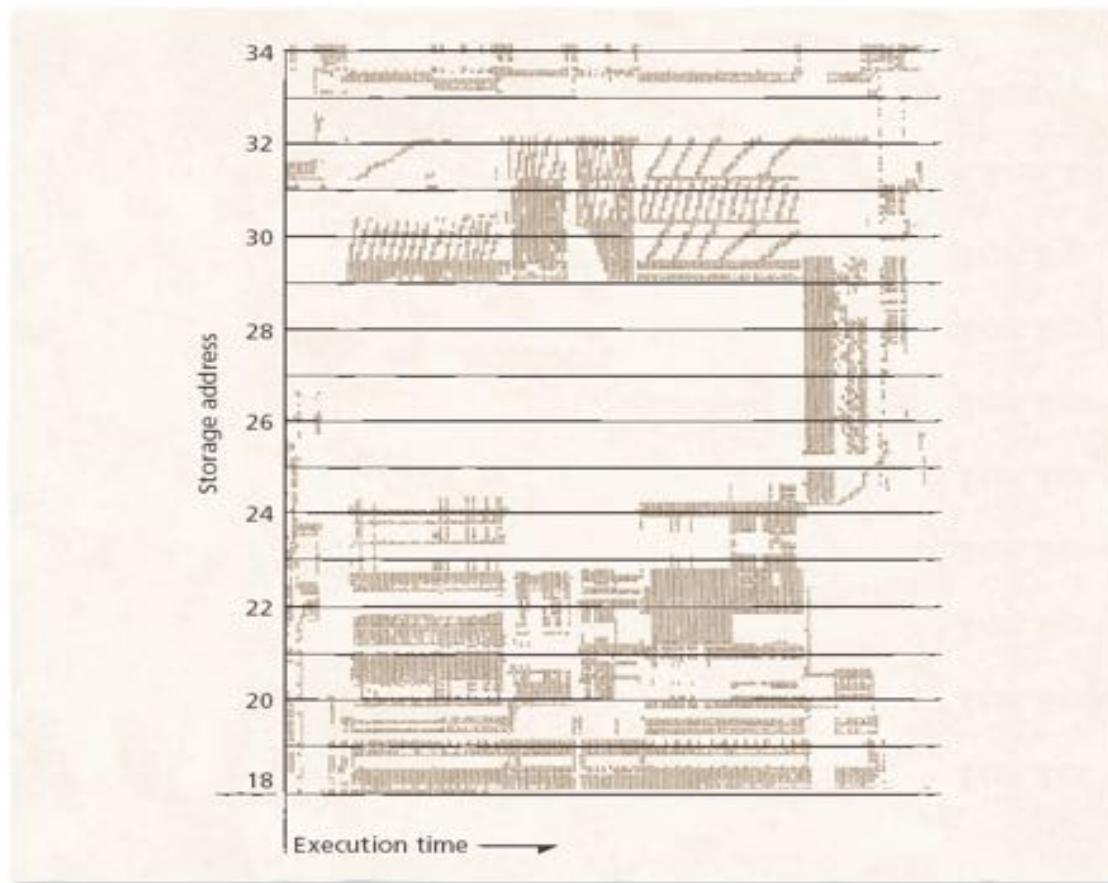


11.7 Working Set Model

- For a program to run efficiently
 - The system must maintain that program's favored subset of pages in main memory
- Otherwise
 - The system might experience excessive paging activity causing low processor utilization called thrashing as the program repeatedly requests pages from secondary storage

11.7 Working Set Model

Figure 11.7 Storage reference pattern exhibiting locality. (Reprinted by permission from IBM Systems Journal. © 1971 by International Business Machines Corporation.)

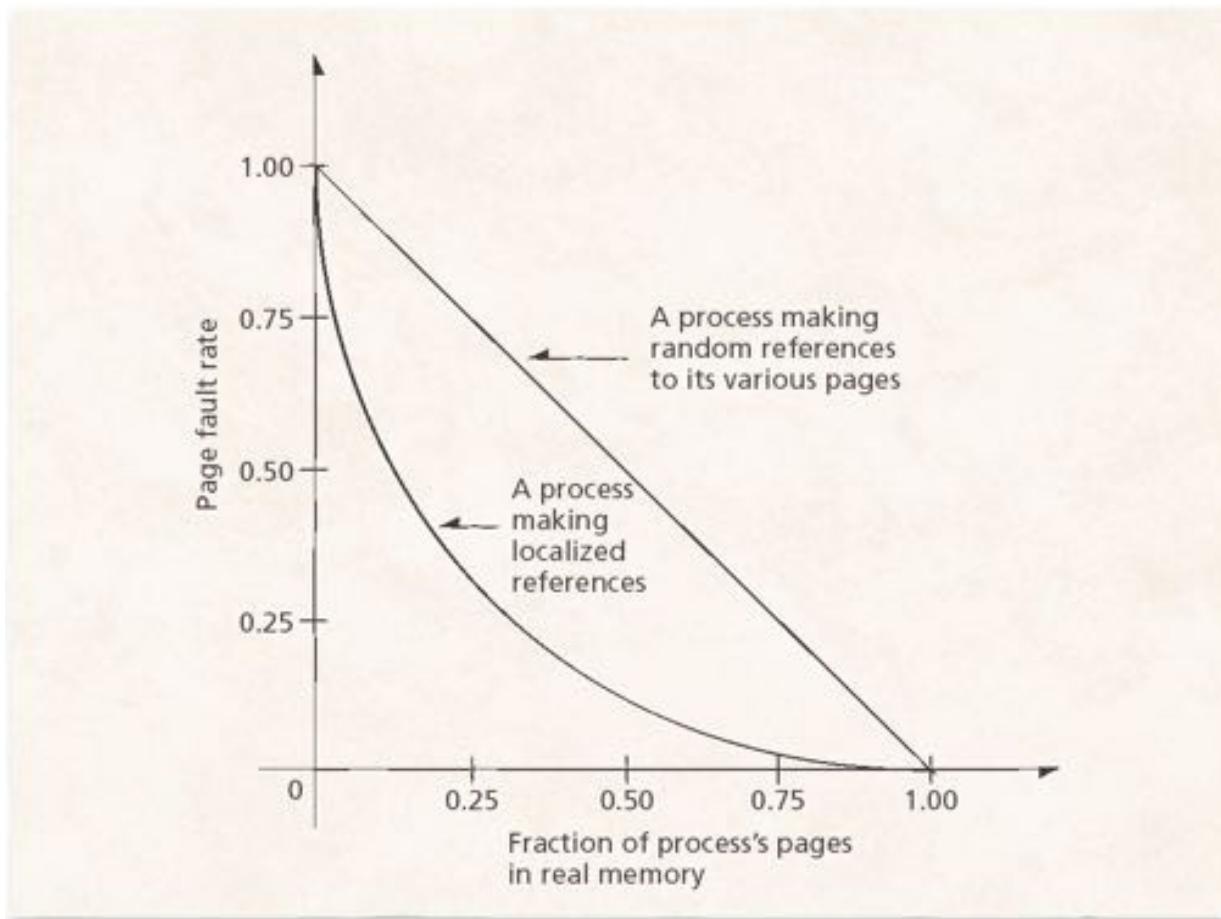


11.7 Working Set Model

- Because processes exhibit locality, increasing the number of page frames allocated to a process has little or no effect on its page fault rate at a certain threshold

11.7 Working Set Model

Figure 11.8 Dependence of page fault rate on amount of memory for a process's pages.

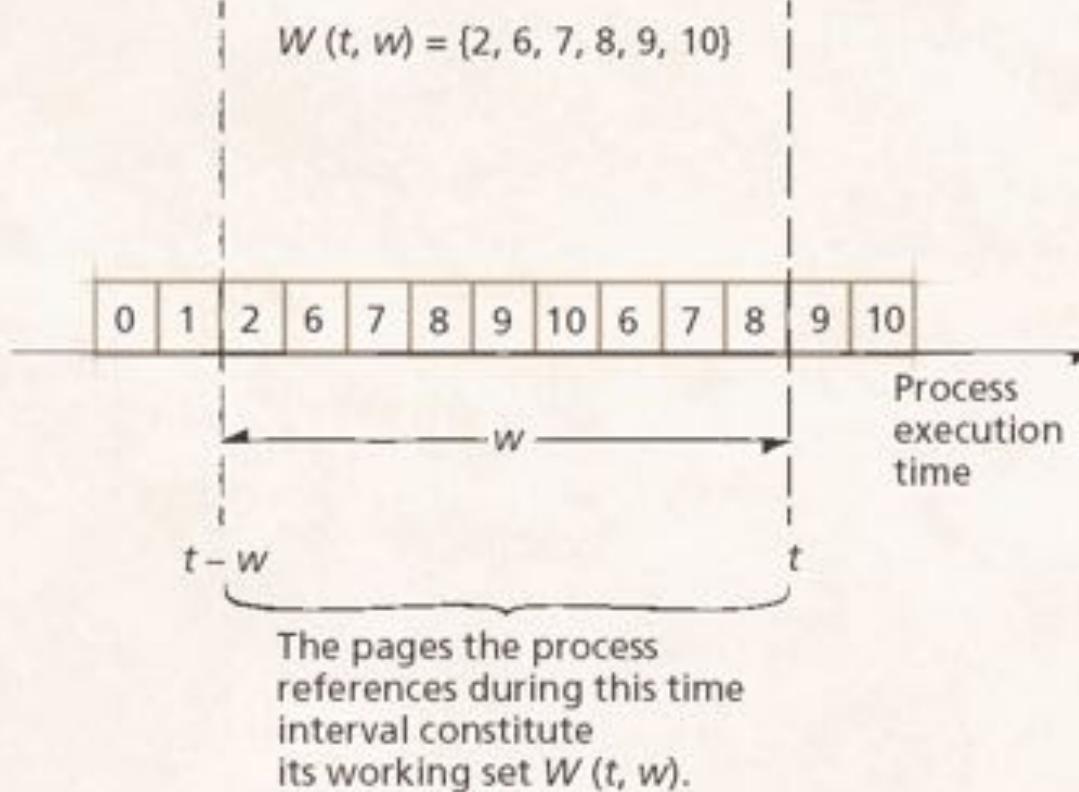


11.7 Working Set Model

- The process's working set of pages, $W(t, w)$, is the set of pages referenced by the process during the process-time interval $t - w$ to t .

11.7 Working Set Model

Figure 11.9 Definition of a process's working set of pages.

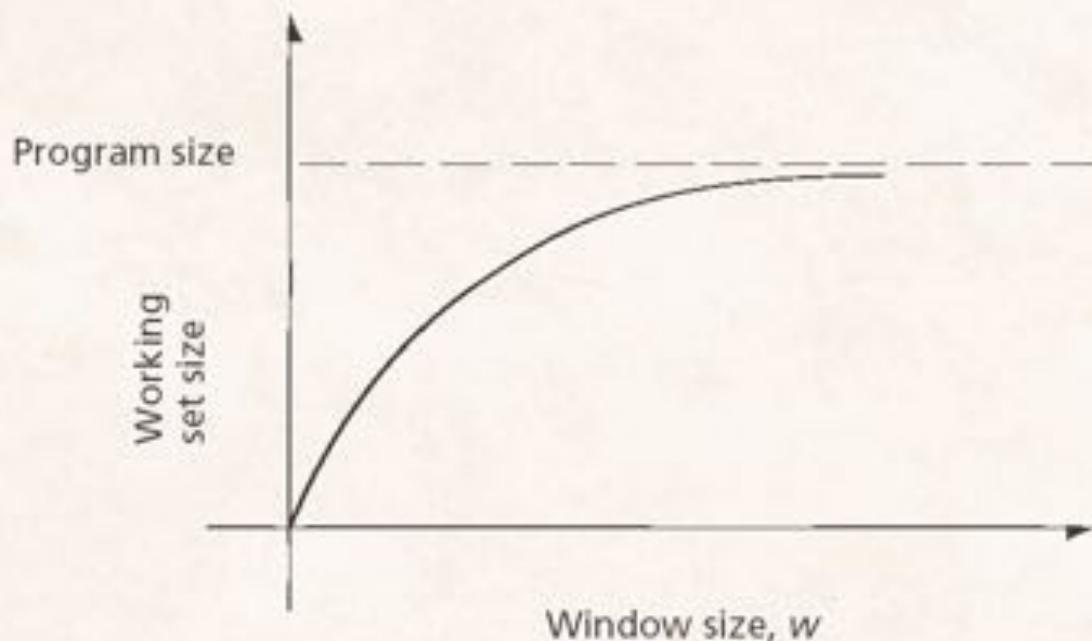


11.7 Working Set Model

- The size of the process's working set increases asymptotically to the process's program size as its working set window increases

11.7 Working Set Model

Figure 11.10 Working set size as a function of window size.

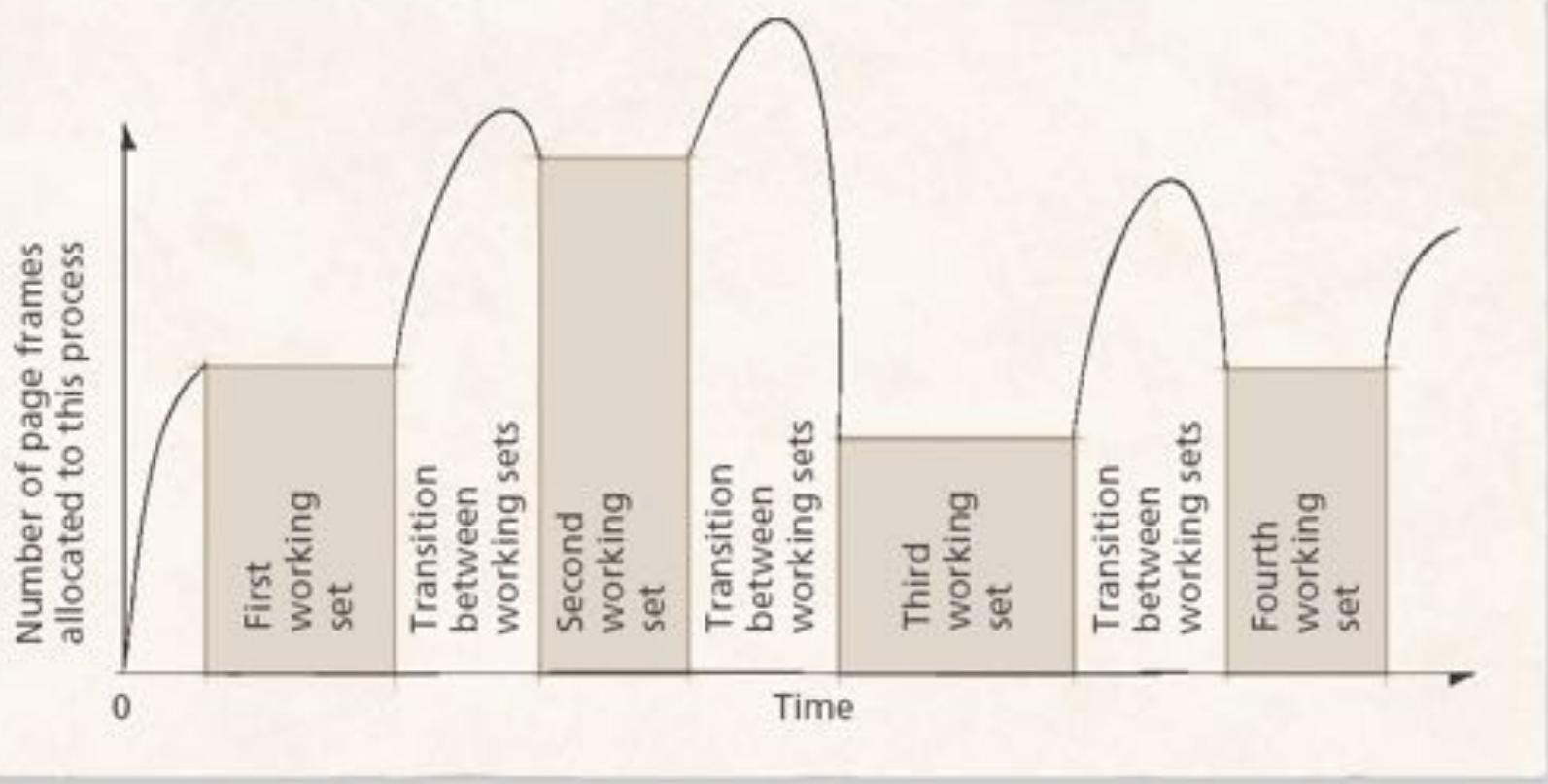


11.7 Working Set Model

- As a process transitions between working sets, the system temporarily maintains in memory pages that are no longer in the process's current working set
 - Goal of working set memory management is to reduce this misallocation

11.7 Working Set Model

Figure 11.11 Main memory allocation under working set memory management.



11.8 Page-Fault-Frequency (PFF) Page Replacement

- Adjusts a process's resident page set
 - Based on frequency at which the process is faulting
 - Based on time between page faults, called the process's interfault time
- Advantage of PFF over working set page replacement
 - Lower overhead
 - PFF adjusts resident page set only after each page fault
 - Working set mechanism must operate after each memory reference

11.9 Page Release

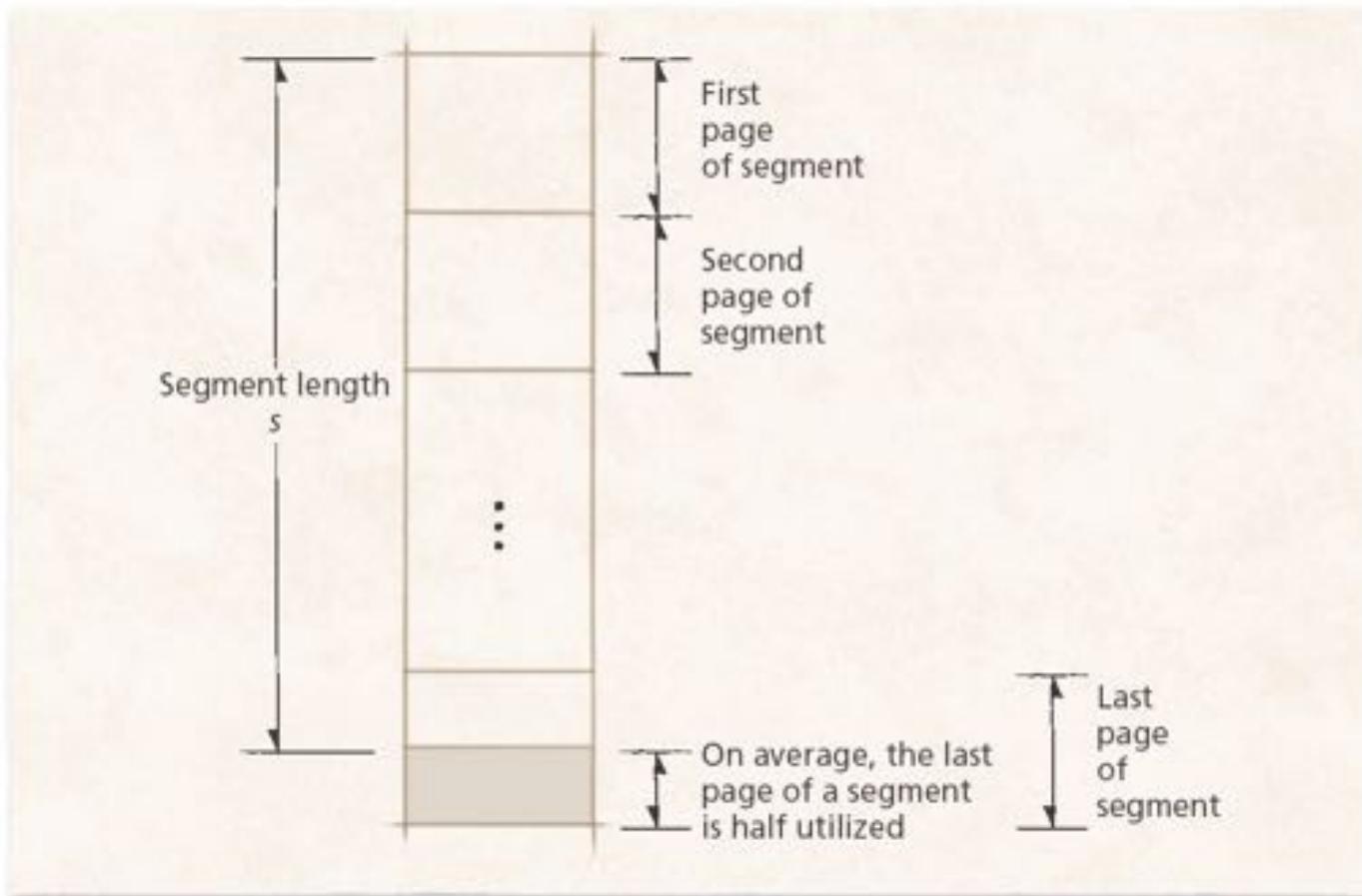
- Inactive pages can remain in main memory for a long time until the management strategy detects that the process no longer needs them
 - One way to solve the problem
 - Process issues a voluntary page release to free a page frame that it knows it no longer needs
 - Eliminate the delay period caused by letting process gradually pass the page from its working set
 - The real hope is in compiler and operating system support

11.10 Page Size

- Some systems improve performance and memory utilization by providing multiple page sizes
 - Small page sizes
 - Reduce internal fragmentation
 - Can reduce the amount of memory required to contain a process's working set
 - More memory available to other processes
 - Large page size
 - Reduce wasted memory from table fragmentation
 - Enable each TLB entry to map larger region of memory, improving performance
 - Reduce number of I/O operations the system performs to load a process's working set into memory
 - Multiple page size
 - Possibility of external fragmentation

11.10 Page Size

Figure 11.12 Internal fragmentation in a paged and segmented system.



11.10 Page Size

Figure 11.13 Page sizes in various processor architectures.

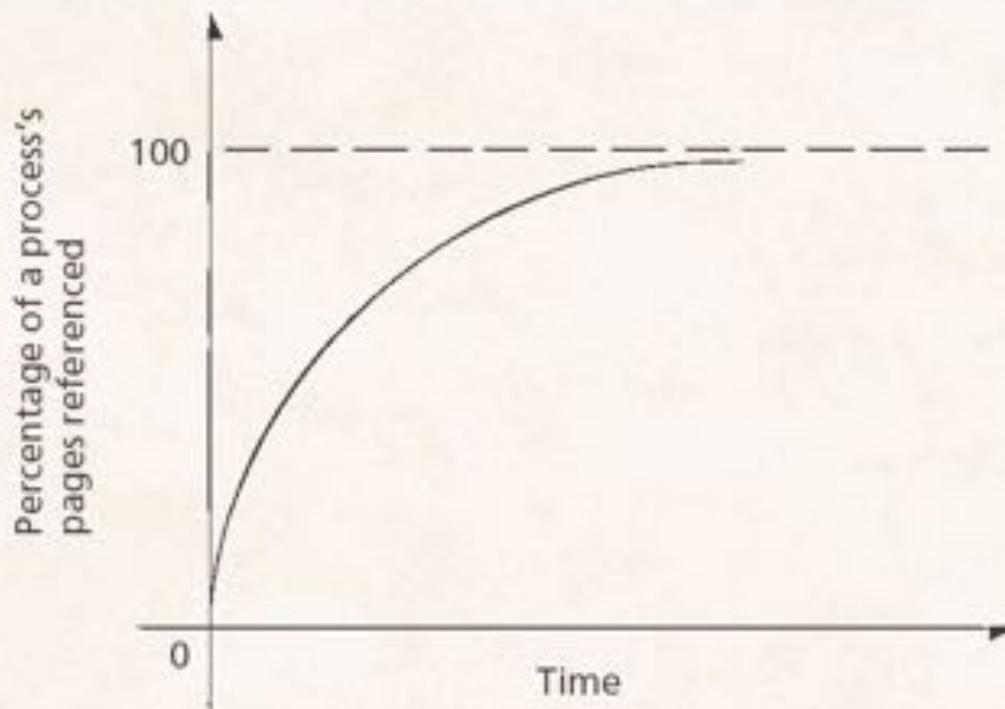
<i>Manufacturer</i>	<i>Model</i>	<i>Page Size</i>	<i>Real address size</i>
Honeywell	Multics	1KB	36 bits
IBM	370/168	4KB	32 bits
DEC	PDP-10 and PDP-20	512 bytes	36 bits
DEC	VAX 8800	512 bytes	32 bits
Intel	80386	4KB	32 bits
Intel / AMD	Pentium 4 / Athlon XP	4KB or 4MB	32- or 36 bits
Sun	UltraSparc II	8KB, 64KB, 512KB, 4MB	44 bits
AMD	Opteron / Athlon 64	4KB, 2MB and 4MB	32, 40, or 52 bits
Intel-HP	Itanium, Itanium 2	4KB, 8KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB, 256MB	Between 32 and 63 bits
IBM	PowerPC 970	4KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB	32 or 64 bits

11.11 Program Behavior under Paging

- Processes tend to reference a significant portion of their pages within a short time after execution begins
- They access most of their remaining pages at a slower rate

11.11 Program Behavior under Paging

Figure 11.14 Percentage of a process's pages referenced with time.

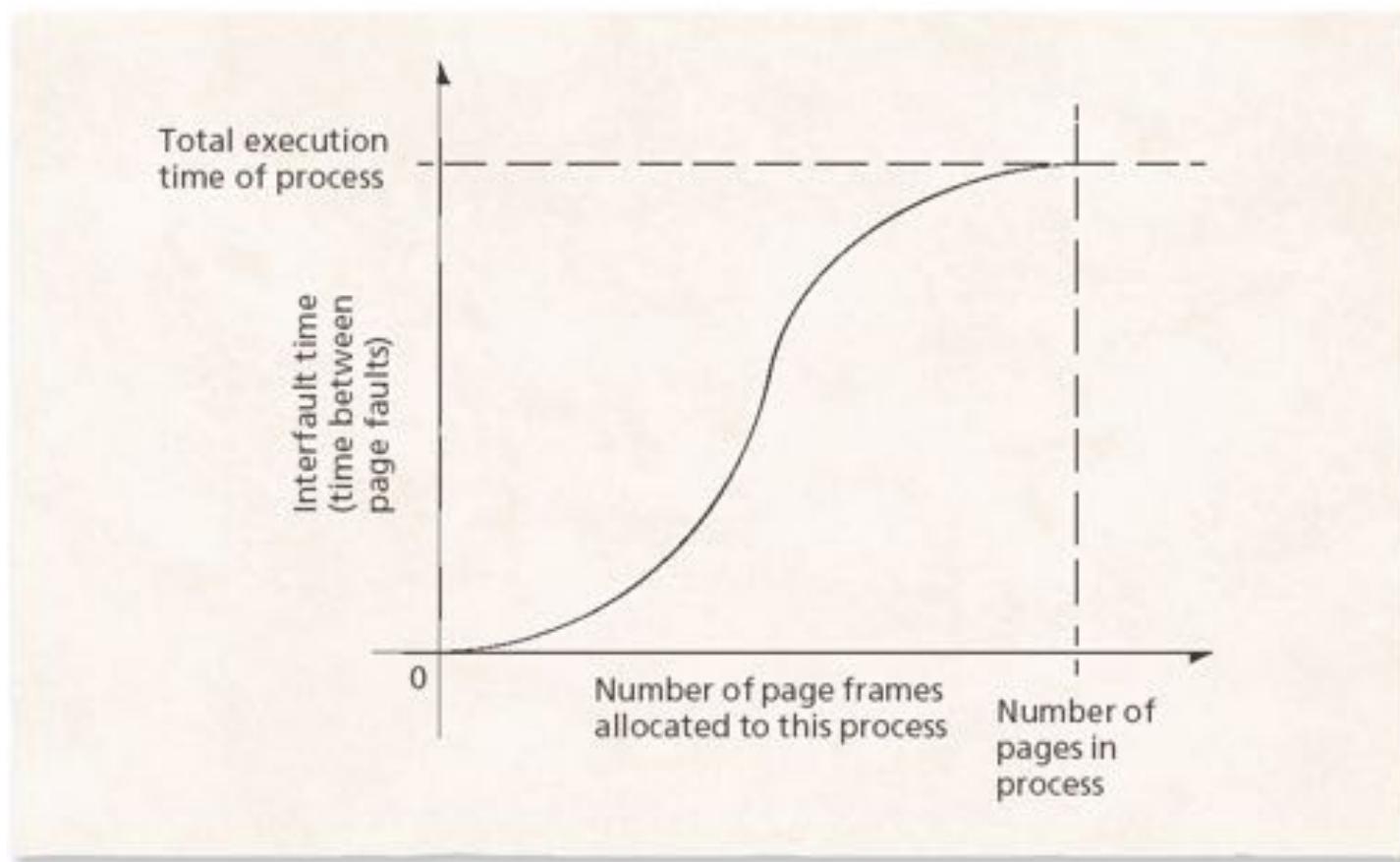


11.11 Program Behavior under Paging

- Average interfault time monotonically increases in general
 - The more page frames a process has, the longer the time between page faults

11.11 Program Behavior under Paging

Figure 11.15 Dependency of interfault time on the number of page frames allocated to a process.



11.12 Global vs. Local Page Replacement

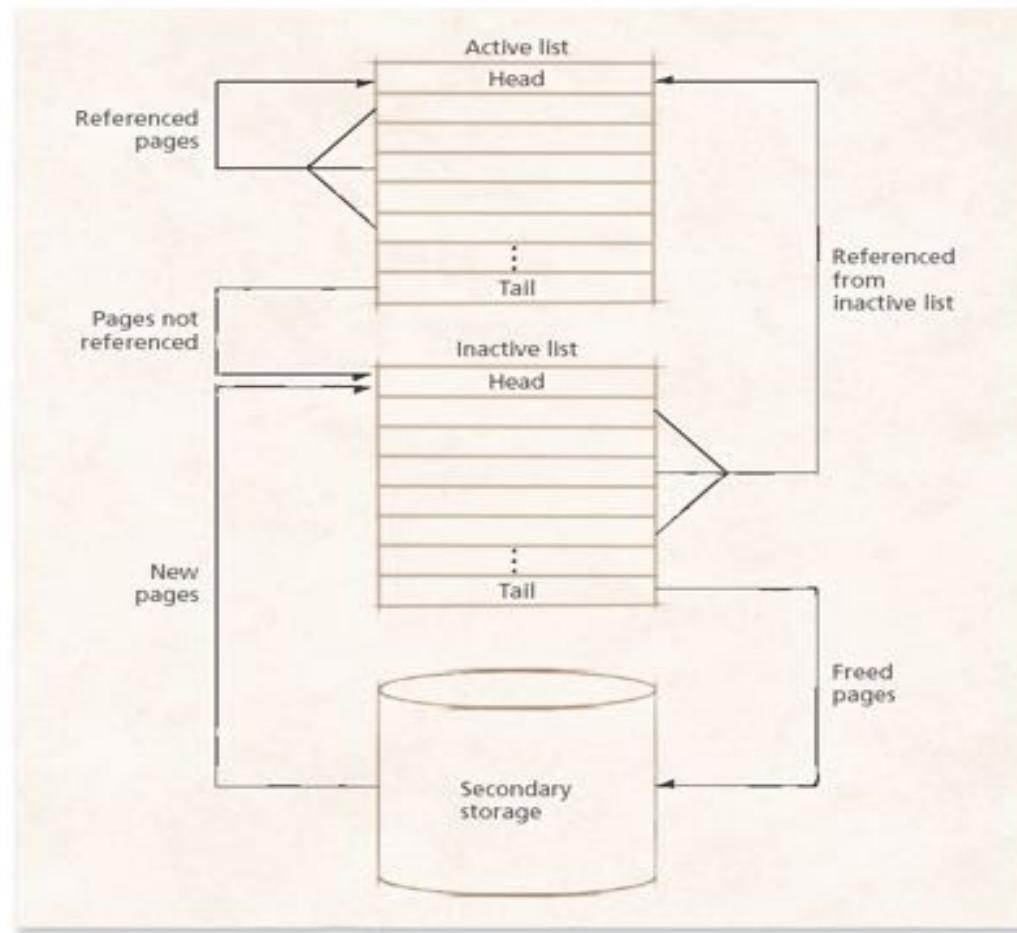
- Implementing a paged virtual memory system
 - Global page-replacement strategies: applied to all processes as a unit
 - Tend to ignore characteristics of individual process behavior
 - Global LRU (gLRU) page-replacement strategy
 - Replaces the least-recently-used page in entire system
 - SEQ (sequence) global page-replacement strategy
 - Uses LRU strategy to replace pages until sequence of page faults to contiguous pages is detected, at which point it uses most-recently-used (MRU) page-replacement strategy
 - Local page-replacement strategies: Consider each process individually
 - Enables system to adjust memory allocation according to relative importance of each process to improve performance

11.13 Case Study: Linux Page Replacement

- Linux uses a variation of the clock algorithm to approximate an LRU page-replacement strategy
- The memory manager uses two linked lists
 - Active list
 - Contains active pages
 - Most-recently used pages are near head of active list
 - Inactive list
 - Contains inactive pages
 - Least-recently used pages near tail of inactive list
 - Only pages in the inactive list are replaced

11.13 Case Study: Linux Page Replacement

Figure 11.16 Linux page replacement overview.



Chapter 12 – Disk Performance Optimization

Outline

- 12.1 Introduction
- 12.2 Evolution of Secondary Storage
- 12.3 Characteristics of Moving-Head Disk Storage
- 12.4 Why Disk Scheduling Is Necessary
- 12.5 Disk Scheduling Strategies
 - 12.5.1 First-Come-First-Served (FCFS) Disk Scheduling
 - 12.5.2 Shortest-Seek-Time-First (SSTF) Disk Scheduling
 - 12.5.3 SCAN Disk Scheduling
 - 12.5.4 C-SCAN Disk Scheduling
 - 12.5.5 FSCAN and N-Step SCAN Disk Scheduling
 - 12.5.6 LOOK and C-LOOK Disk Scheduling
- 12.6 Rotational Optimization
 - 12.6.1 SLTF Scheduling
 - 12.6.2 SPTF and SATF Scheduling



Chapter 12 – Disk Performance Optimization

Outline (cont.)

- 12.7 System Considerations
- 12.8 Caching and Buffering
- 12.9 Other Disk Performance Techniques
- 12.10 Redundant Arrays of Independent Disks (RAID)
 - 12.10.1 RAID Overview
 - 12.10.2 Level 0 (Striping)
 - 12.10.3 Level 1 (Mirroring)
 - 12.10.4 Level 2 (Bit-Level Hamming ECC Parity)
 - 12.10.5 Level 3 (Bit-Level XOR ECC Parity)
 - 12.10.6 Level 4 (Block-Level XOR ECC Parity)
 - 12.10.7 Level 5 (Block-Level Distributed XOR ECC Parity)

Objectives

- After reading this chapter, you should understand:
 - how disk input/output is accomplished.
 - the importance of optimizing disk performance.
 - seek optimization and rotational optimization.
 - various disk scheduling strategies.
 - caching and buffering.
 - other disk performance improvement techniques.
 - key schemes for implementing redundant arrays of independent disks (RAID).

12.10 Redundant Arrays of Independent Disks

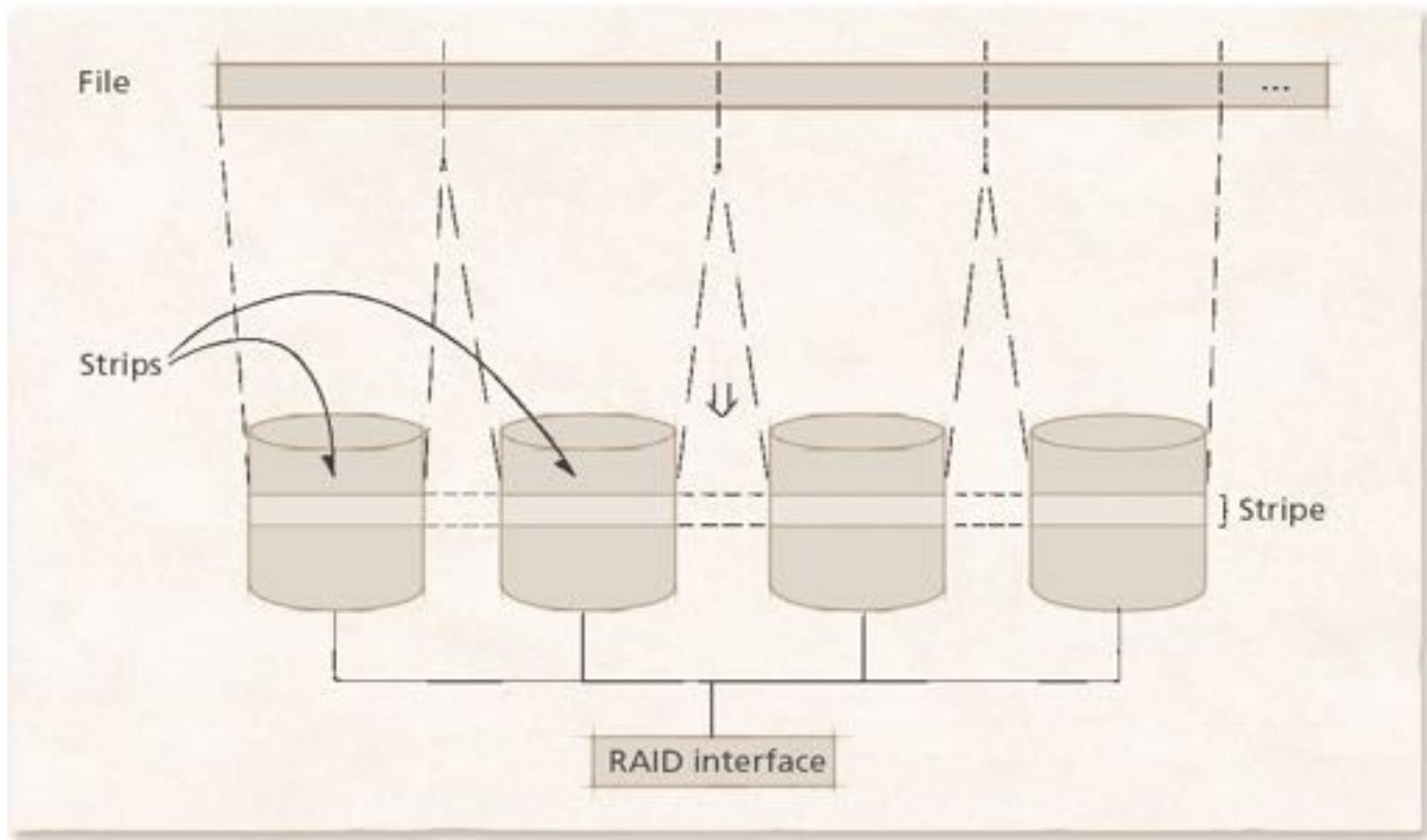
- Patterson et al. observed that memory and processor speeds tend to increase much faster than disk I/O speeds
- RAID developed to avoid the “pending I/O” crisis
 - Attempts to improve disk performance and/or reliability
 - Multiple disks in an array can be accessed simultaneously
 - Performs accesses in parallel to increase throughput
 - Additional drives can be used to improve data integrity

12.10.1 RAID Overview

- Common RAID characteristics
 - Data stored in strips
 - Spread across set of disks
 - Strips form stripes
 - Set of strips at same location on each disk
 - Fine-grained strip
 - Yields high transfer rates (many disks service request at once)
 - Array can only process one request at once
 - Coarse-grained strips
 - Might fit an entire file on one disk
 - Allow multiple requests to be filled at once

12.10.1 RAID Overview

Figure 12.16 Strips and stripe created from a single file in RAID systems.



12.10.1 RAID Overview

- Potential drawbacks
 - Larger number of disks decreases mean-time-to-failure (MTTF)
 - More disks provide more points of failure
 - Data stored in many RAID levels can be lost if more than one drive in array fails

12.10.1 RAID Overview

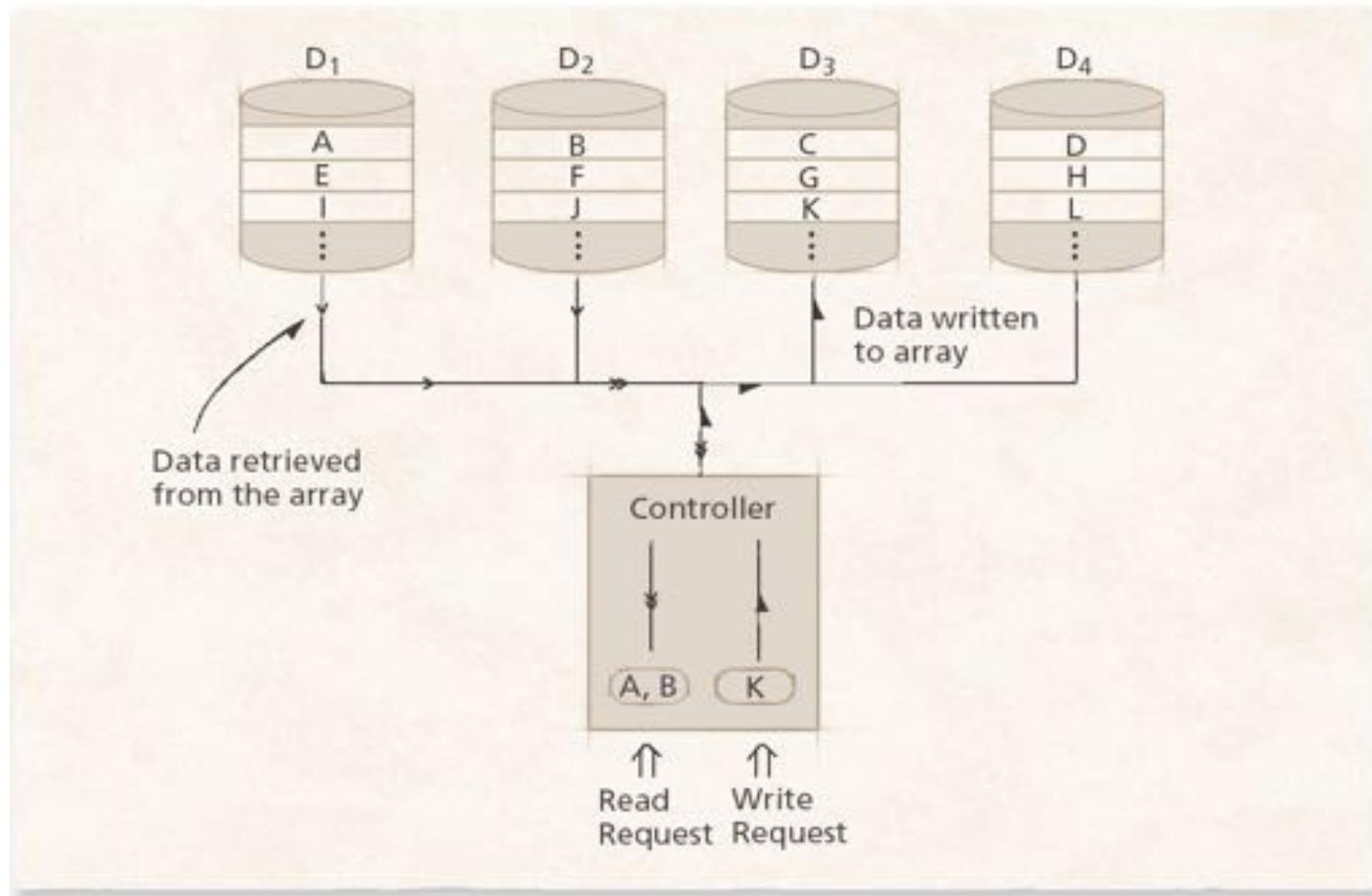
- Related technologies
 - Disk mirroring
 - One disk is simply a copy of another
 - Simple way to achieve redundancy and fault tolerance, but incurs significant storage overhead
 - RAID controller
 - Special-purpose hardware dedicated to RAID operations
 - Offloads most responsibility from operating system/processor
 - Can be expensive

12.10.2 Level 0 (Striping)

- **Level 0**
 - Simplest RAID implementation
 - Includes striping but no redundancy (not a “true” RAID level)
 - Highest-performing RAID level for a fixed number of disks
 - High risk of data loss
 - Multiple drives involved
 - Could lose all data in array with one drive failure
 - Appropriate where performance greatly outweighs reliability

12.10.2 Level 0 (Striping)

Figure 12.17 RAID level 0 (striping).

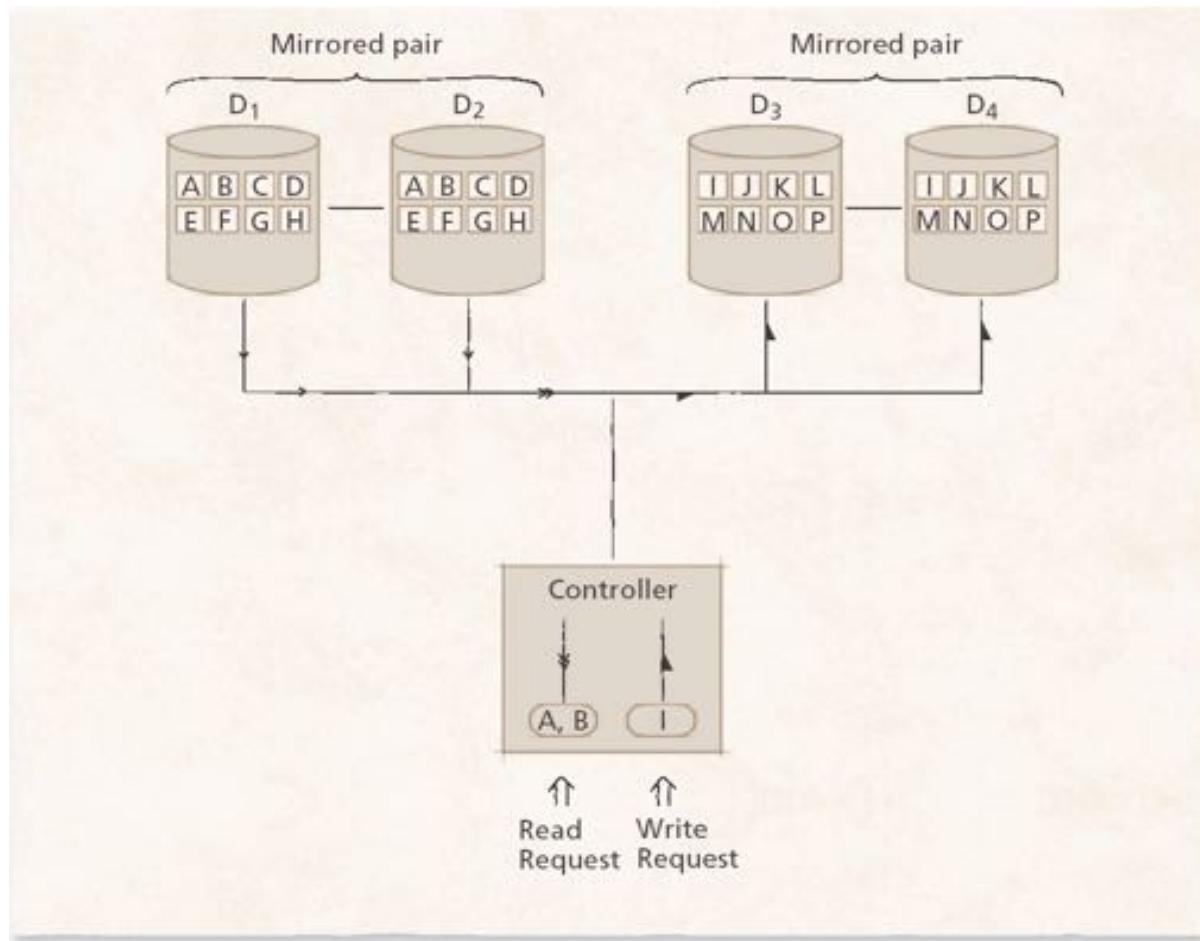


12.10.3 Level 1 (Mirroring)

- **Level 1**
 - Highest level of redundancy/fault tolerance for traditional RAID levels
 - Each drive has a mirrored copy in array
 - No striping at this level
 - Improves read performance over single disks because multiple disks can be read at once
 - Slower write performance because two disks must be accessed for each modified data item to maintain mirroring
 - High storage overhead
 - Only half array stores unique data
 - Most suitable where reliability is primary concern

12.10.3 Level 1 (Mirroring)

Figure 12.18 RAID level 1 (mirroring).

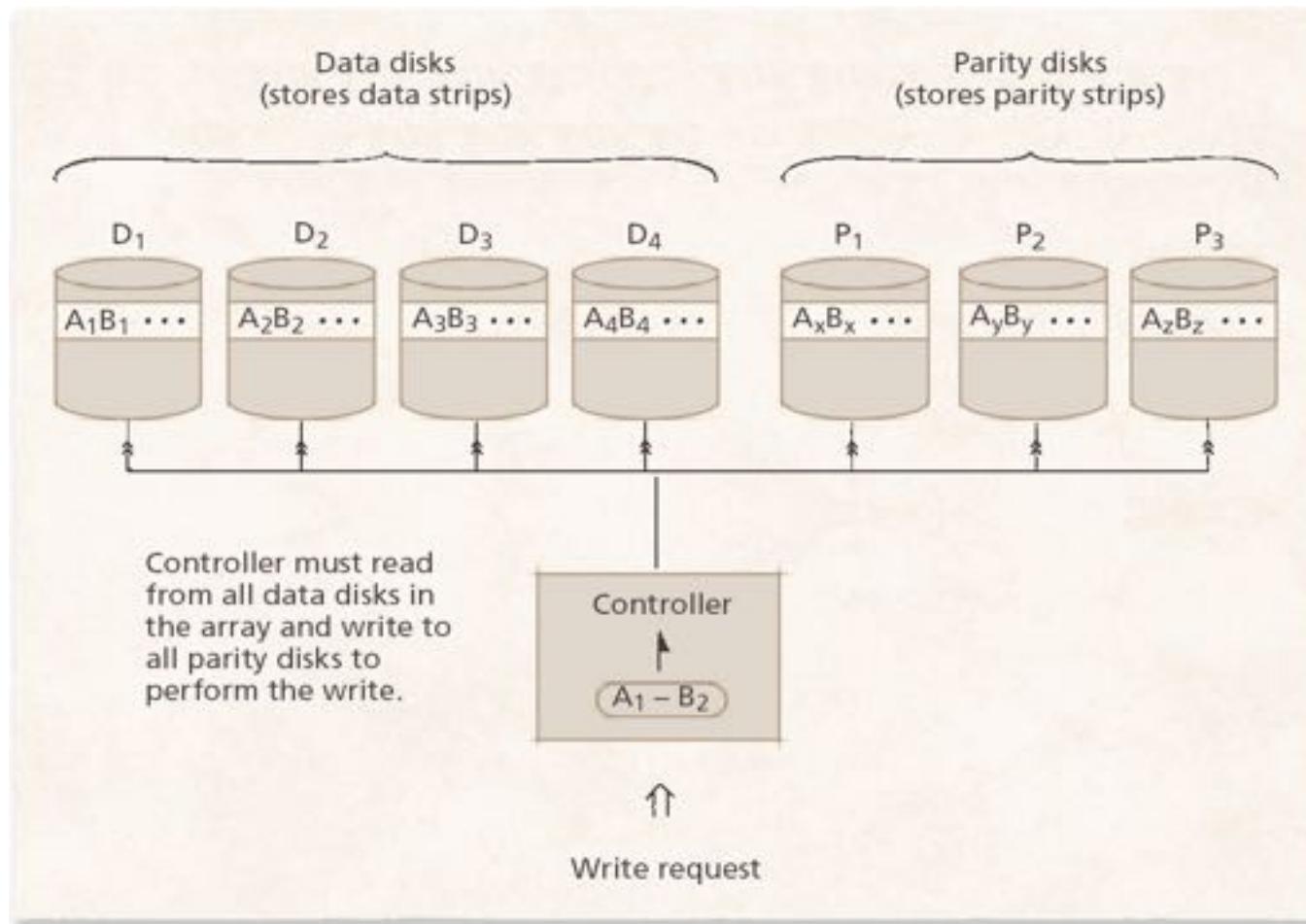


12.10.4 Level 2 (Bit-Level Hamming ECC Parity)

- Level 2
 - Implements redundancy and striping
 - Striped at bit level
 - Uses Hamming ECC to check data integrity
 - Parity bits store the evenness or oddness of a sum of bits
 - ECC data stored on separate drive
 - Significant overhead in storage (though less than level 1 arrays) and performance (due to calculating ECC data)
 - Not the most appropriate error checking method; ECC is performed internally by most hard disks
 - Rarely seen in modern systems

12.10.4 Level 2 (Bit-Level Hamming ECC Parity)

Figure 12.19 RAID level 2 (bit-level ECC parity).

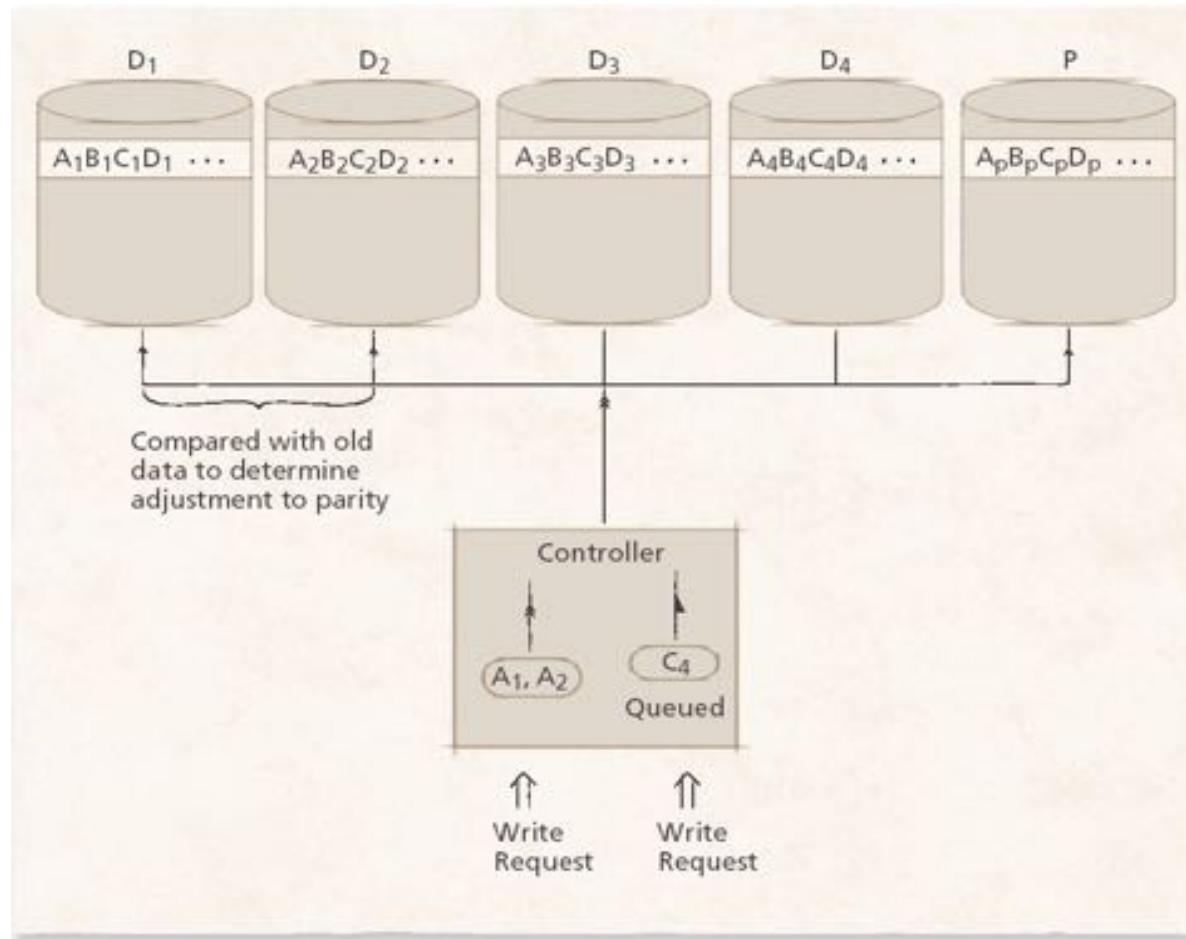


12.10.5 Level 3 (Bit-Level XOR ECC Parity)

- **Level 3**
 - Also stripes at the bit level
 - Uses XOR to calculate parity for ECC
 - Much simpler than Hamming ECC
 - Requires only one disk for parity information regardless of the size of the array
 - Cannot determine which bit contains error, but this information can be gathered easily by inspecting the array for a failed disk
 - High transfer rates, but only one request serviced at a time

12.10.5 Level 3 (Bit-Level XOR ECC Parity)

Figure 12.20 RAID level 3 (bit-level, single parity disk).

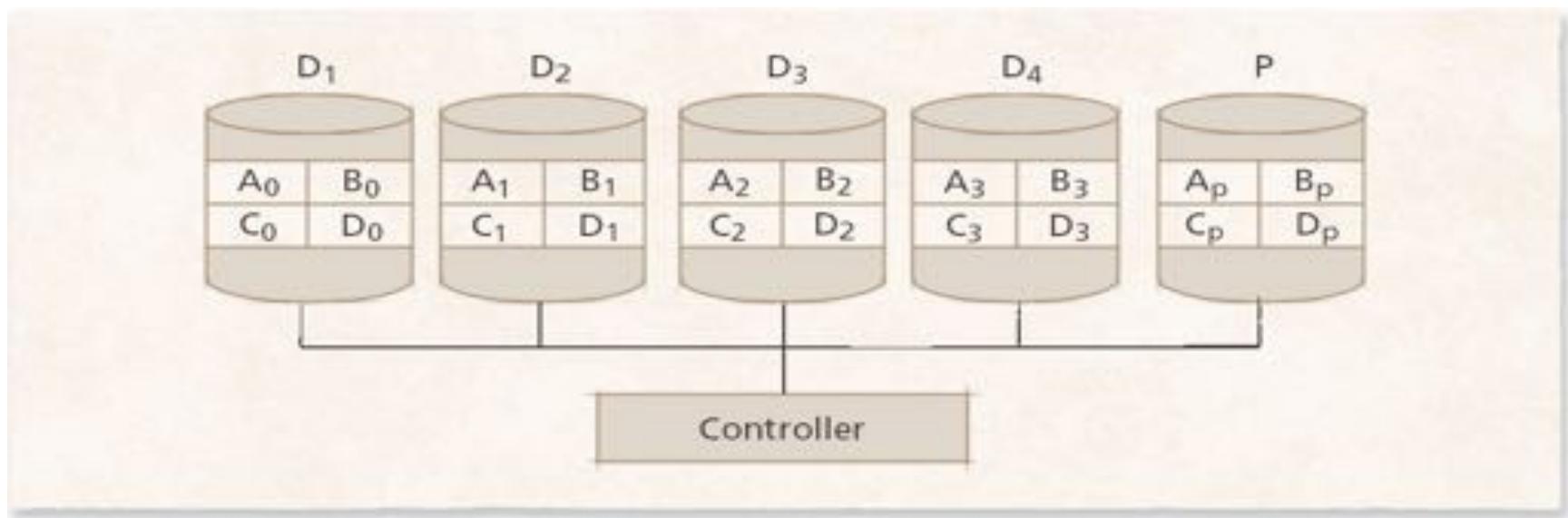


12.10.6 Level 4 (Block-Level XOR ECC Parity)

- **Level 4**
 - Similar to RAID level 3
 - Stores larger strips than other levels
 - Can service more requests simultaneously as files are more likely to be on one disk
 - Write requests must be performed one at a time
 - Restriction eliminated in level 5
 - Rarely implemented because level 5 is similar but superior

12.10.6 Level 4 (Block-Level XOR ECC Parity)

Figure 12.21 RAID level 4 (block-level parity).

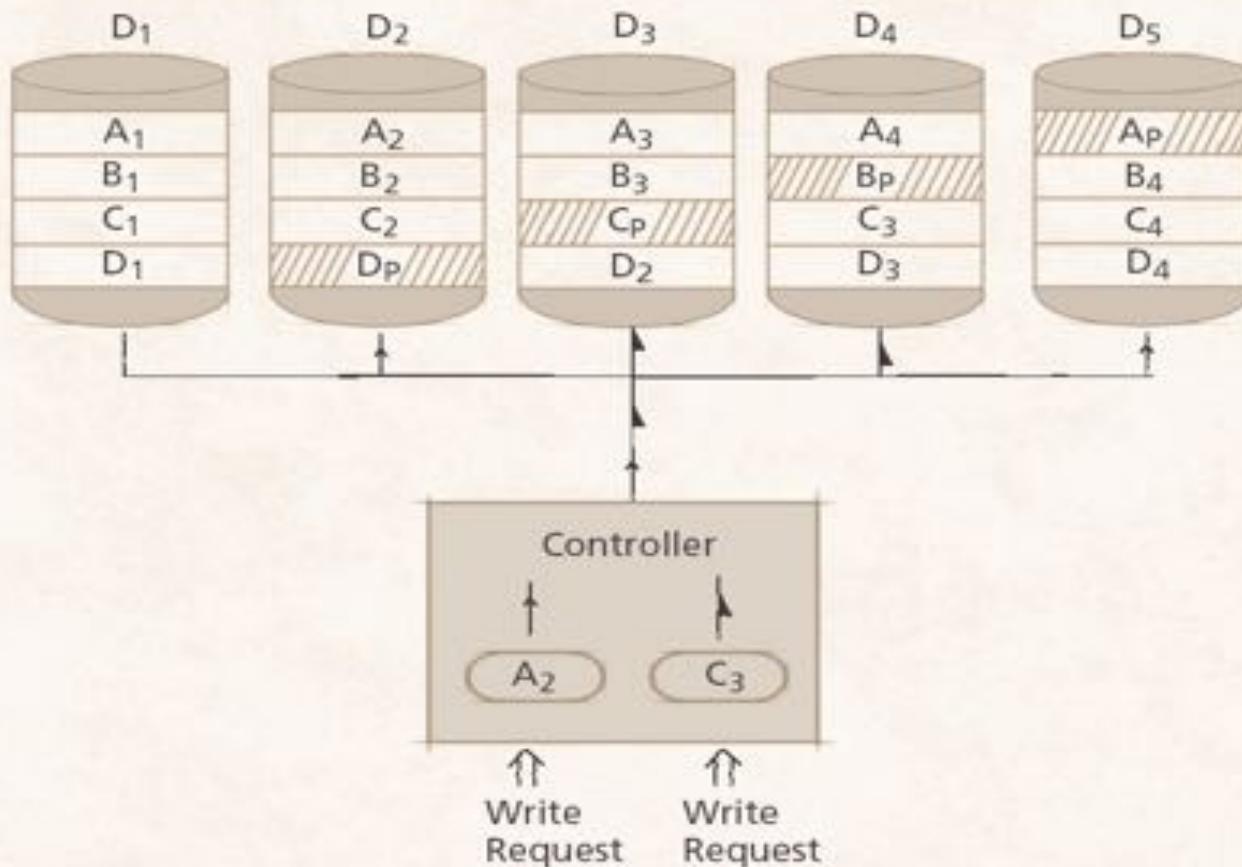


12.10.7 Level 5 (Block-Level Distributed XOR ECC Parity)

- **Level 5**
 - Similar to level 4
 - Removes write bottleneck of RAID level 4, because parity blocks are distributed across disks
 - Still must update parity information
 - For many small write operations, overhead can be substantial
 - Caching mechanisms can help alleviate this
 - For example, parity logging, update image and AFRAID
 - Faster and more reliable than levels 2–4
 - Costlier and more complex as well
 - Among most commonly implemented RAID levels

12.10.7 Level 5 (Block-Level Distributed XOR ECC Parity)

Figure 12.22 RAID level 5 (block-level distributed parity).



12.10.7 Level 5 (Block-Level Distributed XOR ECC Parity)

Figure 12.23 Comparison of RAID levels 0-5.

<i>RAID level</i>	<i>Read Concurrency</i>	<i>Write Concurrency</i>	<i>Redundancy</i>	<i>Striping Level</i>
0	Yes	Yes	None	Block
1	Yes	No	Mirroring	None
2	No	No	Hamming ECC parity	Bit
3	No	No	XOR ECC parity	Bit/byte
4	Yes	No	XOR ECC parity	Block
5	Yes	Yes	Distributed XOR ECC parity	Block

12.10.7 Level 5 (Block-Level Distributed XOR ECC Parity)

- Other RAID levels exist
 - No standard naming convention
 - RAID level 6 provides additional parity information to improve fault tolerance
 - RAID level 0 + 1: set of striped disks that are copied to set of mirror disks
 - RAID level 10: set of mirrored data that is striped across a set of disks
 - Others include 0+3, 0+5, 50, 1+5, 51, 53 and RAID level 7

Chapter 13 – File and Database Systems

Outline

- 13.1 **Introduction**
- 13.2 **Data Hierarchy**
- 13.3 **Files**
- 13.4 **File Systems**
 - 13.4.1 **Directories**
 - 13.4. **Metadata**
 - 13.4. **Mounting**
- 13.5 **File Organization**
- 13.6 **File Allocation**
 - 13.6.1 **Contiguous File Allocation**
 - 13.6.2 **Linked-List Noncontiguous File Allocation**
 - 13.6.3 **Tabular Noncontiguous File Allocation**
 - 13.6.4 **Indexed Noncontiguous File Allocation**
- 13.7 **Free Space Management**
- 13.8 **File Access Control**
 - 13.8.1 **Access Control Matrix**
 - 13.8.2 **Access Control by User Classes**
- 13.9 **Data Access Techniques**

Chapter 13 – File and Database Systems

Outline (continued)

- 13.10 Data Integrity Protection
 - 13.10.1 Backup and Recovery
 - 13.10.2 Data Integrity and Log-Structured File Systems
- 13.11 File Servers and Distributed Systems
- 13.12 Database Systems
 - 13.12.1 Advantages of Database Systems
 - 13.12.2 Data Access
 - 13.12.3 Relational Database Model
 - 13.12.4 Operating Systems and Database Systems

Objectives

- After reading this chapter, you should understand:
 - the need for file systems.
 - files, directories and the operations that can be performed on them.
 - organizing and managing a storage device's data and free space.
 - controlling access to data in a file system.
 - backup, recovery and file system integrity mechanisms.
 - database systems and models.

13.1 Introduction

- Files
 - Named collection of data that is manipulated as a unit
 - Reside on secondary storage devices
- Operating systems can create an interface that facilitates navigation of a user's files
 - File systems can protect such data from corruption or total loss from disasters
 - Systems that manage large amounts of shared data can benefit from databases as an alternative to files

13.2 Data Hierarchy

- Information is stored in computers according to a data hierarchy.
- Lowest level of data hierarchy is composed of bits
 - Bit patterns represent all data items of interest in computer systems

13.2 Data Hierarchy

- Next level in the data hierarchy is fixed-length patterns of bits such as bytes, characters and words
 - Byte: typically 8 bits
 - Word: the number of bits a processor can operate on at once
 - Characters map bytes (or groups of bytes) to symbols such as letters, numbers, punctuation and new lines
 - Three most popular character sets in use today: ASCII, EBCDIC and Unicode
 - Field: a group of characters
 - Record: a group of fields
 - File: a group of related records

13.2 Data Hierarchy

- Highest level of the data hierarchy is a file system or database
- A volume is a unit of data storage that may hold multiple files

13.3 Files

- File: a named collection of data that may be manipulated as a unit by operations such as:
 - Open
 - Close
 - Create
 - Destroy
 - Copy
 - Rename
 - List

13.3 Files

- Individual data items within a file may be manipulated by operations like:
 - Read
 - Write
 - Update
 - Insert
 - Delete
- File characteristics include:
 - Location
 - Accessibility
 - Type
 - Volatility
 - Activity
- Files can consist of one or more records

13.4 File Systems

- **File systems**
 - Organize files and manages access to data
 - Responsible for file management, auxiliary storage management, file integrity mechanisms and access methods
 - Primarily are concerned with managing secondary storage space, particularly disk storage

13.4 File Systems

- File system characteristics
 - Should exhibit device independence:
 - Users should be able to refer to their files by symbolic names rather than having to use physical device names
 - Should also provide backup and recovery capabilities to prevent either accidental loss or malicious destruction of information
 - May also provide encryption and decryption capabilities to make information useful only to its intended audience

13.4.1 Directories

- Directories:
 - Files containing the names and locations of other files in the file system, to organize and quickly locate files
- Directory entry stores information such as:
 - File name
 - Location
 - Size
 - Type
 - Accessed
 - Modified and creation times

13.4.1 Directories

Figure 13.1 Directory file contents example.

<i>Directory Field</i>	<i>Description</i>
Name	Character string representing the file's name.
Location	Physical block or logical location of the file in the file system (i.e., a pathname).
Size	Number of bytes consumed by the file.
Type	Description of the file's purpose (e.g., data file or directory file).
Access time	Time the file was last accessed.
Modified time	Time the file was last modified.
Creation time	Time the file was created.

13.4.1 Directories

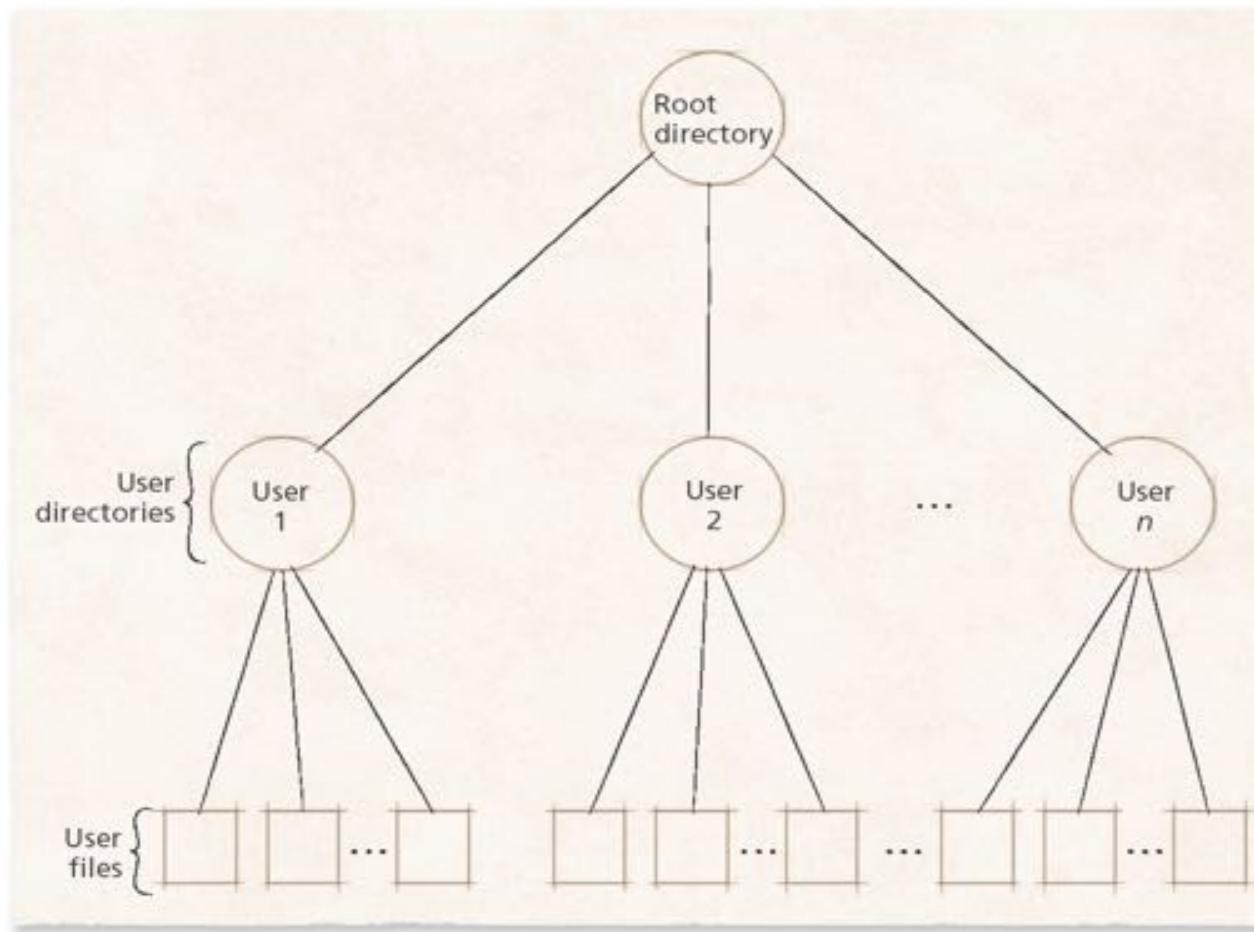
- Single-level (or flat) file system:
 - Simplest file system organization
 - Stores all of its files using one directory
 - No two files can have the same name
 - File system must perform a linear search of the directory contents to locate each file, which can lead to poor performance

13.4.1 Directories

- Hierarchical file system:
 - A root indicates where on the storage device the root directory begins
 - The root directory points to the various directories, each of which contains an entry for each of its files
 - File names need be unique only within a given user directory
 - The name of a file is usually formed as the pathname from the root directory to the file

13.4.1 Directories

Figure 13.2 Two-level hierarchical file system.

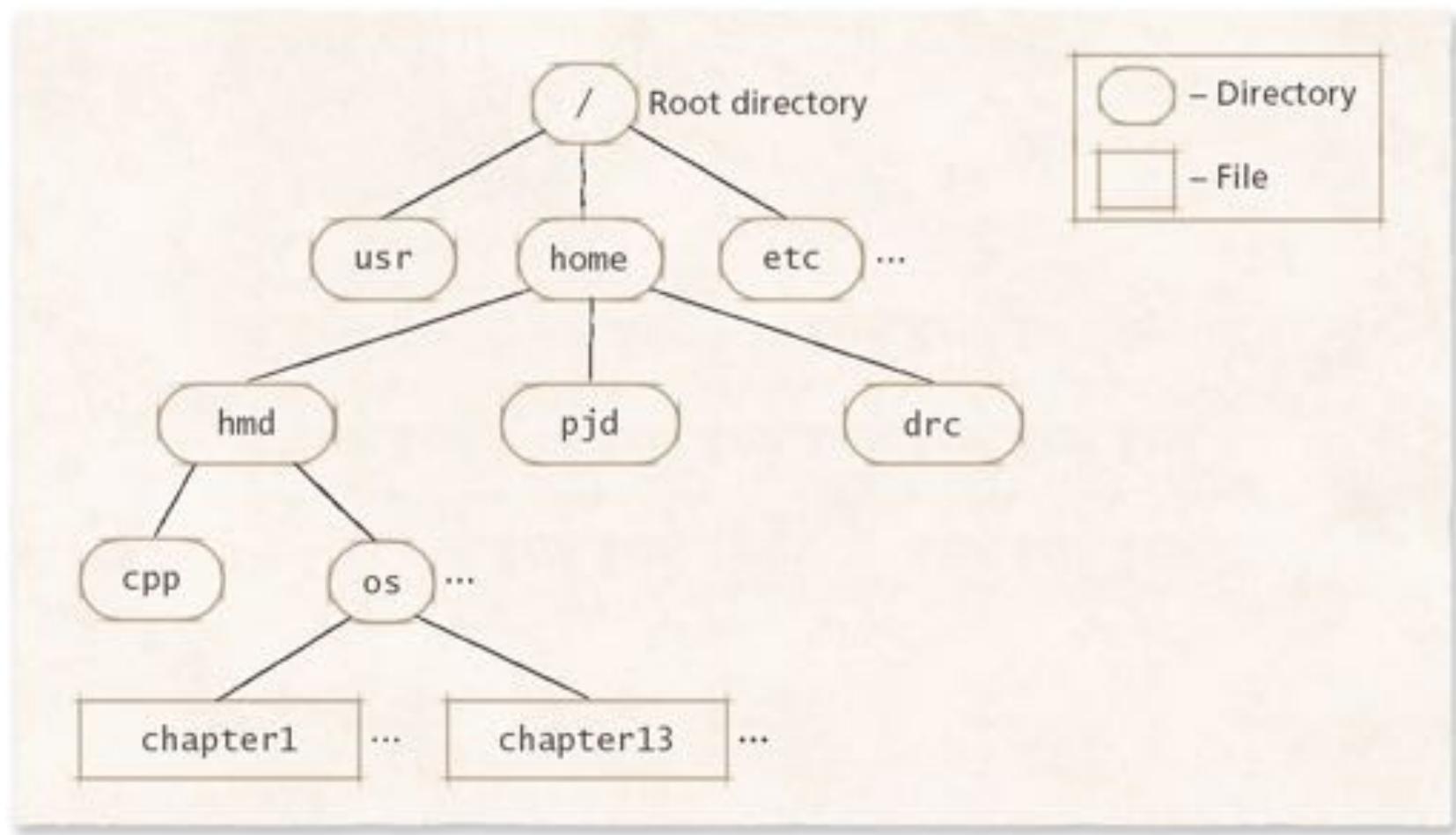


13.4.1 Directories

- Working directory
 - Simplifies navigation using pathnames
 - Enables users to specify a pathname that does not begin at the root directory (i.e., a relative path)
 - Absolute path (i.e., the path beginning at the root) = working directory + relative path

13.4.1 Directories

Figure 13.3 Example hierarchical file system contents.



13.4.1 Directories

- Link: a directory entry that references a data file or directory located in a different directory
 - Facilitates data sharing and can make it easier for users to access files located throughout a file system's directory structure
 - Soft link: directory entry containing the pathname for another file
 - Hard link: directory entry that specifies the location of the file (typically a block number) on the storage device

13.4.1 Directories

- Links (Cont.)
 - Because a hard link specifies a physical location of a file, it references invalid data when the physical location of its corresponding file changes
 - Because soft links store the logical location of the file in the file system, they do not require updating when file data is moved
 - However, if a user moves a file to different directory or renames the file, any soft links to that file are no longer valid

13.4.1 Directories

Figure 13.4 Links in a file system.

Directory	
Name	Location
foo	467
bar	843
:	:
foo_hard	467
foo_soft	./foo

Soft link

Hard link to foo

The diagram illustrates a directory structure. A vertical line on the left represents a directory entry, with an arrow pointing to the 'Name' column of the table. Inside the directory entry, there is a horizontal line labeled 'Soft link'. To the right of the table, an arrow points from the 'foo_soft' entry to the text 'Hard link to foo', indicating that 'foo_soft' is a soft link to 'foo'.

13.4.2 Metadata

- Metadata
 - Information that protects the integrity of the file system
 - Cannot be modified directly by users
- Many file systems create a superblock to store critical information that protects the integrity of the file system
 - A superblock might contain:
 - The file system identifier
 - The location of the storage device's free blocks
 - To reduce the risk of data loss, most file systems distribute redundant copies of the superblock throughout the storage device

13.4.2 Metadata

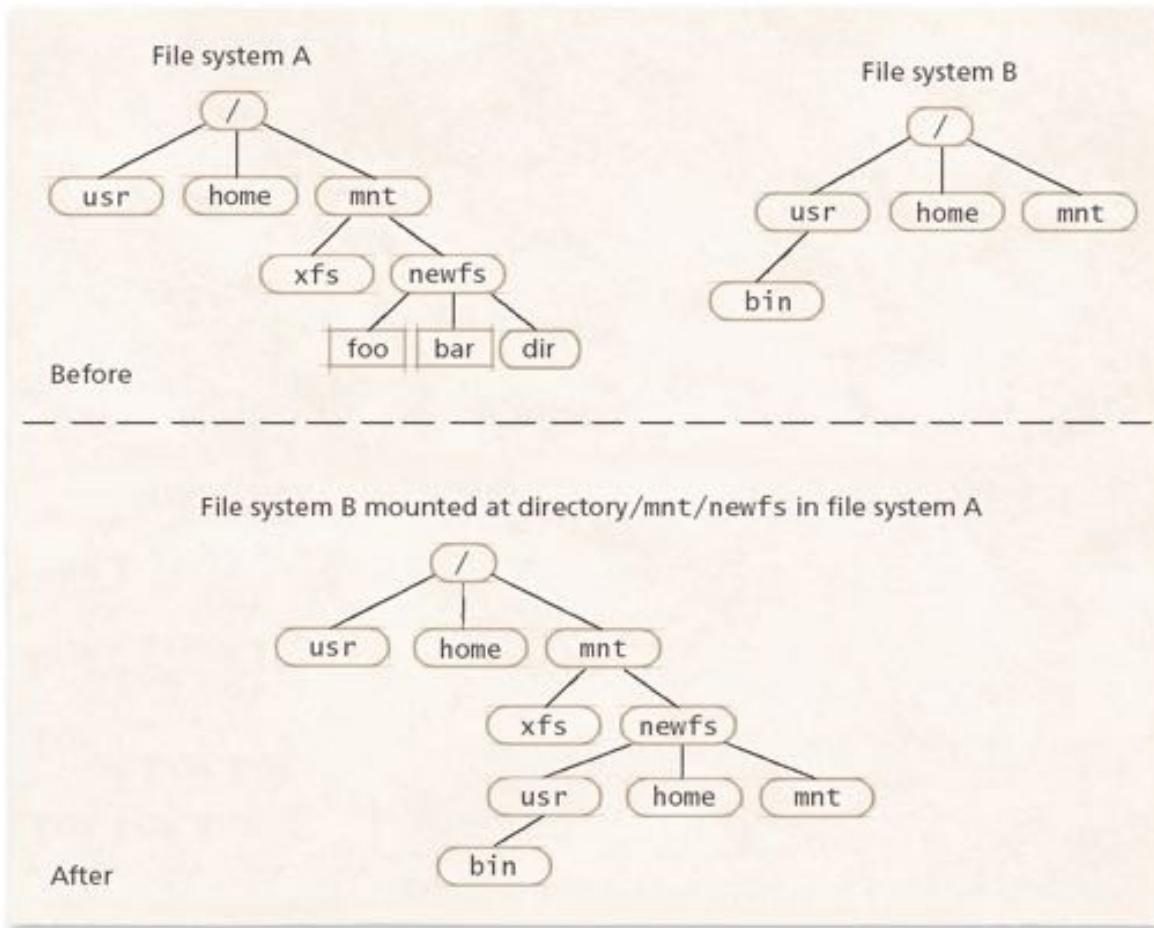
- File open operation returns a file descriptor
 - A non-negative integer index into the open-file table
- From this point on, access to the file is directed through the file descriptor
- To enable fast access to file-specific information such as permissions, the open-file table often contains file control blocks, also called file attributes:
 - Highly system-dependent structures that might include the file's symbolic name, location in secondary storage, access control data and so on

13.4.3 Mounting

- Mount operation
 - Combines multiple file systems into one namespace so that they can be referenced from a single root directory
 - Assigns a directory, called the mount point, in the native file system to the root of the mounted file system
- File systems manage mounted directories with mount tables:
 - Contain information about the location of mount points and the devices to which they point
- When the native file system encounters a mount point, it uses the mount table to determine the device and type of the mounted file system
- Users can create soft links to files in mounted file systems but cannot create hard links between file systems

13.4.3 Mounting

Figure 13.5 Mounting a file system.



13.5 File Organization

- File organization: the manner in which the records of a file are arranged on secondary storage
- File organization schemes include:
 - Sequential
 - Direct
 - Indexed nonsequential
 - Partitioned

13.6 File Allocation

- File allocation
 - Problem of allocating and freeing space on secondary storage is somewhat like that experienced in primary storage allocation under variable-partition multiprogramming
 - Contiguous allocation systems have generally been replaced by more dynamic noncontiguous allocation systems
 - Files tend to grow or shrink over time
 - Users rarely know in advance how large their files will be

13.6.1 Contiguous File Allocation

- Contiguous allocation
 - Place file data at contiguous addresses on the storage device
 - Advantages
 - Successive logical records typically are physically adjacent to one another
 - Disadvantages
 - External fragmentation
 - Poor performance can result if files grow and shrink over time
 - If a file grows beyond the size originally specified and no contiguous free blocks are available, it must be transferred to a new area of adequate size, leading to additional I/O operations.

13.6.2 Linked-List Noncontiguous File Allocation

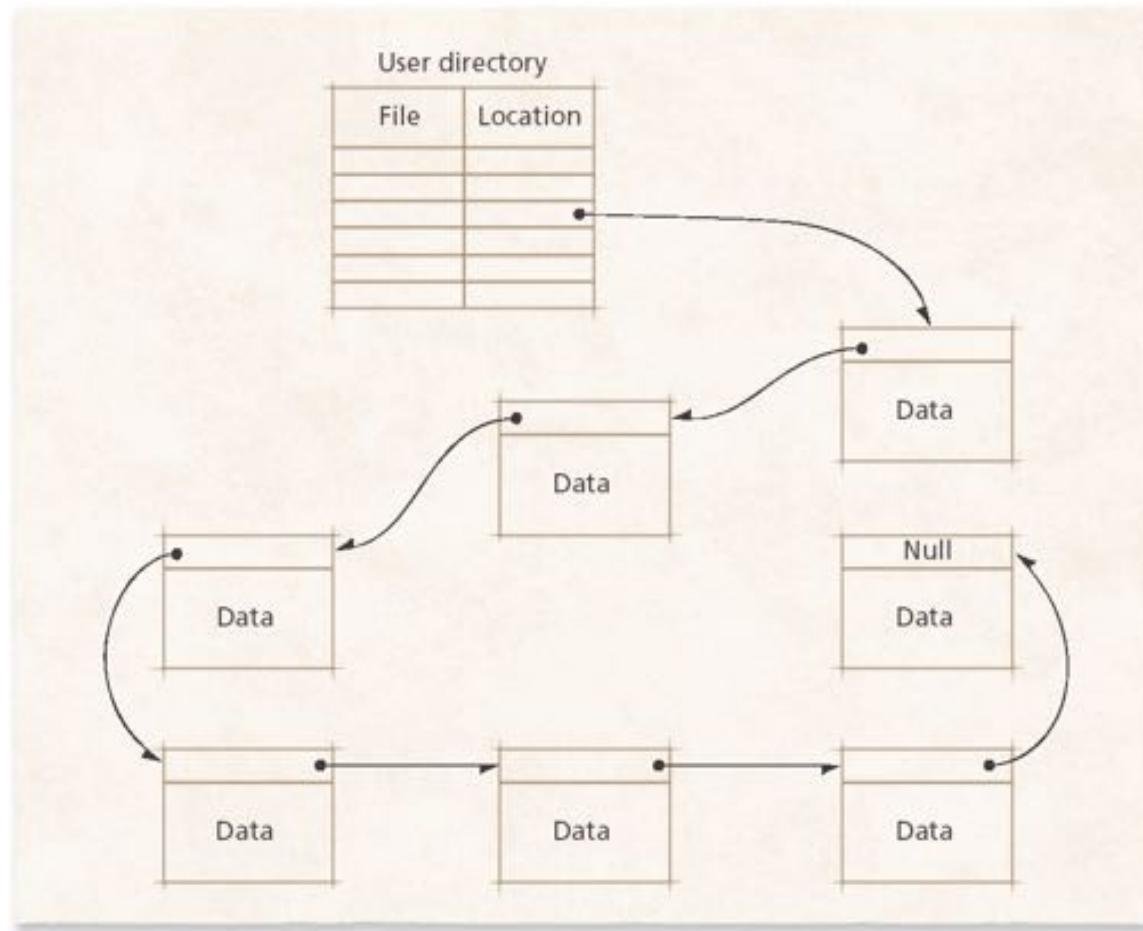
- Sector-based linked-list noncontiguous file allocation scheme:
 - A directory entry points to the first sector of a file
 - The data portion of a sector stores the contents of the file
 - The pointer portion points to the file's next sector
 - Sectors belonging to a common file form a linked list

13.6.2 Linked-List Noncontiguous File Allocation

- When performing block allocation, the system allocates blocks of contiguous sectors (sometimes called extents)
- Block chaining
 - Entries in the user directory point to the first block of each file
 - File blocks contain:
 - A data block
 - A pointer to the next block

13.6.2 Linked-List Noncontiguous File Allocation

Figure 13.6 Noncontiguous file allocation using a linked list.



13.6.2 Linked-List Noncontiguous File Allocation

- When locating a record
 - The chain must be searched from the beginning
 - If the blocks are dispersed throughout the storage device (which is normal), the search process can be slow as block-to-block seeks occur
- Insertion and deletion are done by modifying the pointer in the previous block

13.6.2 Linked-List Noncontiguous File Allocation

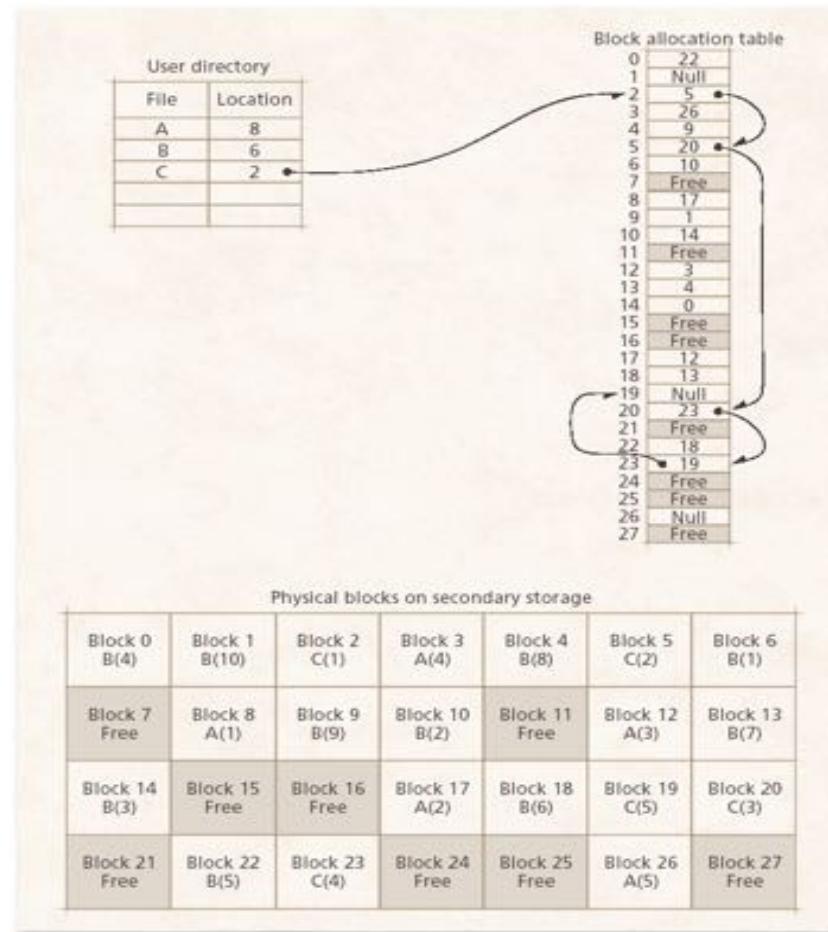
- Large block sizes
 - Can result in significant internal fragmentation
- Small block sizes
 - May cause file data to be spread across multiple blocks dispersed throughout the storage device
 - Poor performance as the storage device performs many seeks to access all the records of a file

13.6.3 Tabular Noncontiguous File Allocation

- Tabular noncontiguous file allocation
 - Uses tables storing pointers to file blocks
 - Reduces the number of lengthy seeks required to access a particular record
 - Directory entries indicate the first block of a file
 - Current block number is used as an index into the block allocation table to determine the location of the next block.
 - If the current block is the file's last block, then its block allocation table entry is null

13.6.3 Tabular Noncontiguous File Allocation

Figure 13.7 Tabular noncontiguous file allocation.



13.6.3 Tabular Noncontiguous File Allocation

- Pointers that locate file data are stored in a central location
 - The table can be cached so that the chain of blocks that compose a file can be traversed quickly
 - Improves access times
- To locate the last record of a file, however:
 - The file system might need to follow many pointers in the block allocation table
 - Could take significant time

13.6.3 Tabular Noncontiguous File Allocation

- When a storage device contains many blocks:
 - The block allocation table can become large and fragmented
 - Reduces file system performance
- A popular implementation of tabular noncontiguous file allocation is Microsoft's FAT file system

13.6.4 Indexed Noncontiguous File Allocation

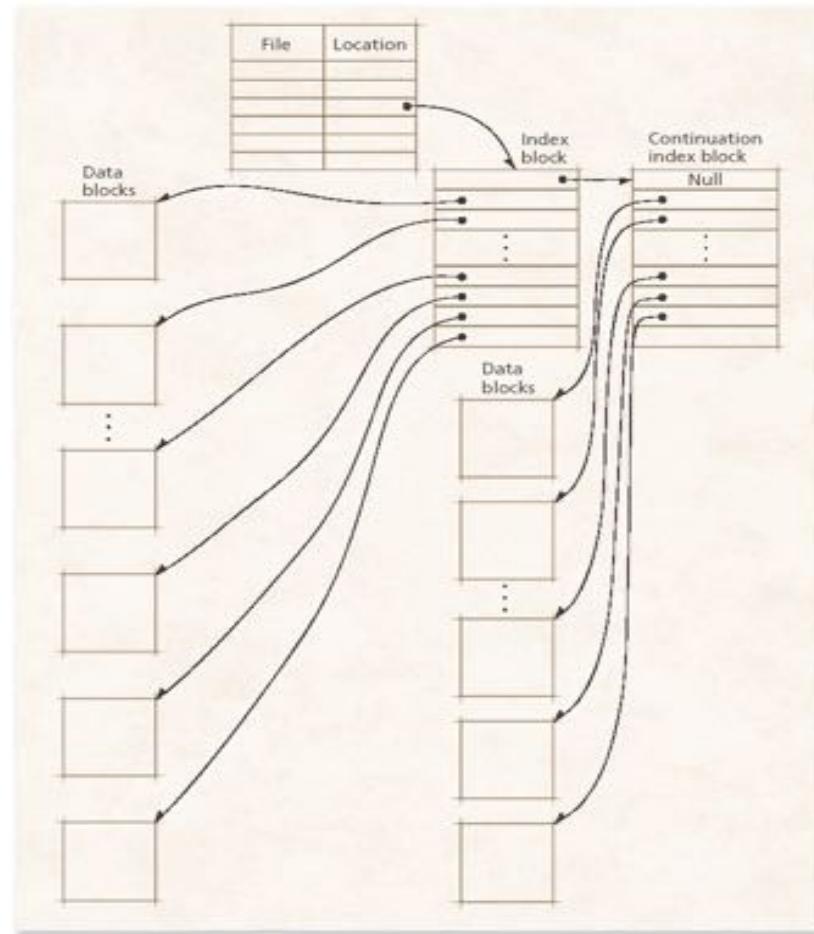
- Indexed noncontiguous file allocation:
 - Each file has an index block or several index blocks
 - Index blocks contain a list of pointers that point to file data blocks
 - A file's directory entry points to its index block, which may reserve the last few entries to store pointers to more index blocks, a technique called chaining

13.6.4 Indexed Noncontiguous File Allocation

- Primary advantage of index block chaining over simple linked-list implementations:
 - Searching may take place in the index blocks themselves.
 - File systems typically place index blocks near the data blocks they reference, so the data blocks can be accessed quickly after their index block is loaded

13.6.4 Indexed Noncontiguous File Allocation

Figure 13.8 Index block chaining.

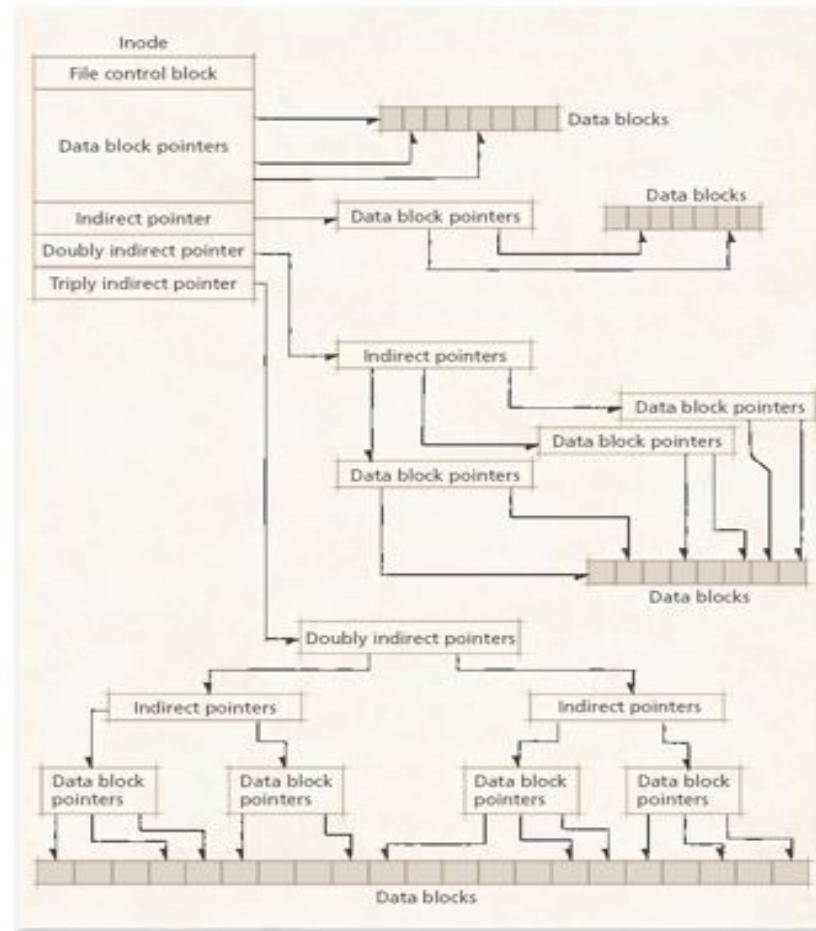


13.6.4 Indexed Noncontiguous File Allocation

- Index blocks are called inodes (i.e., index nodes) in UNIX-based operating systems

13.6.4 Indexed Noncontiguous File Allocation

Figure 13.9 Inode structure.

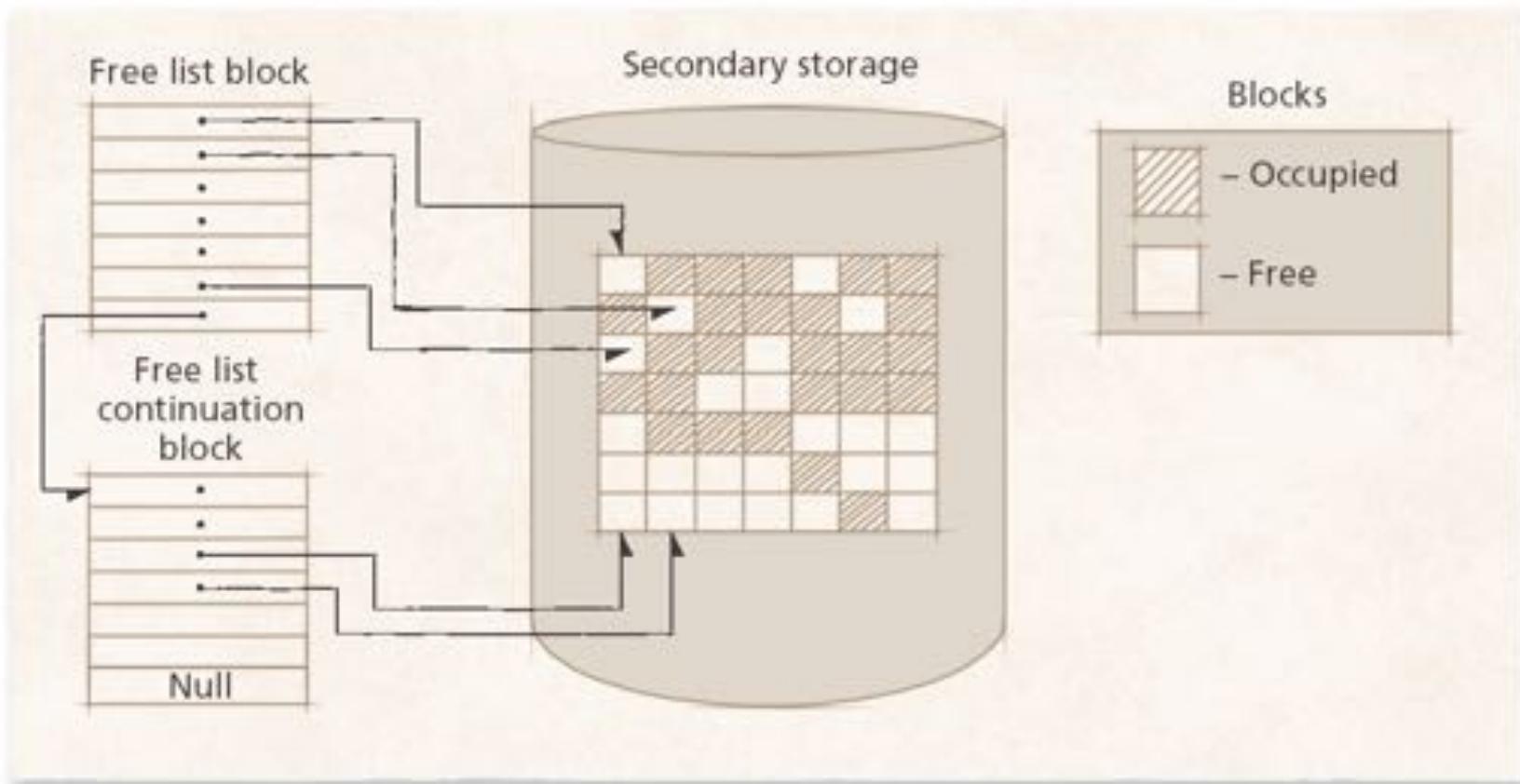


13.7 Free Space Management

- Some systems use a free list to manage the storage device's free space
 - Free list: Linked list of blocks containing the locations of free blocks
 - Blocks are allocated from the beginning of the free list
 - Newly freed blocks are appended to the end of the list
- Low overhead to perform free list maintenance operations
- Files are likely to be allocated in noncontiguous blocks
 - Increases file access time

13.7 Free Space Management

Figure 13.10 Free space management using a free list.

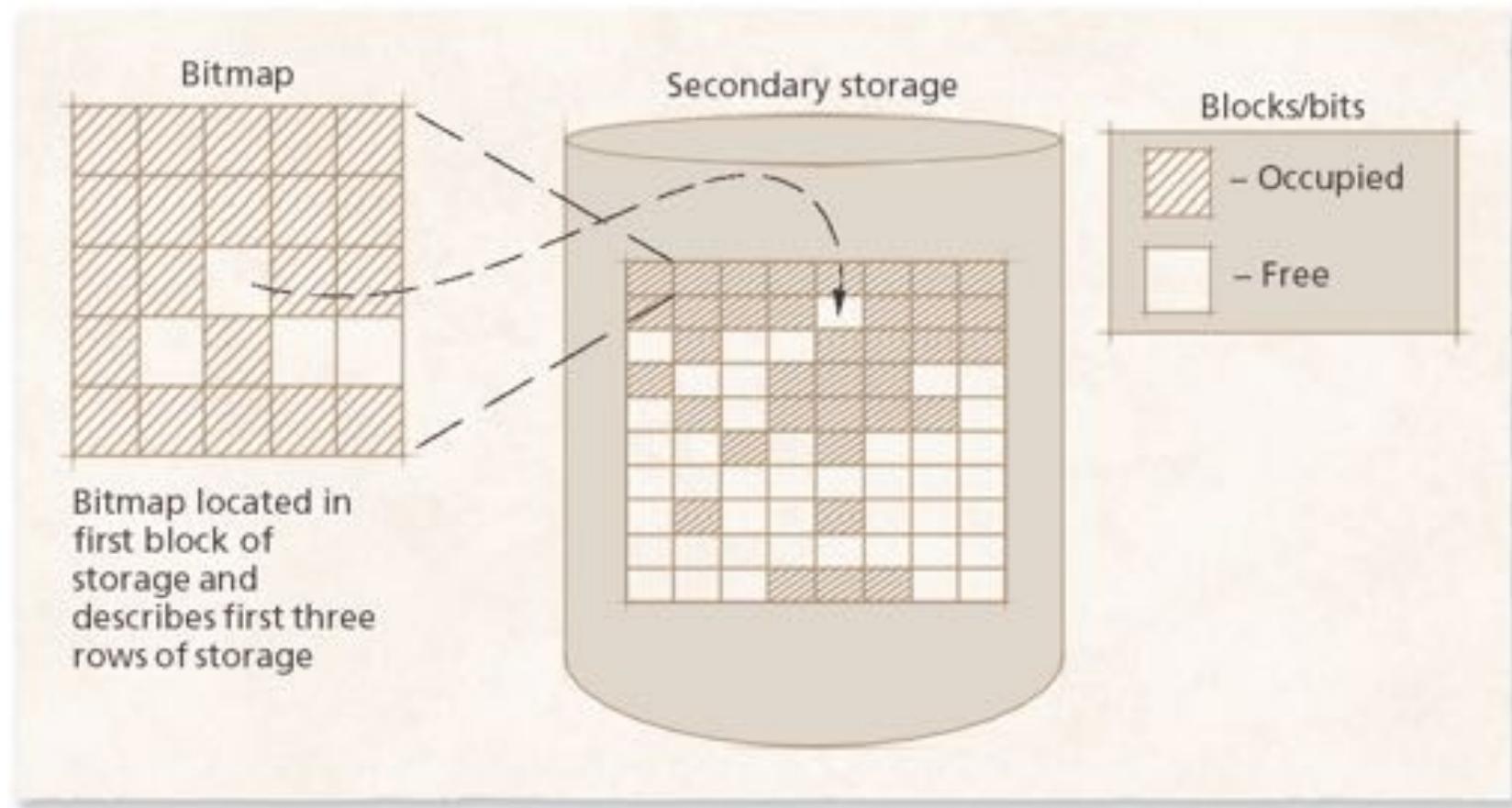


13.7 Free Space Management

- A bitmap contains one bit for each block in memory
 - i th bit corresponds to the i th block on the storage device
- Advantage of bitmaps over free lists:
 - The file system can quickly determine if contiguous blocks are available at certain locations on secondary storage
- Disadvantage of bitmaps:
 - The file system may need to search the entire bitmap to find a free block, resulting in substantial execution overhead

13.7 Free Space Management

Figure 13.11 Free space management using a bitmap.



13.8 File Access Control

- Files are often used to store sensitive data such as:
 - Credit card numbers
 - Passwords
 - Social security numbers
- Therefore, they should include mechanisms to control user access to data.
 - Access control matrix
 - Access control by user classes

13.8.1 Access Control Matrix

- Two-dimensional access control matrix:
 - Entry a_{ij} is 1 if user i is allowed access to file j
 - Otherwise $a_{ij} = 0$
- In an installation with a large number of users and a large number of files, this matrix generally would be large and sparse
- Inappropriate for most systems

13.8.1 Access Control Matrix

Figure 13.12 Access control matrix.

User \ File	1	2	3	4	5	6	7	8	9	10
1	1	1	0	0	0	0	0	0	0	0
2	0	0	1	0	1	0	0	0	0	0
3	0	1	0	1	0	1	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	1	1	1	1	1
6	0	0	0	0	0	1	1	0	0	0
7	1	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	0	0	0	0	0
9	1	1	1	1	0	0	0	0	1	1
10	1	1	0	0	1	1	0	0	0	1

13.8.2 Access Control by User Classes

- A technique that requires considerably less space is to control access to various user classes
- User classes can include:
 - The file owner
 - A specified user
 - Group
 - Project
 - Public
- Access control data
 - Can be stored as part of the file control block
 - Often consumes an insignificant amount of space

13.9 Data Access Techniques

- Today's operating systems generally provide many access methods
- Queued access methods
 - Used when the sequence in which records are to be processed can be anticipated, such as in sequential and indexed sequential accessing
 - Perform anticipatory buffering and scheduling of I/O operations
- Basic access methods
 - Normally used when the sequence in which records are to be accessed cannot be anticipated, particularly with direct accessing

13.9 Data Access Techniques

- Memory-mapped files
 - Map file data to a process's virtual address space instead of using a file system cache
 - Because references to memory-mapped files occur in a process's virtual address space, the virtual memory manager can make page-replacement decisions based on each process's reference pattern

13.10 Data Integrity Protection

- Computer systems often store critical information, such as:
 - Inventories
 - Financial records
 - Personal information
- System crashes, natural disasters and malicious programs can destroy this information
- The results of such events can be catastrophic
- Operating systems and data storage systems should be fault tolerant:
 - Account for the possibility of disasters and provide techniques to recover from them

13.10.1 Backup and Recovery

- Backup techniques
 - Store redundant copies of information
- Recovery techniques
 - Enable the system to restore data after a system failure
- Physical safeguards such as locks and fire alarms are the lowest level of data protection
- Performing periodic backups is the most common technique used to ensure the continued availability of data

13.10.1 Backup and Recovery

- Physical backups
 - Duplicate a storage device's data at the bit level
- Logical backups
 - Store file system data and its logical structure
 - Inspect the directory structure to determine which files need to be backed up, then write these files to a backup device in a common, often compressed, archival format
- Incremental backups are logical backups that store only file system data that has changed since the previous backup

13.10.2 Data Integrity and Log-Structured File Systems

- In systems that cannot tolerate loss of data or downtime, RAID and transaction logging are appropriate
- If a system failure occurs during a write operation, file data may be left in an inconsistent state
- Transaction-based file systems
 - Reduce data loss using atomic transactions:
 - Perform a group of operations in their entirety or not at all
 - If an error occurs that prevents a transaction from completing, it is rolled back by returning the system to the state before the transaction began

13.10.2 Data Integrity and Log-Structured File Systems

- Transaction-based file systems
 - Reduce data loss using atomic transactions:
 - Perform a group of operations in their entirety or not at all
 - If an error occurs that prevents a transaction from completing, it is rolled back by returning the system to the state before the transaction began

13.10.2 Data Integrity and Log-Structured File Systems

- Atomic transactions
 - Can be implemented by recording the result of each operation in a log file instead of modifying existing data
 - Once the transaction has completed, it is committed by recording a sentinel value in the log
- Checkpoints
 - To reduce the time spent reprocessing transactions in the log, most transaction-based systems maintain checkpoints that point to the last transaction that has been transferred to permanent storage
 - If the system crashes, it need only examine transactions after the checkpoint
- Shadow paging implements atomic transactions by writing modified data to a free block instead of the original block

13.10.2 Data Integrity and Log-Structured File Systems

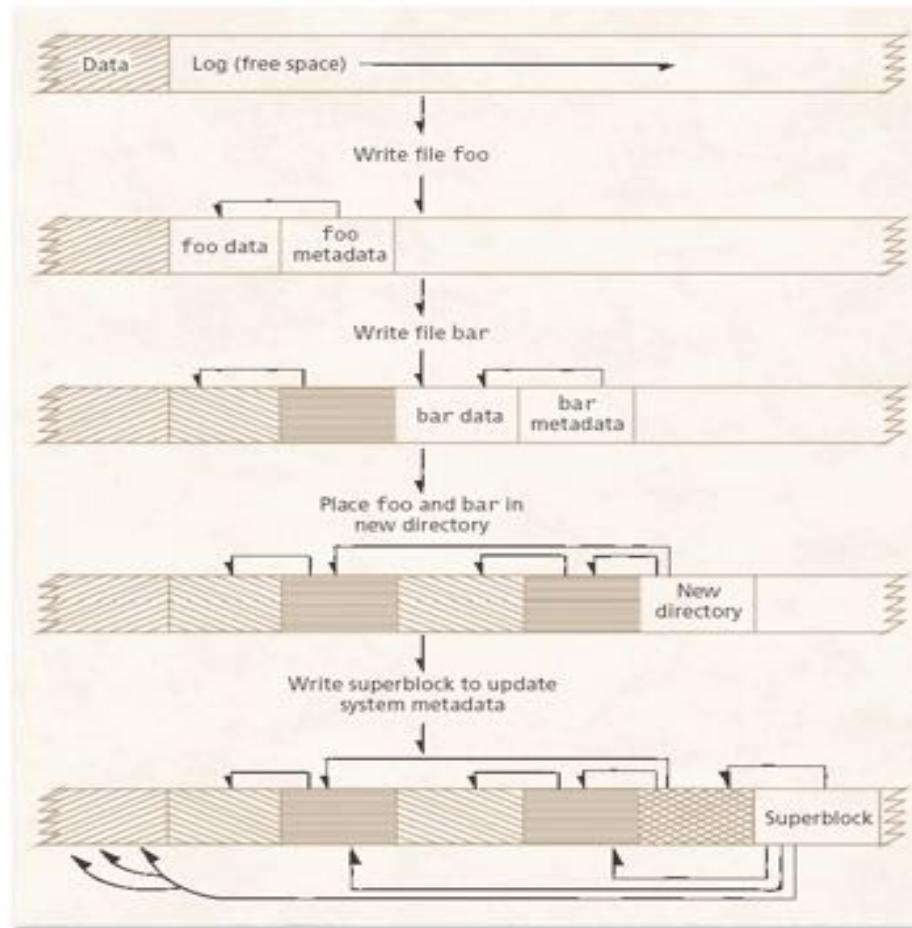
- Log-structured file systems (LFS)
 - Also called journaling file systems
 - Perform all file system operations as logged transactions to ensure that they do not leave the system in an inconsistent state
 - The entire disk serves as a log file
 - New data is written sequentially in the log file's free space

13.10.2 Data Integrity and Log-Structured File Systems

- Because modified directories and metadata are always written to the end of the log, an LFS might need to read the entire log to locate a particular file, leading to poor read performance
- To reduce this problem, an LFS
 - Caches locations of file system metadata
 - Occasionally writes inode maps or superblocks to the log to indicate the location of other metadata
 - Enables the operating system to locate and cache file metadata quickly when the system boots

13.10.2 Data Integrity and Log-Structured File Systems

Figure 13.13 Log-structured file system.



13.10.2 Data Integrity and Log-Structured File Systems

- Some file systems attempt to reduce the cost of log-structured file systems by using a log only to store metadata
 - Ensures file system integrity with relatively low overhead
 - Does not ensure file integrity in the event of a system failure

13.10.2 Data Integrity and Log-Structured File Systems

- Data is written to the log sequentially
 - Each LFS write requires only a single seek while there is still space on disk
- When the log fills, the file system's free space is likely to be fragmented
- This can lead to poor read and write performance
- To address this issue, an LFS can create contiguous free space in the log by copying valid data to a contiguous region at the end of the log

13.11 File Servers and Distributed Systems

- One approach to handling nonlocal file references in a computer network is to route all such requests to a file server
 - A computer system dedicated to resolving intercomputer file references
 - Centralizes control of these references
 - File server could easily become a bottleneck, because all client computers send all requests to the server
- A better approach is to let the separate computers communicate directly with one another

13.12 Database Systems

- **Database**
 - Centrally controlled collection of data stored in a standardized format
- **A database system involves:**
 - Data
 - Hardware on which the data resides
 - Software that controls access to data (called a database management system or DBMS)

13.12.1 Advantages of Database Systems

- Databases reduce data redundancy and prevent data from being in an inconsistent state
 - Redundancy is reduced by combining identical data from separate files
- Databases also facilitate data sharing

13.12.2 Data Access

- **Data independence**
 - Applications need not be concerned with how data is physically stored or accessed
 - Makes it possible for the storage structure and accessing strategy to be modified in response to the installation's changing requirements, but without the need to modify functioning applications

13.12.2 Data Access

- Database languages
 - Allow database independence by providing a standard way to access information
 - A database language consists of:
 - Data definition language (DDL)
 - Data manipulation language (DML)
 - Query language

13.12.2 Data Access

- Database languages (cont.)
 - A DDL specifies how data are organized and related and the DML enables data to be modified
 - A query language is a part of the DML that allows users to create queries that search the database for data that meets certain criteria
 - The Structured Query Language (SQL) is currently one of the most popular database languages

13.12.2 Data Access

- **Distributed database**
 - Database that is spread throughout the computer systems of a network
 - Facilitates efficient data access across many sets of data that reside on different computers

13.12.3 Relational Database Model

- Databases are based on models that describe how data and their relationships are viewed
- Relational model is a logical structure rather than a physical one
- The principles of relational database management are independent of the physical implementation of data structures

13.12.3 Relational Database Model

- Relations
 - Indicate the various attributes of an entity
 - Any particular element of a relation is called a tuple (row).
 - Each attribute (column) of the relation belongs to a single domain
 - The number of attributes in a relation is the degree of the relation
 - A projection operation forms a subset of the attributes
 - A join operation combines relations to produce more complex relations
- The relational database model is relatively easy to implement

13.12.3 Relational Database Model

Figure 13.14 Relation in a relational database.

Relation: EMPLOYEE					
Number	Name	Department	Salary	Location	
23603	Jones, A.	413	1100	New Jersey	
24568	Kerwin, R.	413	2000	New Jersey	
34589	Larson, P.	642	1800	Los Angeles	
35761	Myers, B.	611	1400	Orlando	
47132	Neumann, C.	413	9000	New Jersey	
78321	Stevens, T.	611	8500	Orlando	

A tuple {

Primary key An attribute

13.12.3 Relational Database Model

Figure 13.15 Relation formed by projection.

Relation: DEPARTMENT-LOCATOR

Department

413

611

642

location

NEW JERSEY

ORLANDO

LOS ANGELES

13.12.3 Relational Database Model

Figure 13.16 SQL query.

```
1 -- SQL query to generate the table in Fig. 13.15
2 SELECT DISTINCT Department, Location
3 FROM EMPLOYEE
4 ORDER BY Department ASC
5
```

13.12.4 Operating Systems and Database Systems

- Various operating system services support database management systems, namely:
 - Buffer pool management
 - File system
 - Scheduling
 - Process management
 - Interprocess communication
 - Consistency control
 - Paged virtual memory
- Most of these features are not specifically optimized to DBMS environments, so DBMS designers have tended to bypass operating system services in favor of supplying their own