OPERATING SYSTEMS
M.Sc (COMPUTER SCIENCE)
SEMESTER-I, PAPER-III

Lesson Writers:

Dr. Neelima Guntupalli
Asst.Professor,
Dept. of Comp.Science,
Acharya Nagarjuna University,
Nagarjunanagar – 522 510

Dr. Vasantha Rudramalla
Faculty
Dept. CS &E
Acharya Nagarjuna University
Nagarjunanagar – 522 510

Mrs. Appikatla Pushpa Latha
Faculty, Dept. of CS&E
Acharya Nagarjuna University
Nagarjunanagar – 522 510

Dr. U Surya Kameswari
Assistant Professor
Department of CS&E
Acharya Nagarjuna University
Nagarjunanagar – 522 510

Editor

Dr. K. Lavanya
Asst. Professor,
Dept. Of CS&E
Acharya Nagarjuna University
Nagarjunanagar – 522 510.

Director, I/c.
Prof. V. Venkateswarlu
M.A., M.P.S., M.S.W ., M.Phil., Ph.D.
Professor
Centre for Distance Education
Acharya Nagarjuna University
Nagarjuna Nagar 522 510

Ph: 0863-2346222, 2346208
0863- 2346259 (Study Material)
Website www.anucde.info
E-mail: anucdedirector@gmail.com

M.Sc Computer Science

First Edition : 2025

No. of Copies :

This book is exclusively prepared for the use of students of M.Sc (Computer Science), Centre for Distance Education, Acharya Nagarjuna University and this book is meant for limited circulation only.

Published by:

Director **I/c**
Prof. V. Venkateswarlu,
M.A., M.P.S., M.S.W . M.Phil., Ph.D.
Centre for Distance Education,
Acharya Nagarjuna University

Printed at:

FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A+' grade from the NAAC in the year 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 affiliated colleges spread over the two districts of Guntur and Prakasam.

The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the door step of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.Sc., B.A., B.B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavors.

Prof. K. Gangadhara Rao
M.Tech., Ph.D.,

Vice-Chancellor I/c
Acharya Nagarjuna University

M.Sc. Computer Science
Semester-I, Paper-III
103CP24 -Operating System

Syllabus

UNIT-1

Introduction : What Operating Systems Do – Computer System Orgranization – Computer system Architecture – Operating System Structure – Operating System Operations – Process Management – Memory Management – Storage Management – Protection and Security – Distributed Systems – Special purpose Systems – Computing Environments.

System Structure: Operating System Services – User Operating System Interface – System Calls – Types of System Calls – System Programs – Operating System Design and Implementation – Operating System Structure – Virtual Machine – Operating System Generation – System Boot.

Process Concept: Overview – Process Scheduling – Operations on Processes – Interprocess Communication – Examples of IPC Systems – Communication in Client Server Systems.

UNIT–II:

Multithreaded Programming : Overview – Multithreading Models – Thread Libraries – Threading Issues – Operating System Examples.
Process Scheduling: Basic Concepts – Scheduling Criteria – Scheduling Algorithms – Multiple Processor Scheduling – Thread Scheduling.
Synchronization: Background – The Critical Section Problem – Peterson's solution – Synchronization Hardware – Semaphores – Classic Problem of Synchronization – Monitors – Synchronization Examples – Atomic Transaction.

UNIT–III:

Deadlocks : System Model – Deadlock Characterization – Methods for Handling Deadlocks – Deadlock Prevention – Deadlock Avoidance – Deadlock Detection – Recovery from Deadlock.
Memory Management Strategies: Background – Swapping – Contiguous Memory Allocation – Paging – Structure of the Page Table – Segmentation – Example: The Intel Pentium.
Virtual Memory Management: Background – Demand Paging – Copy on Write – Page Replacement – Allocation of Frames – Thrashing.

UNIT–IV:

File System : File Concept – Access Methods – Directory Structure – File System Mounting – File Sharing – Protection.
Implementing File Systems : File System Structure – File System Implementation – Directory Implementation – Allocation Methods – Free Space Management – Efficiency and Performance – Recovery – Log structured File Systems.
Secondary Storage Structure : Overview of Mass – Storage Structure – Disk Structure – Disk Attachment – Disk Scheduling – Disk Management – Swap Space Management – RAID

structure.
I/O Systems: Overview – I/O Hardware – Application I/O Interface – Kernal I/O Interface – Transforming I/O requests to Hardware Operations – Streams – Performance.

Prescribed Book:

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. "Operating System Principles", Seventh Edition, Wiley.

Reference Book:

1. William Stallings, "Operating Systems – Internals and Design Principles", Fifth Edition, Pearson Education (2007)
2. Achyut S Godbole, "Operating Systems", Second Edition, TMH (2007).
3. Flynn/McHoes, "Operating Systems", Cengage Learning (2008).
4. Deitel&Deitel, "Operating Systems",Third Edition, Pearson Education (2008)

(103CP24)

M.SC DEGREE EXAMINATION, Model QP

Computer Science – First Semester

Operating System

Time: 3hours Max. Marks: 70

Answer ONE Question from each unit 5 x 14-70 M

UNIT – I

1. Explain the Operating System Structure?

OR

2. Write the Types of System Calls?

UNIT – II

3. Define Thread? Discuss the Threading Issues.

OR

4. Briefly explain the Scheduling Algorithms?

UNIT – III

5. Describe the Semaphores?

OR

6. a) What is Deadlock? Write the Methods for Handling Deadlocks?
   b) Differences between the Deadlock Prevention and Deadlock Avoidance?

UNIT – IV

7. Discuss about the Paging?

OR

8.Explain the Thrashing?

UNIT – V

9. What is File system? Discuss the File Sharing?

OR

10. Explain the RAID structure?

CONTENTS

CHAPTER 1

INTRODUCTION

Aim and Objectives:

- Learn the basic functions of operating systems.
- Understand computer system organization and architecture.
- Explore operating system structures and operations.
- Study process, memory, and storage management.
- Understand protection, security, and distributed systems.
- Learn about special-purpose systems and computing environments.

Structure:

1.1 What Operating Systems Do
  1.1.1 User View
  1.1.2 System View
  1.1.3 Defining Operating Systems
1.2 Computer System Organization
  1.2.1 Computer-System Operation
  1.2.2 Storage Structure
  1.2.3 I/O Structure
1.3 Computer system Architecture
  1.3.1 Single-Processor Systems
  1.3.2 Multiprocessor Systems
  1.3.3 Clustered Systems
1.4 Operating System Structure
1.5 Operating System Operations
  1.5.1 Dual-Mode Operation
  1.5.2 Timer
1.6 Process Management
1.7 Memory Management
1.8 Storage Management
  1.8.1 File-System Management
  1.8.2 Mass-Storage Management
  1.8.3 Caching
  1.8.4 I/O Systems
1.9 Protection and Security
1.10 Distributed Systems
1.11 Special purpose Systems
  1.11.1 Real-Time Embedded Systems
  1.11.2 Multimedia Systems
  1.11.3 Handheld Systems

1.1 WHAT OPERATING SYSTEMS DO

An operating system acts as the bridge between a computer's hardware (CPU, memory, and I/O devices), application programs (e.g., word processors, browsers), and users. It controls and coordinates hardware usage to enable efficient operation and problem-solving. The system consists of hardware, software, and data, with the operating system ensuring their proper utilization. Similar to a government, it provides an environment for programs to perform useful tasks. To better understand its role, we explore the operating system from user and system perspectives.

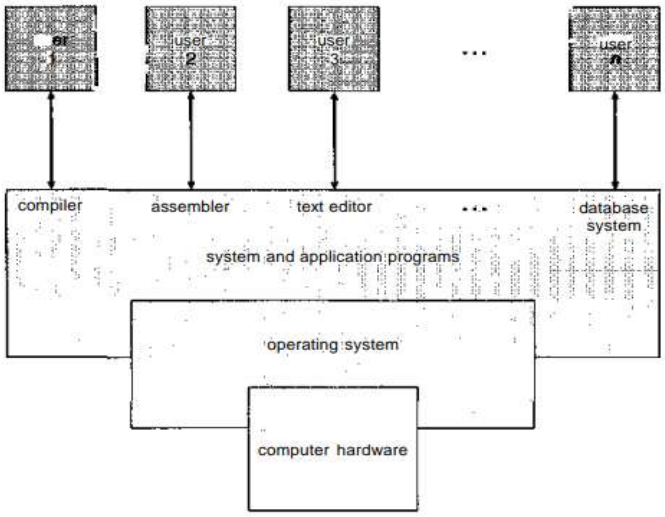


Figure 1.1 Abstract view of the components of a computer system.

1.1.1 User View

The user's view of a computer varies by interface and system type:

- **Personal Computers (PCs):** Designed for single users, these systems prioritize ease of use and performance over resource utilization.
- **Mainframes and Minicomputers:** Multiple users share resources via terminals. Here, the operating system focuses on efficient resource utilization and fair sharing.

- **Networked Workstations:** Users have dedicated resources but also share network resources like servers. The operating system balances usability and resource efficiency.
- **Handheld Devices:** Optimized for individual usability, these systems emphasize performance and battery efficiency, with limited remote operations.
- **Embedded Systems:** Found in devices like appliances and cars, these systems operate autonomously with minimal user interaction, focusing on functionality.

1.1.2 System View

From the computer's perspective, the operating system is closely tied to the hardware, acting as both a **resource allocator** and a **control program**:

- **Resource Allocator:** It manages resources like CPU time, memory, storage, and I/O devices, ensuring efficient and fair allocation to programs and users, especially in multi-user systems.
- **Control Program:** It oversees the execution of user programs, prevents errors, and manages I/O devices to ensure proper operation.

1.1.3 Defining Operating Systems

Defining an operating system is complex, as no universally accepted definition exists. At its core, an operating system facilitates the execution of user programs by managing common operations like I/O control and resource allocation, bridging hardware and application programs to create a usable computing system.

The definition varies:

- It may include all software shipped with the system, ranging from minimal setups to expansive systems with graphical interfaces.
- A widely accepted view is that the operating system consists of the core program always running (the **kernel**), with other components classified as system or application programs.

The scope of an operating system can be controversial. For instance, in 1998, Microsoft faced legal issues for integrating a web browser into its operating system, allegedly stifling competition by bundling excessive functionality.

1.2 COMPUTER SYSTEM ORGANIZATION

To understand how computer systems work, we first need a basic understanding of their structure. This section focuses on computer-system organization and can be skipped if you're already familiar with these concepts.

1.2.1 Computer-System Operation

A modern general-purpose computer system includes one or more CPUs and device controllers connected via a common bus, sharing memory managed by a memory controller for synchronized access.

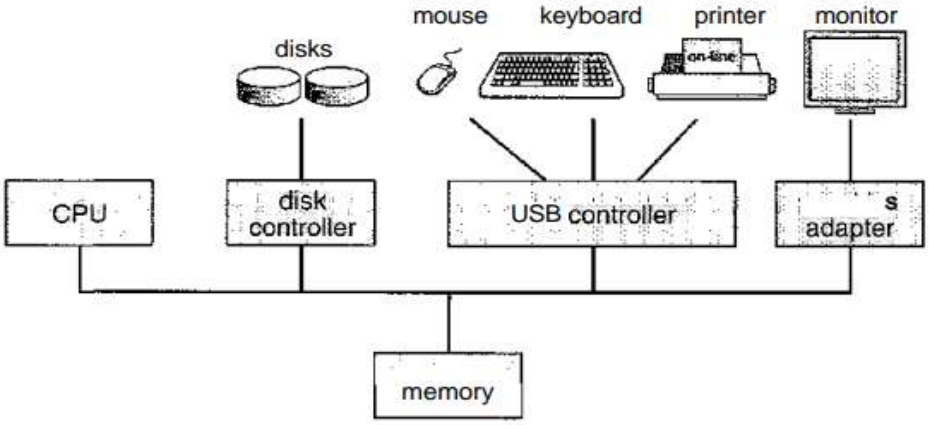


Figure 1.2 A modern computer system

When powered on or rebooted, the system runs a **bootstrap program** stored in firmware (ROM or EEPROM). This program initializes hardware, loads the operating system kernel into memory, and starts the first process (e.g., "init").

Interrupts manage events triggered by hardware or software. Hardware sends signals via the system bus, while software uses system calls. On an interrupt, the CPU halts current operations, executes the corresponding interrupt service routine, and then resumes its tasks. Interrupts are crucial for efficient system operation, with each computer having a unique mechanism for handling them.

To handle interrupts efficiently, systems use an **interrupt vector**, a table of pointers stored in low memory, linking each device to its specific interrupt service routine. Upon an interrupt, the routine is called directly via this table, avoiding delays.

The interrupt architecture also saves the address of the interrupted instruction, often on the system stack in modern designs. If the routine modifies the processor state, it must save and restore the state before returning. Once the interrupt is handled, the return address is loaded into the program counter, and the interrupted process resumes seamlessly.

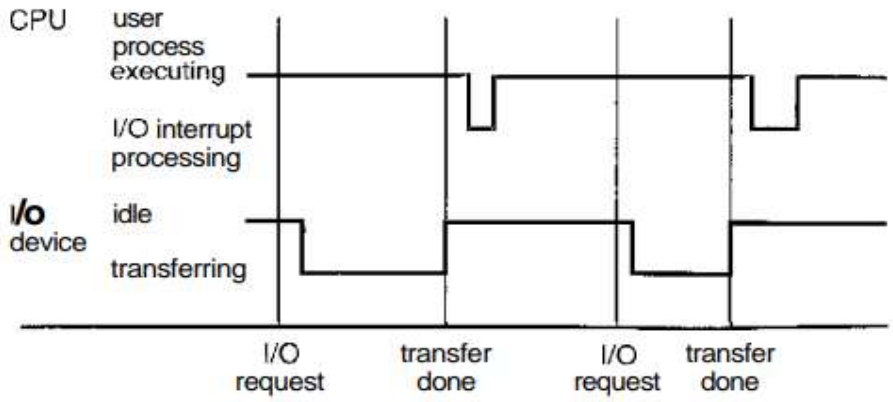


Figure 1.3 Interrupt time line for a single process doing output

1.2.2 Storage Structure

Programs must reside in **main memory (RAM)** to execute, as the CPU can directly access it. Main memory, typically made of **DRAM**, stores data in memory words with unique addresses, accessed through **load** (to CPU registers) and **store** (to memory) instructions.

In a von Neumann architecture, the CPU fetches, decodes, and executes instructions from memory, storing results back into memory. The memory only processes address streams without differentiating between instructions and data.

Programs and data cannot reside permanently in main memory due to two reasons:

1. **Limited capacity**—main memory is too small to hold all programs and data.
2. **Volatility**—memory contents are lost when power is off.

Most computer systems include **secondary storage** to complement main memory, providing permanent storage for large data volumes. The most common device is the **magnetic disk**, which stores programs and data, serving as a source and destination during processing. Proper disk management is critical for system performance.

Beyond disks, storage systems include **cache memory, CD-ROMs, magnetic tapes**, and others. These systems differ in **speed, cost, size, and volatility**.

Storage is organized in a **hierarchy** based on speed and cost: higher levels are faster but more expensive, while lower levels are slower but cheaper. This hierarchy balances cost and access time, with obsolete storage like paper tape replaced by faster, cheaper alternatives like magnetic tape and semiconductor memory.

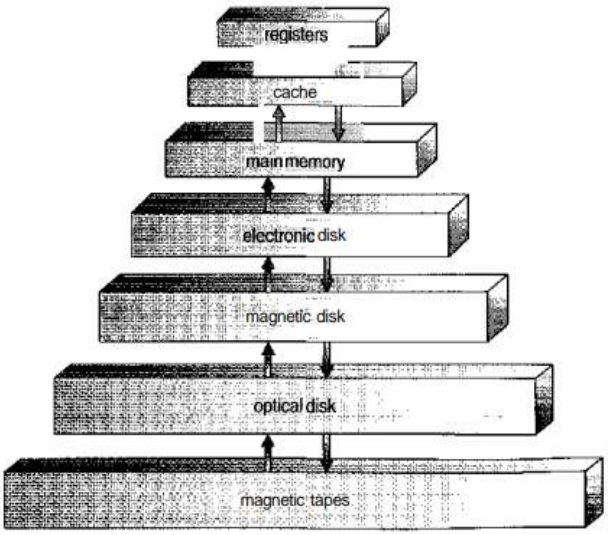


Figure 1.4 Storage-device hierarchy

Storage systems can be either **volatile** or **nonvolatile**. Volatile storage loses its data when power is lost, while nonvolatile storage retains data without power. In the storage hierarchy, systems above the electronic disk are volatile, and those below are nonvolatile.

Electronic disks are typically volatile, using DRAM, but may include a hidden magnetic disk and battery for backup. If power is lost, data is copied from DRAM to the magnetic disk and restored when power returns. **Flash memory** is nonvolatile, slower than DRAM, and commonly used in cameras, PDAs, and as removable storage. **NVRAM** combines DRAM speed with battery backup, retaining data for a limited time without power.

Designing a memory system requires balancing expensive, fast memory with affordable, nonvolatile memory, using caches to improve performance where access-time differences exist.

1.2.3 I/O Structure

In a computer system, I/O devices are managed by device controllers connected through a common bus. Each controller handles specific devices and maintains a local buffer for data transfer. Operating systems use device drivers to interact with controllers, providing a uniform interface.

I/O operations start when the device driver loads registers in the controller, which then begins data transfer. After completion, the controller notifies the driver via an interrupt, which either returns the data or status information.

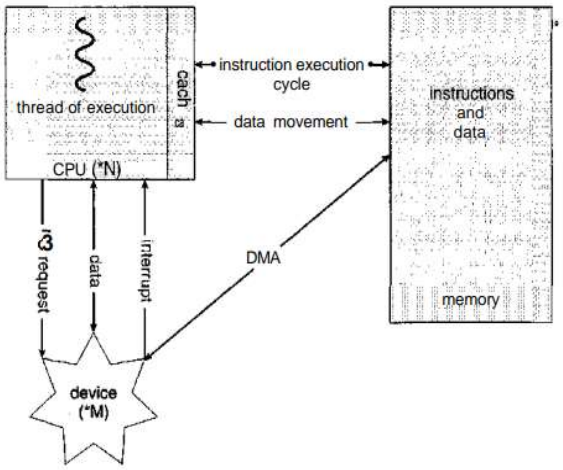


Figure 1.5 How a modern computer system works.

For bulk data transfer, such as disk I/O, **Direct Memory Access (DMA)** is used. DMA allows the device controller to transfer data directly to or from memory without CPU involvement, reducing overhead. Only one interrupt is generated per block transfer, making it more efficient.

High-end systems may use aswitch architecture instead of a shared bus, allowing concurrent communication between components and further improving DMA efficiency.

1.3 Computer-System Architecture

In Section 1.2, we discussed the general structure of a typical computer system. These systems can be organized in various ways, primarily categorized by the number of general-purpose processors used.

1.3.1 Single-Processor Systems

Most systems use a single processor, but these can vary greatly, from PDAs to mainframes. A single-processor system has one main CPU that executes general-purpose instructions, including user processes. However, special-purpose processors, such as disk, keyboard, and graphics controllers, are also commonly used. These processors run limited instruction sets and don’t execute user processes. They are either managed by the operating system or work autonomously, relieving the main CPU from specific tasks. Despite these special processors, a system remains single-processor if it has only one general-purpose CPU.

1.3.2 Multiprocessor Systems

Multiprocessor systems, which have two or more processors in close communication, offer three main advantages:

1. **Increased throughput**: More processors can speed up tasks, but the gain is less than proportional due to overhead and resource contention.
2. **Economy of scale**: Multiprocessors are cost-effective since they share peripherals, storage, and power, reducing costs compared to multiple single-processor systems.
3. **Increased reliability**: If one processor fails, others can take over its tasks, minimizing system downtime and improving overall reliability.

Increased reliability is crucial in many systems, and **graceful degradation** allows a system to continue operating with reduced performance when some hardware fails. **Fault tolerance** goes further, ensuring operation even if any single component fails. For example, the HP NonStop system uses duplicate CPUs that compare results, halting and recovering from faults if necessary, though it's costly.

Multiprocessor systems can be **asymmetric** or **symmetric**. In **asymmetric multiprocessing (AMP)**, one master processor controls the system, and other processors perform predefined tasks. In **symmetric multiprocessing (SMP)**, all processors are equal, performing tasks within the OS, allowing for better performance and flexibility. Most modern systems, including Windows, Linux, and Solaris, use SMP.

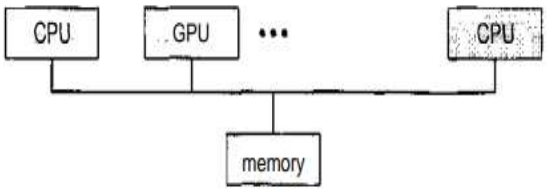


Figure 1.6 Symmetric multiprocessing architecture

Recent trends include **multi-core CPUs**, which combine multiple processors on a single chip, and **blade servers**, where independent processor boards with their own OS run in a shared chassis. Blade servers blur the lines between multiprocessor systems and can have multiple processors on a single board.

1.3.3 Clustered Systems

A **clustered system** is a type of multiple-CPU system where two or more individual systems are linked together, typically via a local-area network (LAN) or a faster connection like InfiniBand. These systems share storage and provide high availability, ensuring service continues even if a system fails. Cluster software monitors nodes, and if one fails, another can take over its tasks with minimal disruption to users.

Clustering can be **asymmetric**, with one machine in standby mode, or **symmetric**, where all nodes run applications and monitor each other. Symmetric

clustering is more efficient. Other types include **parallel clusters**, where multiple systems access shared data, and **WAN clustering**, which extends clusters over large distances. Technologies like **Storage Area Networks (SANs)** enable shared storage, improving performance and reliability by allowing any node to take over tasks if another fails.

1.4 OPERATING-SYSTEM STRUCTURE

An **operating system (OS)** provides the environment for executing programs and managing system resources. One key function of an OS is **multiprogramming**, which increases CPU utilization by keeping multiple jobs in memory and switching between them when one job waits for I/O operations. This ensures the CPU is never idle as long as there are jobs to execute.

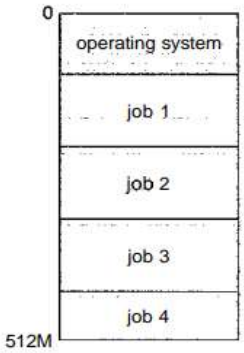


Figure 1.7 Memory layout for a multiprogramming system.

An extension of multiprogramming is **time-sharing (or multitasking)**, where the CPU switches rapidly between jobs, allowing users to interact with programs while they run. In a time-shared system, multiple users can access the computer simultaneously, with each user appearing to have the system dedicated to them. The system responds quickly to user input, typically within a second, providing an interactive experience.

A **time-shared operating system** allows multiple users to share a single computer by rapidly switching between programs. Each user has a separate program in memory, known as a **process**. Since interactive I/O (like user inputs) can be slow, the CPU switches to other user programs to avoid idling.

Programs are initially stored on the disk in a **job pool** and are loaded into memory as needed. This requires **memory management** and **CPU scheduling**, which are covered in the respective chapters. **Virtual memory** allows programs to run even if they don't completely fit in physical memory, enabling larger applications. Time-sharing systems also manage disk storage through a **file system** and provide mechanisms for job synchronization, communication, and protection of resources.

1.5 OPERATING-SYSTEM OPERATIONS

Modern operating systems are interrupt-driven, meaning they remain idle until an interrupt or trap occurs. A trap is a software-generated interrupt caused by errors or specific user requests. The operating system responds to interrupts with an interrupt service routine. To prevent one program's errors from affecting others, the operating system ensures that each program is isolated, protecting the system from malicious or incorrect behavior. This isolation ensures that a bug in one program doesn't disrupt other processes or the operating system.

1.5.1 Dual-Mode Operation

To ensure proper execution, operating systems differentiate between user code and OS code using hardware support for different execution modes: user mode and kernel mode. A mode bit indicates the current mode (0 for kernel, 1 for user). The system starts in kernel mode during boot, switches to user mode when running applications, and returns to kernel mode for system calls or interrupts. This dual-mode operation protects the OS and user programs by restricting privileged instructions (like I/O control and interrupt management) to kernel mode. Any attempt to execute privileged instructions in user mode triggers a trap to the OS.

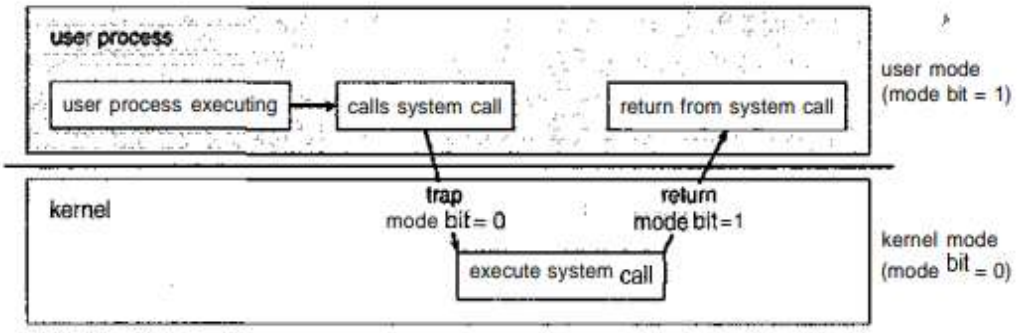


Figure 1.8 Transition from user to kernel mode.

System calls allow user programs to request services from the operating system by invoking a trap to a specific location in the interrupt vector. This triggers a software interrupt, switching the system to kernel mode. The OS examines the system call, checks parameters, executes the request, and returns control to the user program. Without dual-mode operation, older systems like MS-DOS lacked protection, allowing errors to overwrite the OS or cause issues with multiple programs accessing devices simultaneously. Modern CPUs, like Intel's Pentium, support dual-mode, enhancing protection. Hardware errors are detected, leading to traps that transfer control to the OS, which handles errors by terminating programs and providing error messages and memory dumps for troubleshooting.

1.5.2 Timer

To ensure the operating system maintains control over the CPU, a timer can be used to prevent user programs from getting stuck or not returning control to the OS. The timer is set to interrupt after a specified period, either fixed or variable. The OS configures a counter that decrements with each clock tick, and when it reaches 0, an interrupt occurs. This ensures the OS can regain control if a program exceeds its allowed runtime. For example, a program with a 7-minute limit would have its counter initialized to 420, and the OS terminates the program if the counter reaches 0.

1.6 PROCESS MANAGEMENT

A process is a program in execution, requiring resources like CPU time, memory, files, and I/O devices to perform its task. While a program is a passive entity, a process is active. A single-threaded process follows a sequential execution, with one instruction executed at a time. Multiple processes can be associated with the same program, but they are distinct execution sequences. A multithreaded process has multiple program counters, each for a different thread. The operating system allocates resources to processes, provides initialization data, and reclaims resources when the process terminates.

A process is the basic unit of work in a system, comprising operating system and user processes, which can execute concurrently. The operating system manages processes by creating and deleting them, suspending and resuming execution, and providing mechanisms for synchronization, communication, and deadlock handling.

1.7 MEMORY MANAGEMENT

Main memory is a central component in modern computers, storing data and instructions for CPU and I/O device access. Programs must be loaded into memory, mapped to absolute addresses, and executed by accessing instructions and data stored there. To enhance CPU utilization and responsiveness, multiple programs are kept in memory, necessitating memory management.

The operating system handles memory management by:

- Tracking memory usage and users.
- Deciding which processes and data to load or remove.
- Allocating and deallocating memory as needed.

1.8 STORAGE MANAGEMENT

The operating system simplifies storage for users by providing a logical view through files, abstracting the physical properties of storage devices. It maps files to physical media and manages access via storage devices.

1.8.1 File-System Management

File management is a key function of an operating system, handling storage on media like magnetic disks, optical disks, and tapes, each with unique characteristics. A file is a collection of related data, such as programs or information, and can vary in format and structure. Files are organized into directories for easier access, and access control ensures proper usage. The operating system manages file-related tasks like:

- Creating/deleting files and directories
- Manipulating files/directories
- Mapping files to storage
- Backing up files to stable media.

1.8.2 Mass-Storage Management

Secondary storage is essential as main memory is limited and volatile. Disks serve as the primary on-line storage medium for programs and data, requiring efficient management. The operating system handles:

- Free-space management
- Storage allocation
- Disk scheduling

Efficient disk usage is critical for system performance, as it impacts overall speed. For slower, cost-effective tertiary storage like magnetic tapes, CDs, and DVDs, typically used for backups or archival, the OS may manage tasks such as:

- Mounting/unmounting media
- Allocating devices
- Migrating data between secondary and tertiary storage.

1.8.3 Caching

Caching improves system performance by temporarily storing frequently used data in faster storage (the cache). When data is needed, the cache is checked first, and if the data is not there, it is fetched from the source and stored in the cache for future use.

Registers provide a high-speed cache for main memory, and hardware-based caches, like instruction and data caches, speed up CPU operations. Cache management, including choosing cache size and replacement policies, is crucial for performance.

Main memory acts as a fast cache for secondary storage, and file-system data may be cached in memory. For efficient storage, magnetic-disk data is backed up to

magnetic tapes or removable disks, and some systems archive old data to tertiary storage like tape jukeboxes to reduce costs.

Level	1	2	3	4
Name	registers	cache	raaJft memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 –10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operatingsystem	operating system
Backed by	cache	main memory	disk	CD or tape

Figure 1.9 Performance of various levels of storage

Data movement between storage levels can be explicit or implicit, depending on the hardware and operating system. For example, data transfer between cache, CPU, and registers is usually handled by hardware, while data transfer between disk and memory is managed by the operating system.

In a hierarchical storage system, the same data can exist at multiple levels. For instance, when an integer is incremented, it may first reside on disk, then in memory, cache, and an internal register. In a multitasking environment, it's essential to ensure that processes access the most recent value of the data.

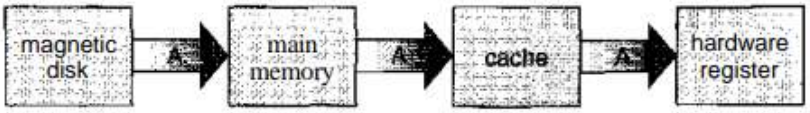


Figure 1.10 Migration of integer A from disk to register

In multiprocessor systems, where each CPU has its local cache, cache coherency must be maintained to ensure that updates in one cache are reflected across all caches. In a distributed environment, multiple copies of a file may exist on different machines, and updates to one replica must be propagated to all other replicas

1.8.4 I/O Systems

An operating system hides hardware peculiarities from users, and in UNIX, this is done by the I/O subsystem. The I/O subsystem has three main components:

- Memory-management, including buffering, caching, and spooling

- A general device-driver interface
- Specific device drivers

Device drivers handle the unique characteristics of each hardware device. The I/O subsystem uses interrupt handlers and device drivers for efficient data transfer and I/O management

1.9 PROTECTION AND SECURITY

In a multi-user computer system, access to resources like files, memory, and CPU must be controlled to ensure that only authorized processes can access them. This is achieved through protection mechanisms that regulate access and enforce controls. Protection helps improve reliability by detecting errors at subsystem interfaces and preventing unauthorized access.

However, protection alone is not enough; security is also needed to defend against internal and external attacks, such as viruses, identity theft, and denial-of-service attacks. Security features are increasingly important in operating systems and are a growing area of research.

Operating systems distinguish users using unique identifiers (user IDs), which are linked to their processes and threads. Additionally, groups of users can be defined for specific permissions, such as file access in UNIX. Systems may also allow privilege escalation for users to gain extra permissions, like access to restricted devices, through methods like the setuid attribute in UNIX.

1.10 DISTRIBUTED SYSTEMS

A distributed system is a network of separate, potentially diverse computer systems that share resources to improve speed, functionality, data availability, and reliability. Access to these resources can be handled either through file access protocols or by directly invoking network functions. The choice of protocols, such as FTP, NFS, or proprietary systems, influences the system's efficiency and popularity.

Networks, essential to distributed systems, vary in protocols, distances between nodes, and transport media. Common network protocols like TCP/IP are supported by most operating systems, while others may use proprietary protocols. Networks are classified by distance: Local Area Networks (LAN), Wide Area Networks (WAN), and Metropolitan Area Networks (MAN), with newer technologies such as Bluetooth and 802.11 offering wireless connectivity.

Network media include copper wires, fiber optics, and wireless technologies like satellite or infrared communication. Operating systems manage network connections, with network operating systems providing features like file sharing and

communication between different processes. In contrast, distributed operating systems create an environment where different systems appear to operate as a single cohesive system.

1.11 SPECIAL-PURPOSE SYSTEMS

The discussion so far has centered on general-purpose computer systems, but there are other types of computer systems designed for more specific tasks, with a focus on limited computation domains.

1.11.1 Real-Time Embedded Systems

Embedded systems are specialized computers found in devices like cars, robots, and household appliances. They typically run simple operating systems with limited features and are focused on specific tasks, often without a user interface. These systems vary in complexity, from general-purpose computers running specialized applications to devices with custom operating systems or application-specific circuits. The use of embedded systems is growing, including in smart homes where devices can be controlled remotely.

Most embedded systems use real-time operating systems, which must meet strict time constraints. These systems are critical in applications like medical imaging, industrial control, and automotive systems, where delays could cause failure. Unlike time-sharing systems, which prioritize fast responses, real-time systems must deliver results within set time limits to function correctly.

1.11.2 Multimedia Systems

Operating systems traditionally handle data like text files and documents, but with the rise of multimedia, they now also manage audio and video data. Multimedia applications, such as MP3s, DVD movies, video conferencing, and live webcasts, require timely delivery, such as streaming 30 video frames per second. These applications can combine audio and video, and are not limited to desktop PCs; they are increasingly used on mobile devices like PDAs and cell phones. For example, stock quotes can be delivered in real-time to a trader’s PDA.

1.11.3 Handheld Systems

Handheld systems, such as PDAs and smartphones, use special-purpose embedded operating systems and face challenges due to their small size. These devices typically have limited memory (512 KB to 128 MB), slower processors to conserve battery, and small screens for display. The operating system must manage memory efficiently, and applications must be optimized for limited processor power.

Input is limited to small keyboards, handwriting recognition, or touchscreens, while output is constrained by small displays. Many handheld devices use wireless technologies like Bluetooth or 802.11 for internet access, though some rely on syncing with PCs for data transfer. Despite their limitations, PDAs offer convenience and portability, with expanding uses as they integrate features like cameras and MP3 players.

1.12 COMPUTING ENVIRONMENTS

We've covered an overview of computer-system organization and key operating-system components, and now conclude with a brief look at how these elements are utilized in different computing environments.

1.12.1 Traditional Computing

As computing evolves, the boundaries between traditional environments are blurring. In the past, office environments consisted of PCs connected to a network with servers for file and print services. Remote access was limited, and portability was achieved with laptops or mainframe terminals.

Today, web technologies have expanded access, with companies offering portals for internal server access. Network computers act as terminals with web-based computing, and handheld devices synchronize with PCs for portability. Wireless access allows PDAs to connect to company portals and other web resources.

At home, high-speed internet connections have become more affordable, enabling users to host web pages, run networks, and secure their systems with firewalls.

In the past, computing resources were scarce, and systems operated in batch or interactive modes. Today, time-sharing systems are less common, but scheduling techniques are still used to manage processes, often for a single user or system processes, enabling multitasking on workstations and servers.

1.12.2 Client-Server Computing

As PCs have become faster, more powerful, and affordable, centralized systems have been replaced by PCs. User-interface functionality, once handled by centralized systems, is now managed by PCs. This shift has led to the rise of client-server systems, where servers handle requests from client systems.

Server systems can be divided into:

- **Compute servers**: Handle client requests to perform actions (e.g., reading data) and return results, such as a database server responding to data requests.

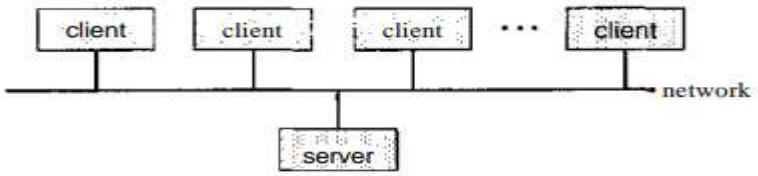


Figure 1.11 General structure of a client-server system.

- **File servers**: Provide a file-system interface for clients to manage files (e.g., a web server delivering files to web browsers).

1.12.3 Peer-to-Peer Computing

In a **peer-to-peer (P2P)** system, all nodes are considered peers, meaning each can act as both a client and a server, depending on the role it plays (requesting or providing services). Unlike client-server systems where the server is a bottleneck, P2P systems distribute services across multiple nodes, avoiding this issue.

To participate in a P2P system, a node must join the network and can then provide and request services. Service discovery is done in two ways:

1. **Centralized lookup**: Nodes register services with a central server, which clients contact to find services.
2. **Broadcasting**: Clients broadcast a request, and nodes that provide the service respond.

P2P networks gained popularity in the late 1990s with file-sharing systems like **Napster** (centralized lookup) and **Gnutella** (broadcasting requests). Despite legal concerns around copyright issues, P2P technology continues to have potential for services like file exchange, searching, and email.

1.12.4 Web-Based Computing

The **Web** has become ubiquitous, accessible through a variety of devices such as PCs, workstations, handheld PDAs, and cell phones. This shift has emphasized networking, with more devices gaining network access, either wired or wireless, and existing devices benefiting from faster connections.

Web-based computing has introduced new device categories like **load balancers**, which distribute network traffic among servers. Operating systems have evolved from basic web clients like Windows 95 to more advanced systems like **Linux** and **Windows XP**, capable of functioning as both web clients and servers. Overall, the Web has increased device complexity as users demand web-enabled functionality.

1.13 SUMMARY

An operating system (OS) manages hardware, runs applications, and provides a user interface. It handles memory, processes, storage, and security while maximizing CPU use with multiprogramming and timesharing. Systems can be uniprocessor, multiprocessor, or clustered, supporting resource sharing and fault tolerance. Distributed systems connect devices via LANs or WANs. Special-purpose OSes, like real-time and multimedia systems, serve specific needs, while modern OSes integrate web and networking features.

1.14 KEY TERMS

Operating System, Process, Memory, Storage, Management, Protection, Security, Distributed Systems, Caching, Kernel, Clustered systems, Virtual memory, Multiprocessor.

1.15 SELF-ASSESSMENT QUESTIONS

1. Explain the primary functions of an operating system.
2. Describe the organization and architecture of a computer system.
3. Discuss different operating system structures with examples.
4. What is process management? Explain its key components.
5. Explain the role of memory management and storage management in operating systems.
6. Discuss the importance of protection, security, and distributed systems in modern computing.

1.16 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”, Seventh Edition, Wiley.
2. William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3. Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4. Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5. Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Dr. Neelima Guntupalli

CHAPTER 2

OPERATING-SYSTEM STRUCTURES

Aim and objectives:

- Understand the services provided by operating systems to users and processes.
- Compare different structural designs of operating systems.
- Learn how to install and customize operating systems.
- Explore the booting process and system initialization.
- Understand hardware interaction during startup.

Structure:

2.1 OPERATING-SYSTEM SERVICES

An operating system creates an environment for program execution, offering various services to programs and users. While the services differ across operating systems, common types exist to simplify programming tasks and enhance convenience for developers.

Operating systems offer essential services to users and programs:

- **User Interface (UI):** Provides ways for users to interact with the system, such as command-line interfaces (CLI), batch interfaces, or graphical user interfaces (GUI) featuring windows, menus, and pointing devices.
- **Program Execution:** Loads and executes programs, allowing them to terminate normally or with errors.
- **I/O Operations:** Manages input/output operations for files and devices, ensuring efficient and protected access to hardware.
- **File-System Management:** Enables file and directory operations, such as creating, deleting, reading, writing, searching, and managing permissions.
- **Communication:** Facilitates data exchange between processes, either on the same system or across networks, using shared memory or message passing.
- **Error Detection:** Monitors and handles errors in hardware, devices, or user programs to maintain system stability and support debugging.

Operating systems ensure efficient system operation with key functions:

- **Resource Allocation:** Manages and allocates resources like CPU, memory, and I/O devices among multiple users or jobs, using scheduling and allocation routines for optimal performance.
- **Accounting:** Tracks resource usage to support billing, usage statistics, and system improvements.
- **Protection and Security:** Ensures controlled access to system resources, safeguards against unauthorized interference, and requires user authentication. It includes securing external devices and monitoring for unauthorized access to maintain overall system integrity.

2.2 USER OPERATING-SYSTEM INTERLACE

Users can interface with an operating system in two main ways: through a **command-line interface (CLI)** for direct command entry or a **graphical user interface (GUI)** for interaction using visuals like windows and menus.

2.2.1 Command Interpreter

Operating systems handle command interpreters in two ways:

1. **Integrated Command Interpreter:** Some systems include it in the kernel, while others, like Windows XP and UNIX, treat it as a separate program running at login. UNIX and Linux offer multiple interpreters (shells), such as Bourne, C, and Korn shells, with users selecting based on preference.
2. **Command Implementation:**

- In one approach, the interpreter directly executes commands using its built-in code (e.g., MS-DOS).
- Alternatively, as in UNIX, commands are implemented as external system programs. The interpreter loads and executes the program associated with the command (e.g., `rm file.txt` loads and runs the `rm` program). This approach allows easy addition of new commands without modifying the interpreter.

2.2.2 Graphical User Interfaces

A **Graphical User Interface (GUI)** offers a user-friendly way to interact with the operating system using a mouse-driven window and menu system. Users can click icons representing files, programs, and functions on a virtual desktop to perform actions like opening programs or selecting files.

GUIs originated from research at Xerox PARC in the 1970s, with the first GUI appearing on the Xerox Alto in 1973. They gained popularity with the Apple Macintosh in the 1980s, evolving over time with features like the Aqua interface in Mac OS X. Microsoft introduced its GUI with Windows 1.0, built on MS-DOS, later enhancing its functionality and appearance in subsequent versions, including Windows Explorer.

UNIX systems traditionally rely on **command-line interfaces (CLI)** but also support graphical interfaces like **CDE** and **X-Windows** in commercial versions (e.g., Solaris, AIX). Open-source projects like **KDE** and **GNOME** provide advanced GUIs for Linux and UNIX, available under open-source licenses.

The choice between CLI and GUI depends on user preference. UNIX users often favor CLI for its powerful shell capabilities, while Windows users primarily use GUIs. Historically, Mac OS only offered a GUI, but Mac OS X introduced both a GUI (Aqua) and CLI due to its UNIX-based kernel.

User interfaces vary across systems and are typically independent of the operating system's internal structure. This book focuses on providing adequate services to programs, treating user and system programs equally.

2.3 System Calls

System calls provide an interface to the operating system's services, typically written in C/C++ or assembly for hardware-level tasks.

For example, a program copying data between two files uses system calls to:

1. Obtain file names (via prompts or GUI menus).
2. Open the input file and create the output file.
3. Handle errors, like missing files or access issues, by printing messages or terminating the program.

Interactive systems may prompt users to resolve conflicts (e.g., overwriting an existing file) using additional system calls for messages and inputs. Each operation—

reading, writing, opening, or closing files—requires specific system calls, demonstrating their integral role in program execution.

Once the input and output files are set up, a loop reads from the input file and writes to the output file, using system calls for each operation. These calls also handle errors, such as reaching the end of the file, hardware failures, or issues like insufficient disk space.

After copying, the program closes the files, writes a completion message, and terminates using system calls. Even simple programs involve frequent system calls, with modern systems executing thousands per second.

Programmers typically use an Application Programming Interface (API) to simplify access to system calls. Popular APIs include the Win32 API (Windows), POSIX API (UNIX/Linux/Mac OS X), and Java API (for Java Virtual Machine). These APIs define functions, parameters, and return values for developers.

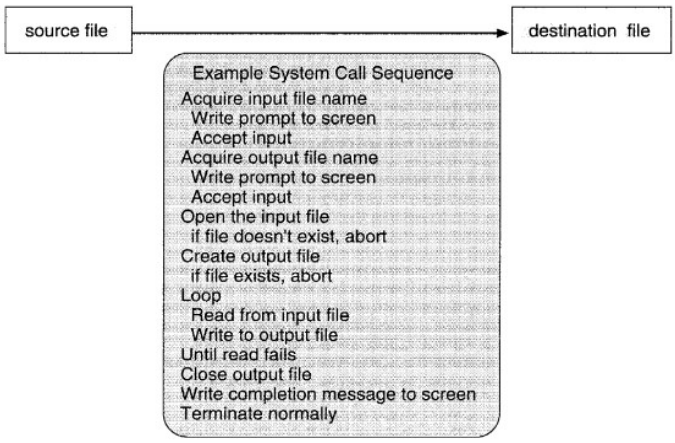


Figure 2.1 Example of how system calls are used

EXAMPLE OF STANDARD API

As an example of a standard API, consider the ReadFile() function in the Win32 API—a function for reading from a file. The API for this function appears in Figure 2.2.

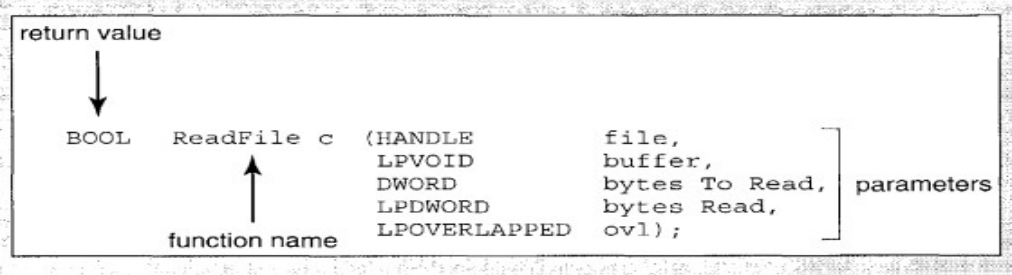


Figure2.2 The API for the ReadFile() function

A description of the parameters passed to ReadFile() is as follows:

- HANDLE file—the file to be read.
- LPVOID buffer—a buffer where the data will be read into and written from.
- DWORD bytesToRead—the number of bytes to be read into the buffer.
- LPDWORD bytesRead—the number of bytes read during the last read.
- LPOVERLAPPED ovl—indicates if overlapped I/O is being used.

System-call names in this text are generic, as each operating system uses its own naming conventions. APIs, like the Win32 API or POSIX API, simplify programming by abstracting system calls. For example, the Win32 CreateProcess() API calls the underlying Windows kernel system call NTCreateProcess().

Using an API instead of directly invoking system calls offers portability, as programs written for an API can run on systems supporting the same API, though architectural differences may still pose challenges. APIs also simplify complex system calls, making them easier for programmers.

A runtime support system, included in programming language libraries, links API calls to system calls. The system-call interface maps API calls to the appropriate system calls using a table of call numbers, invoking them in the kernel, and returning their status and results. This abstraction hides system-call details from programmers, who only need to follow the API's specifications. The relationship between the API, system-call interface, and operating system simplifies user application interactions.

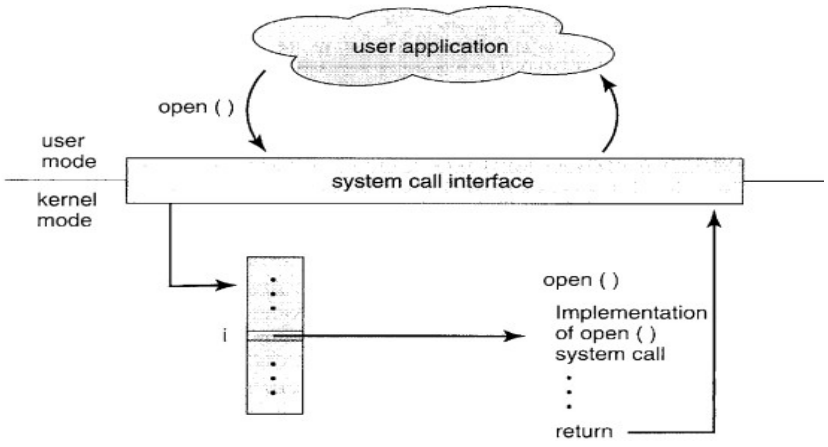


Figure 2.3 The handling of a user application invoking the open() system call

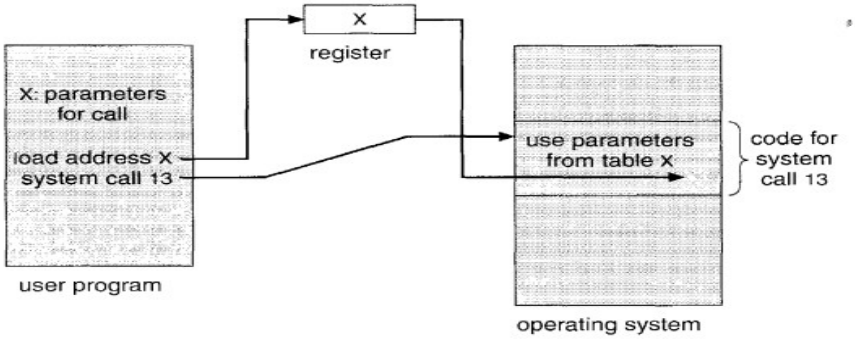


Figure 2.4 Passing of parameters as a table.

System calls vary by operating system and may require parameters like the source file or device, memory buffer address, and length.

There are three common methods to pass parameters to the operating system:

1. **Registers:** Parameters are passed directly in CPU registers.
2. **Memory block:** Parameters are stored in a memory block, and its address is passed in a register (used by Linux and Solaris).
3. **Stack:** Parameters are pushed onto the stack by the program and popped by the operating system.

The block and stack methods are preferred for flexibility, as they allow unlimited parameter size and number.

2.4 TYPES OF SYSTEM CALLS

System calls can be broadly categorized into five types: **process control**, **file manipulation**, **device manipulation**, **information maintenance**, and **communications**. These system calls support various operating system functions, which are detailed in later chapters. Figure 2.5 provides an overview of these system call types.

2.4.1 Process Control

A running program can terminate normally (end) or abnormally (abort). In case of an abnormal termination or error, the system may generate a memory dump and error message for debugging. The operating system then transfers control to the command interpreter. In interactive systems, the interpreter processes the next command, while in GUI systems, a pop-up may alert the user. In batch systems, the job is typically terminated, and the next job is executed.

• Process control

- end, abort
- load, execute
- create process, terminate process
- get process attributes, set process attributes
- wait for time

- wait event, signal event
- allocate and free memory

• File management

- create file, delete file
- open, close
- read, write, reposition
- get file attributes, set file attributes

• Device management

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

• Information maintenance

- get time or date, set time or date
- get system data, set system data
- get process, file, or device attributes
- set process, file, or device attributes

• Communications

- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

Figure 2.5 Types of system calls

Some systems use control cards (batch system commands) for error recovery actions. Errors can be categorized with levels, where level 0 indicates normal termination. This helps the command interpreter or subsequent programs decide on the next steps automatically.

A running program may load and execute another program, triggered by user commands, mouse clicks, or batch commands. Control after the new program's termination can either return to the original program (requiring its memory to be saved) or allow both programs to run concurrently, creating a new job or process.

New jobs or processes require management, including setting or modifying attributes like priority and execution time, and may be terminated if no longer needed. This functionality is supported by system calls for creating, managing, or terminating processes.

EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call(s) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program. This is shown in Figure 2.6.

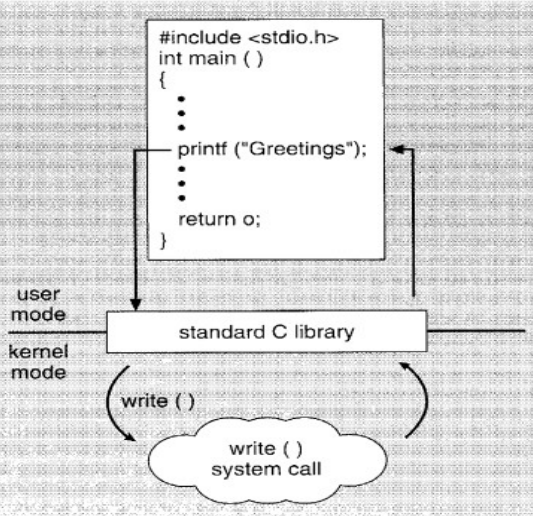


Figure 2.6 C library handling of `write()`.

System calls often handle process coordination, debugging, and performance analysis. Processes may need to wait for specific events (wait event) or time durations (wait time) and signal when those events occur (signal event), which are crucial for managing concurrent processes.

Debugging support includes memory dumps, program tracing (tracking executed instructions), and single-step CPU modes for detailed analysis. Time profiling tracks execution time at various program locations using timer interrupts or tracing, offering insights into program performance.

For example, MS-DOS, a single-tasking system, runs programs by loading them into memory, overwriting most of itself to maximize available memory. The command interpreter resumes after the program terminates, reloads itself, and provides any saved error codes for user or subsequent program use.

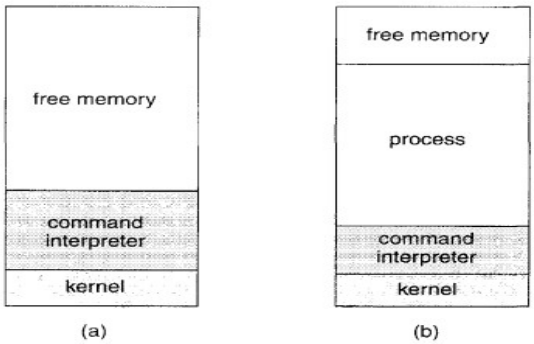


Figure 2.7 MS-DOS execution. (a) At system startup. (b) Running a program.

FreeBSD, a multitasking system, allows users to run a shell that accepts commands and executes programs. Unlike MS-DOS, FreeBSD's shell can continue running while executing other programs. To start a new process, the shell uses the fork() system call to create a process and exec() to load and run a program.

Depending on the command, the shell either waits for the process to finish or runs it in the background. Background processes handle I/O through files or a GUI since the shell controls the keyboard. Users can execute other commands, monitor processes, or adjust priorities concurrently. When a process completes, it uses the exit() system call to terminate, returning a status code to the shell or invoking process.

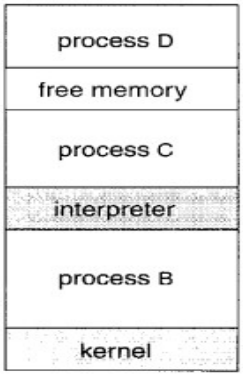


Figure 2.8 FreeBSD running multiple programs.

2.4.2 File Management

The file system, discussed in later chapters, involves several key system calls. These include creating and deleting files, which require the file name and attributes. Once created, files need to be opened for use, with operations like reading, writing, or repositioning (e.g., rewinding).

After usage, files should be closed. For directories, similar operations apply, along with the ability to retrieve and reset file attributes, such as name, type, and protection codes. Additional system calls may support tasks like file moving and copying, and some operating systems offer APIs or system programs to perform these operations.

2.4.3 Device Management

A process requires various resources, such as memory, disk drives, and file access, to execute. If resources are available, they are granted; otherwise, the process must wait. Resources, both physical (e.g., tapes) and virtual (e.g., files), are controlled by the operating system.

For exclusive use, a device must be requested and released once finished, similar to the open and close system calls for files. Many operating systems, like UNIX, treat devices and files similarly, allowing the same system calls for both. Some systems may identify devices using special file names or attributes, making the user interface appear unified despite different underlying system calls.

2.4.4 Information Maintenance

Many system calls are used to transfer information between the user program and the operating system, such as retrieving the current time, system statistics (e.g., number of users, OS version, free memory), or process information. System calls are also used to access and modify process attributes, such as getting and setting process details.

2.4.5 Communication

There are two common models of inter-process communication: message passing and shared memory. In the message-passing model, processes exchange messages either directly or via a mailbox, requiring a connection to be established. Communication happens through system calls like "open connection" and "accept connection." Daemon processes usually wait for connections from clients and exchange messages using "read" and "write" system calls.

In the shared-memory model, processes use system calls like "shared memory create" and "shared memory attach" to access regions of memory owned by other processes. This model allows fast communication but requires careful synchronization to avoid conflicts. Both models are widely used, with message passing being easier for cross-computer communication and shared memory offering faster communication within a computer.

2.5 SYSTEM PROGRAMS

Modern systems include various system programs that provide an environment for program development and execution. These programs can be categorized as follows:

- **File Management**: Programs for creating, deleting, copying, renaming, and manipulating files and directories.
- **Status Information**: Programs that retrieve system status such as date, time, available memory, or performance data, and display it on terminals or in windows.
- **File Modification**: Text editors and commands for creating and modifying file content, and searching or transforming text.
- **Programming Language Support**: Compilers, assemblers, debuggers, and interpreters for languages like C, Java, and PERL.
- **Program Loading and Execution**: Systems for loading and executing compiled programs, including loaders and debugging tools.
- **Communications**: Programs enabling virtual connections for messaging, browsing, email, remote login, and file transfers.

These system programs facilitate program development, management, and communication within the system.

In addition to system programs, operating systems come with utility programs or application software designed for common tasks, such as web browsers, word processors, spreadsheets, database systems, and games.

The user experience is shaped by these applications and system programs rather than the underlying system calls. For example, on a mac OS system, a user might

interact with a GUI or a command-line UNIX shell, both of which use the same system calls but present them in different ways.

2.6 OPERATING-SYSTEM DESIGN AND IMPLEMENTATION

This section addresses challenges in designing and implementing an operating system. While there are no perfect solutions, certain approaches have proven effective.

2.6.1 Design Goals

The first step in designing an operating system is defining goals and specifications, which are influenced by factors like hardware and system type (e.g., batch, time-shared, real-time). Requirements can be categorized into user goals (convenience, reliability, speed) and system goals (flexibility, efficiency, error-free operation).

However, these goals are often vague and interpreted differently. There is no one-size-fits-all solution, as requirements vary widely across systems, like VxWorks for embedded systems versus MVS for IBM mainframes. Designing an OS is a creative task, guided by software engineering principles.

2.6.2 Mechanisms and Policies

A key principle in OS design is separating policy from mechanism. Mechanisms define how tasks are performed, while policies determine what tasks should be done. This separation allows flexibility, as policies may change without requiring changes to underlying mechanisms.

For example, a timer mechanism can be used with different policies on how long it should be set. Microkernel-based OSs take this separation to an extreme, offering basic building blocks that allow users to implement specific policies. In contrast, systems like Windows and macOS combine both policy and mechanism. Policy decisions are crucial in resource allocation, determining what resources to allocate, while mechanisms define how to allocate them.

2.6.3 Implementation

Operating systems were traditionally written in assembly language, but now they are mostly implemented in higher-level languages like C or C++. This shift offers several benefits: faster development, more compact code, easier debugging, and improved portability across hardware. For example, Linux, written in C, runs on various CPUs, while MS-DOS, written in assembly, is limited to Intel CPUs.

While using a higher-level language may result in slightly slower performance and higher storage requirements, modern compilers can optimize code effectively, and today's processors handle complex dependencies efficiently.

Performance improvements in operating systems typically come from better data structures and algorithms rather than hand-written assembly code. Critical routines, like the memory manager and CPU scheduler, can be optimized with

assembly code after identifying bottlenecks through system performance monitoring. This is often done using trace listings that log system events for later analysis.

2.7 OPERATING-SYSTEM STRUCTURE

Modern operating systems are large and complex, requiring careful engineering for proper functionality and ease of modification. A common approach is to divide the system into smaller, well-defined modules, each with clear inputs, outputs, and functions. These modules are then interconnected and integrated into the system's kernel.

2.7.1 Simple Structure

Many commercial operating systems, like MS-DOS, lack well-defined structures. MS-DOS, initially designed to be simple and compact, grew beyond its original scope without modular design. It allows direct access to basic I/O routines, making it vulnerable to system crashes due to faulty programs. MS-DOS was also limited by hardware, as the Intel 8088 lacked dual-mode and hardware protection.

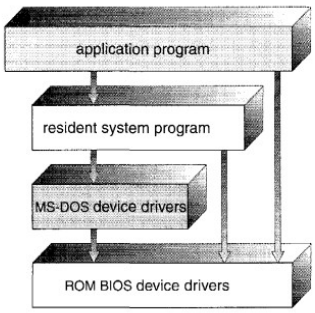


Figure 2.10 MS-DOS layer structure.

Similarly, the original UNIX system had limited structuring, consisting of a kernel and system programs. The kernel, which provides essential OS functions, was initially monolithic and difficult to implement and maintain, especially given the hardware limitations at the time. Over time, UNIX evolved with additional interfaces and device drivers.

2.7.2 Layered Approach

Modular Operating Systems: With proper hardware support, OSes can be modularized into smaller, manageable components, offering better control and flexibility compared to older systems like MS-DOS or UNIX.

Top-Down Design: The system's functionality is divided into components. Information hiding allows programmers to implement low-level routines independently as long as interfaces and tasks remain consistent.

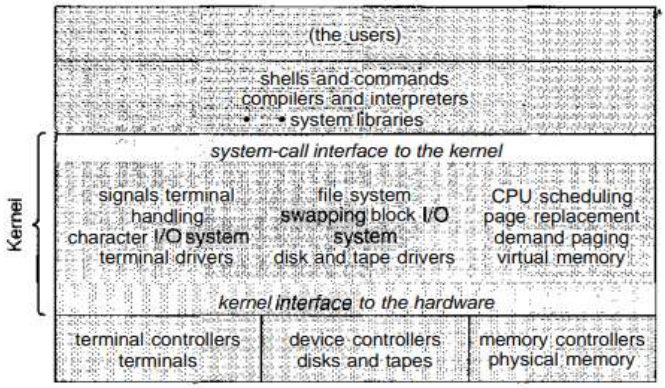


Figure 2.11 UNIX system structure.

Layered Approach:

- The OS is divided into layers (e.g., hardware at the bottom, user interface at the top).
- Each layer provides services to the layer above and relies on the layer below.

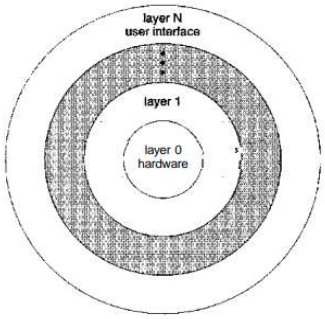


Figure 2.12 A layered operating system.

Advantages of Layering:

- Simplifies construction, debugging, and verification.
- Errors in a layer are easier to isolate since lower layers are pre-debugged.
- Each layer operates abstractly, needing only to know *what* lower-layer operations do, not *how* they are implemented.

Abstraction and Simplification: Layers hide data structures, operations, and hardware details from higher layers, simplifying design and implementation.

Limitations of the Layered Approach:

1. **Difficulty in Defining Layers**:
    - Proper layer definition requires careful planning since each layer depends on lower ones.

- Example: The backing-store driver must be below memory management but may also need to interact with the CPU scheduler, creating conflicts.

2. **Complex Dependencies**:
   - Dependencies can be non-obvious, e.g., in large systems, the CPU scheduler may rely on memory-swapped data, requiring the backing-store driver to be below it.
3. **Efficiency Issues**:
   - Layered systems introduce overhead. A system call must pass through multiple layers (e.g., I/O → memory management → CPU scheduling → hardware), increasing execution time compared to nonlayered systems.
4. **Modern Trends**:
   - To address these issues, newer designs use fewer layers with more functionality, balancing modularity with reduced complexity and overhead.

2.7.3 Microkernels

Concept:

- Microkernels remove nonessential components from the kernel, implementing them as user-space programs, resulting in a smaller kernel.
- Typically, they provide minimal process and memory management along with a communication facility.

Communication:

- Uses **message passing** for interaction between client programs and user-space services. Clients communicate with services indirectly via the microkernel.

Advantages:

- **Extensibility**: New services can be added to user space without modifying the kernel.
- **Portability**: Easier to adapt to different hardware designs.
- **Security and Reliability**: Failures in user-space services do not impact the kernel or other services.

Examples:

- **Mach**: Provides UNIX compatibility by mapping system calls into messages.
- **QNX**: A real-time OS with a microkernel handling message passing, process scheduling, networking, and hardware interrupts.

Challenges:

- **Performance Issues**: Increased overhead from system functions can degrade performance.
- Example: Windows NT started with a microkernel but shifted toward a more monolithic design in later versions (e.g., Windows XP) to improve performance.

2.7.4 Modules

Modular Kernel Design and Hybrid Structures

1. **Modular Kernel Design**:
   - Modern operating systems use **object-oriented programming** to create modular kernels.
   - Core components provide essential services, while additional services are dynamically linked at boot time or runtime.
   - **Dynamically loadable modules** are used in systems like Solaris, Linux, and Mac OS X, allowing flexibility and scalability.

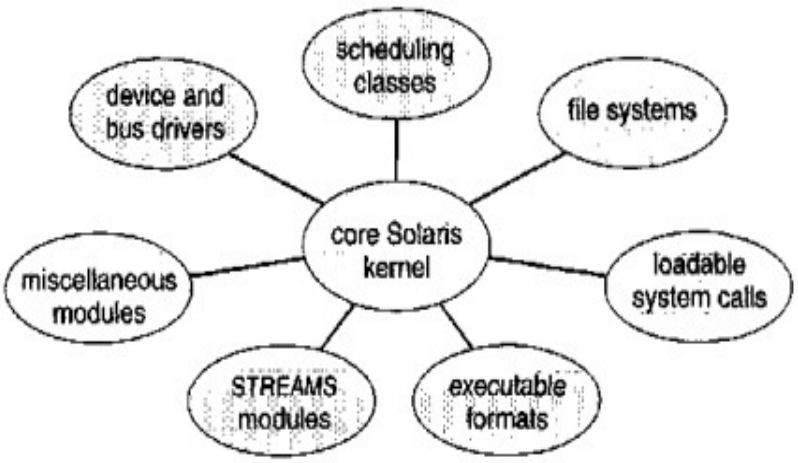


Figure 2.13 Solaris loadable modules

2. **Example: Solaris**:
   - Solaris kernel has core components and supports seven types of loadable modules:
     1. Scheduling classes
     2. File systems
     3. Loadable system calls
     4. Executable formats
     5. STREAMS modules
     6. Miscellaneous
     7. Device and bus drivers
   - Modules can call one another, combining the efficiency of monolithic kernels with the flexibility of microkernels.
3. **Hybrid Structure: Mac OS X (Darwin)**:
   - **Layered design**:
     - Top layers: Application environments and graphical services.
     - Kernel environment: Combines the **Mach microkernel** and **BSD kernel**.

     Mach kernel: Provides memory management, thread scheduling, IPC, and RPC.

     BSD kernel: Offers networking, file systems, POSIX APIs, and Pthreads.

Includes an **I/O kit** for device drivers and kernel extensions.
Applications can utilize both Mach and BSD facilities directly.

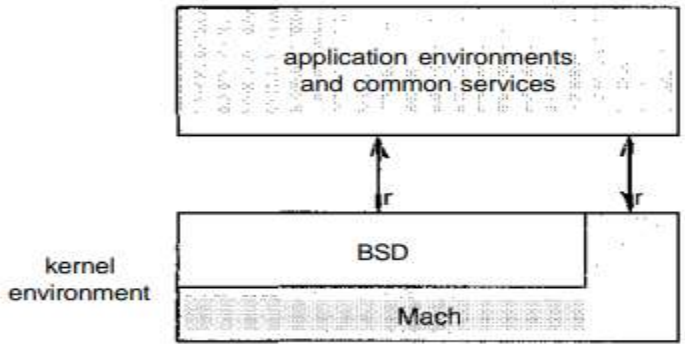


Figure 2.14 The Mac OS X structure.

2.8 Virtual Machines

Concept:

- A **virtual machine (VM)** abstracts a single computer's hardware (CPU, memory, disks, etc.) into multiple independent execution environments.
- Each VM appears to have its own processor and memory, created using CPU scheduling and virtual-memory techniques.
- Unlike processes, VMs replicate the underlying hardware without adding system calls or a file system.

Purpose:

- Enables multiple operating systems or execution environments to share the same hardware.
- Each process is provided with its own virtualized version of the hardware.

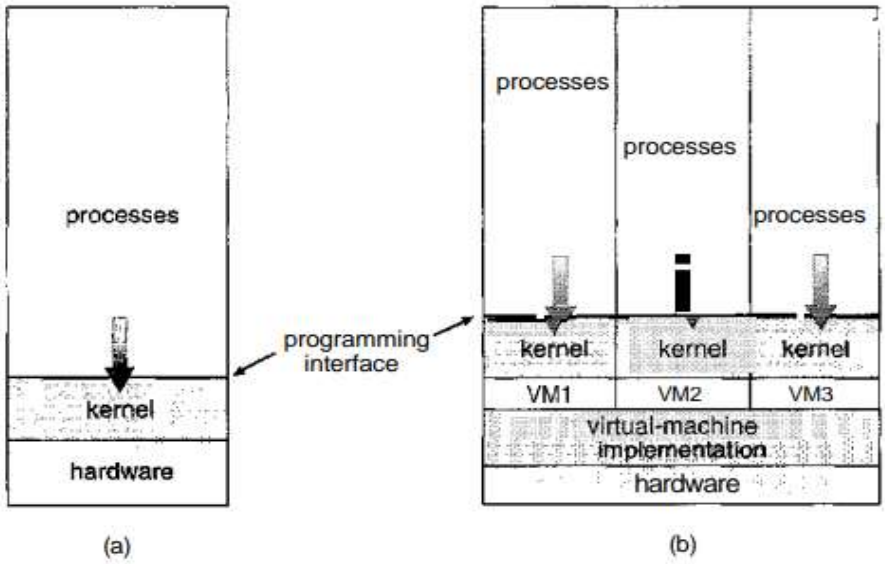


Figure 2.15 System models, (a) Nonvirtual machine, (b) Virtual machine.

Disk Challenges:

- Physical hardware limitations, such as fewer disk drives than virtual machines, are addressed by **virtual disks** (e.g., **minidisks** in IBM's VM system).
- Minidisks allocate portions of physical disk space, ensuring the total size fits within the physical disk capacity.

Applications:

- Users can run any OS or software available on the physical hardware within their VM.
- Example: IBM VM system often runs CMS (a single-user OS) for user interactions.
- VM software manages multiprogramming of VMs but doesn't handle user-support software.

Advantage:

- Simplifies multiuser system design by dividing it into hardware virtualization and user software management.

2.8.1 Implementation

Implementing a virtual machine (VM) is complex due to the need to replicate the underlying machine's behavior. The physical machine operates in user mode and kernel mode, while the VM must simulate these as virtual user and kernel modes, both running in the physical user mode. Actions like system calls or privileged instructions trigger transitions between these modes on both the real and virtual machines.

When a program on a VM makes a system call, control transfers to the virtual-machine monitor. The monitor simulates the system call by adjusting the virtual machine's state and restarts it in virtual kernel mode. The process can be slower due to additional overhead, such as interpreting privileged instructions and sharing CPU time among multiple VMs.

IBM systems optimize this by allowing most instructions to run directly on the hardware, simulating only privileged instructions like I/O operations, which reduces the overall performance impact.

2.8.2 Benefits

Advantages of Virtual Machines

1. **Resource Protection**: Virtual machines are isolated from each other, preventing resource conflicts but not allowing direct resource sharing.

2. **Resource Sharing**: Sharing is possible through software, such as shared minidisks for file access or a virtual network for communication.
3. **Operating System Development**: Virtual machines enable safer OS development by allowing changes to be tested on virtual machines without disrupting the physical system, minimizing downtime.

2.8.3 Examples

Although virtual machines were overlooked for years, they are gaining popularity again for solving system compatibility issues. Modern virtual machines, such as VMware and the Java Virtual Machine, typically run on top of various operating systems, demonstrating that design methods like layers, microkernels, modules, and virtual machines can coexist.

2.8.3.1 VMware

VMware is a commercial application that abstracts Intel 80X86 hardware into isolated virtual machines. It runs on a host OS (Windows or Linux) and enables the host to run multiple guest operating systems as independent virtual machines.

For testing an application on different OSes (Linux, FreeBSD, Windows NT, Windows XP), VMware allows the developer to run all guest operating systems on a single physical machine concurrently, avoiding the need for separate machines or time-consuming reinstallation. Each virtual machine has its own virtual CPU, memory, disk, and network interfaces, with the virtualization layer managing hardware abstraction.

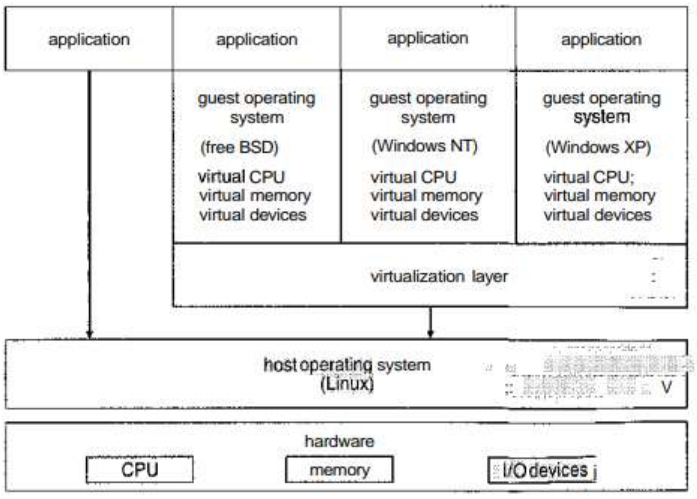


Figure 2.16 VMware architecture

2.8.3.2 The Java Virtual Machine

Java, an object-oriented programming language introduced in 1995, provides a specification for a Java Virtual Machine (JVM) that runs Java programs. Java programs consist of classes, and the compiler produces architecture-neutral bytecode (.class files) that can run on any JVM implementation.

The JVM includes a class loader, which loads compiled bytecode files, and a verifier, which checks that the bytecode is valid and does not cause illegal memory access. After verification, the bytecode is executed by the Java interpreter. The JVM also handles automatic memory management through garbage collection.

The JVM can be implemented in software (on systems like Windows, Linux, or Mac OS X), or in hardware on a chip designed for Java execution. In software, the bytecode is interpreted, but a just-in-time (JIT) compiler can speed up execution by converting bytecode into native machine code. A hardware implementation can directly execute Java bytecode operations, eliminating the need for both an interpreter and JIT compiler.

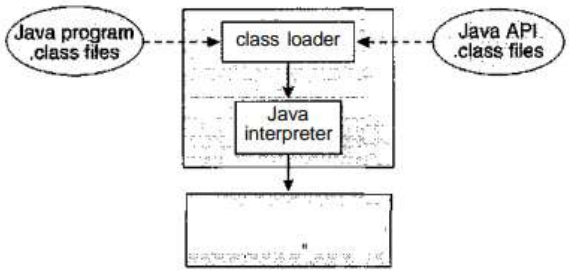


Figure 2.17 The Java virtual machine.

THE .NET FRAMEWORK

The .NET Framework is a platform for software development that includes class libraries and an execution environment, allowing programs to run on any architecture supporting .NET, without worrying about hardware or operating system specifics. The core of the .NET Framework is the Common Language Runtime (CLR), which acts as the .NET virtual machine.

Programs written in languages like C# or VB.NET are compiled into Microsoft Intermediate Language (MS-IL), an architecture-independent language. These compiled files, known as assemblies, contain MS-IL instructions and metadata. When executed, the CLR loads the assemblies into the Application Domain and uses just-in-time (JIT) compilation to convert MS-IL into native code specific to the underlying architecture. Once converted, the native code continues to run on the CPU.

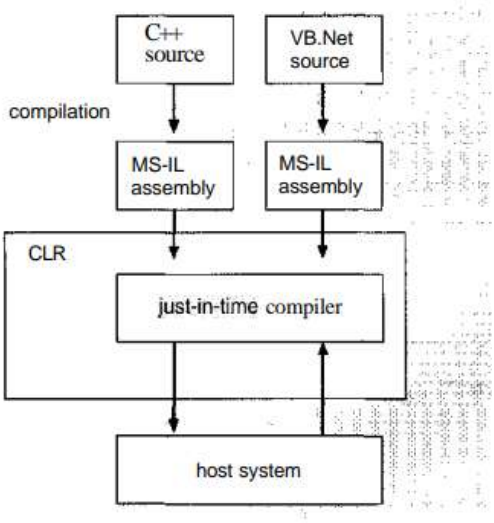


Figure 2.18 Architecture of the CLR for the .NET Framework.

2.9 Operating-System Generation

Operating systems are typically designed to run on a range of machines with varying configurations. To configure the system for a specific site, a process called system generation (SYSGEN) is used. The SYSGEN program gathers information about the machine's hardware configuration, either from a file, operator input, or hardware probing.

The system must determine the following:

- CPU type and options (e.g., extended instruction sets, floating point arithmetic).
- Amount of available memory, which may be determined by probing memory locations.
- Available devices, including their type, model, address, interrupt number, and characteristics.
- Desired operating system options, such as buffer sizes, CPU-scheduling algorithm, and maximum process count.

Once the system configuration information is gathered, it can be used in different ways for system generation.

- **Fully tailored system:** The system administrator modifies the source code, compiling the operating system with specific data declarations, initializations, and constants to create a version tailored to the system.
- **Semi-tailored system:** The system description selects precompiled modules (e.g., device drivers) from a library and links them to form the operating system. This method is faster but results in a more general system.
- **Table-driven system:** All code is part of the system, with selection occurring at execution time based on system tables, offering flexibility with minimal system generation effort.

The key differences lie in the system's size, generality, and ease of modification as hardware changes occur, balanced against the frequency of such changes.

2.10 System Boot

Booting an operating system involves loading the kernel into memory, typically using a bootstrap program or loader. On most systems, this loader first fetches a more complex boot program from disk, which then loads the kernel. Upon startup or reboot, the CPU starts execution from a predefined memory location, where the initial bootstrap program (stored in ROM) is located. The bootstrap program often runs diagnostics, initializes the system, and eventually starts the operating system.

Some systems store the entire operating system in ROM, which is ideal for small systems but limits flexibility since ROM cannot be easily modified. To address this, erasable programmable ROM (EPROM) can be used. However, firmware (ROM) is slower than RAM, and its cost limits the amount available in systems. Some systems store the OS in firmware but load it into RAM for faster execution.

For large operating systems like Windows, Mac OS X, and UNIX, the bootstrap loader is stored in firmware, and the operating system is located on disk. The bootstrap loader performs diagnostics and reads a block of code from the disk (often block zero) into memory to begin the boot process. This code typically loads the rest of the bootstrap program or the operating system itself into memory. A disk with a boot partition is called a boot disk. Once the full bootstrap program is loaded, it locates the operating system kernel, loads it into memory, and starts its execution, at which point the system is considered fully booted.

2.11 SUMMARY

Operating systems provide services through system calls, allowing programs to request functions like process control and file manipulation. Users interact with the OS via command interpreters or shells, which translate commands into system calls. System programs handle common tasks like I/O and status requests.

Designing an OS requires clear goals, modularity, and separation of policies from implementation. Common approaches include layered structures, microkernels, and virtual machines, which improve flexibility and scalability.

Modern OSes are written in higher-level languages for better maintenance and portability. During startup, a bootstrap program initializes the system, progressively loading the operating system into memory for execution.

2.12 KEY TERMS

Command Interpreter, GUI, System Calls, System Programs, Operating System, Virtual Machine, VMware, JVM, CLI, Process control, Modules, Micro kernel, System Boot.

2.14 SELF-ASSESSMENT QUESTIONS

1. What are the key services provided by an operating system?
2. Explain the different types of system calls and their functions.
3. What is the role of system programs in an operating system?
4. Describe the design and implementation process of an operating system.

5. How does the concept of a virtual machine enhance operating system functionality?
6. What is the process of system boot and how does it initialize an operating system?

2.15 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. "Operating System Principles", Seventh Edition, Wiley.
2. William Stallings, "Operating Systems – Internals and Design Principles", Fifth Edition, Pearson Education (2007)
3. Achyut S Godbole, "Operating Systems", Second Edition, TMH (2007).
4. Flynn/McHoes, "Operating Systems", Cengage Learning (2008).
5. Deitel&Deitel, "Operating Systems",Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR
Dr. Neelima Guntupalli

CHAPTER 3

PROCESS CONCEPT

Aim and Objectives:

- Understand the concept of a process as the core unit of computation.
- Explain the role of processes in executing tasks within a system.
- Explore key features of processes, such as scheduling mechanisms.
- Discuss how processes are created and managed in an operating system.
- Describe the process termination and its impact on system resources.

Structure:

3.1 Process Concept

In operating systems, all CPU activities—whether jobs in batch systems, tasks in time-shared systems, or programs in single-user systems like memory management—are collectively referred to as processes. While the term *process* is now preferred, *job* is often used interchangeably due to its historical relevance in operating system terminology, such as in *job scheduling*.

3.1.1 The Process

A process is a program in execution, comprising more than just program code (text section). It includes the program counter, processor registers, process stack (for temporary data like function parameters and local variables), data section (global variables), and a heap (dynamically allocated memory).

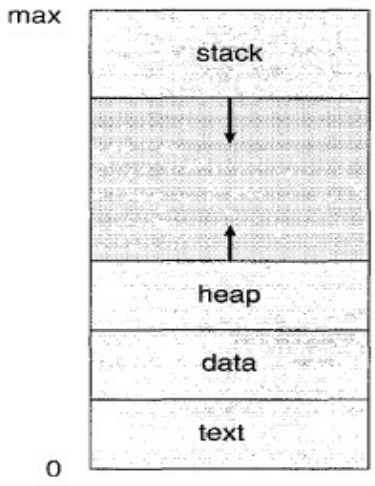


Figure 3.1 Process in memory.

A program is a passive entity (e.g., an executable file), whereas a process is an active entity with allocated resources and execution flow. A program becomes a process when loaded into memory, initiated by actions like double-clicking an icon or entering its name in the command line.

Multiple processes can originate from the same program but are treated as distinct execution sequences, with shared text sections but independent data, stack, and heap sections. For example, multiple users can run separate instances of the same program, and processes can spawn others during execution.

3.1.2 Process State

A process transitions through several states during execution, defined by its current activity:

- **New:** The process is being created.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for an event (e.g., I/O completion).
- **Ready:** The process is waiting to be assigned to a processor.
- **Terminated:** The process has completed execution.

These states are universal but may vary in naming across operating systems. Only one process can run on a processor at a time, while many others may be ready or waiting. The transitions between these states are represented in a state diagram.

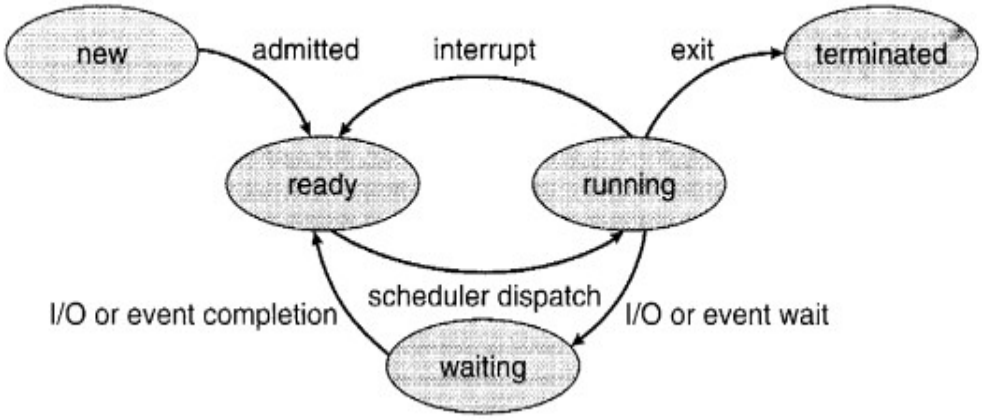


Figure 3.2 Diagram of process state.

3.1.3 Process Control Block

The **Process Control Block (PCB)**, also known as the task control block, represents a process in the operating system and stores process-specific information, including:

- **Process State:** Current state (e.g., new, ready, running, waiting, or terminated).
- **Program Counter:** Address of the next instruction to execute.

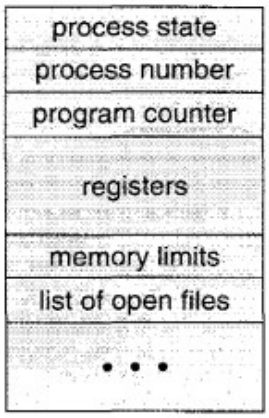


Figure 3.3 Process control block (PCB)

- **CPU Registers:** Includes accumulators, index registers, stack pointers, and other state information saved during interrupts.
- **CPU Scheduling Information:** Contains process priority, scheduling queue pointers, and parameters.
- **Memory Management Information:** Includes base/limit registers, page tables, or segment tables.

- **Accounting Information:** Tracks CPU usage, time limits, process IDs, and other resource usage metrics.
- **I/O Status Information:** Lists allocated I/O devices and open files.

The PCB acts as a repository for all data needed to manage a specific process.

3.1.4 Threads

The traditional process model assumes a single thread of execution, where a process performs one task at a time. For example, a word processor allows typing or running a spell checker, but not both simultaneously. Modern operating systems enhance this concept by enabling multiple threads within a process, allowing concurrent execution of tasks.

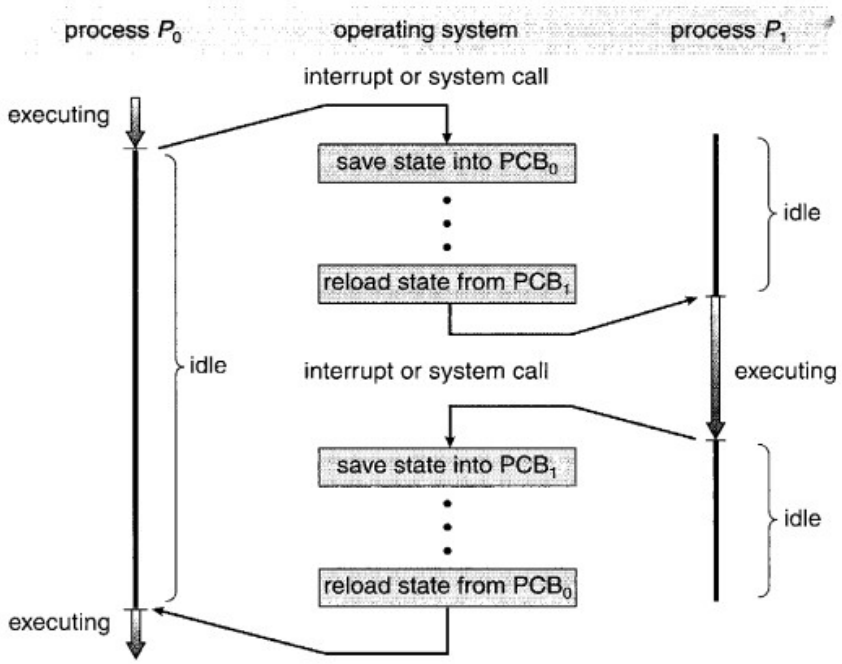


Figure 3.4 Diagram showing CPU switch from process to process.

3.2 Process Scheduling

The goal of **multiprogramming** is to maximize CPU utilization by ensuring a process is always running. **Time sharing** aims to switch the CPU rapidly between processes, allowing user interaction with running programs. The **process scheduler** selects a process for execution from the available ones. On a single-processor system, only one process runs at a time, while others wait for the CPU to become available.

3.2.1 Scheduling Queues

Processes in the system are organized into queues:

- The **job queue** holds all processes in the system.

- The **ready queue**, stored as a linked list, contains processes in main memory that are ready to execute. It has a header pointing to the first and last PCB, with each PCB linking to the next in the queue.
- Additional queues manage processes waiting for specific events, such as I/O completion, or those interrupted or terminated after CPU execution.

PROCESS REPRESENTATION IN LINUX

In Linux, the process control block is represented by the task_struct C structure. It contains essential information about a process, including its state, scheduling and memory management details, open files, and pointers to its parent and children processes.

```
pid_t pid; /* process identifier */
long state ; / * stat e of the process */
unsigned int time..slice / * scheduling information */
struct files_struct *files; / * list of open file s */
struct mm_struct *mm; /* address space of this process */
```

In Linux, a process's state is stored in the long state field within the task_struct. All active processes are organized in a doubly linked list of task_struct, and the kernel uses a pointer current to track the process currently executing.

To change the state of the currently running process, the kernel updates the state field in its task_structusing:

current->state = new_state;, where current points to the executing process

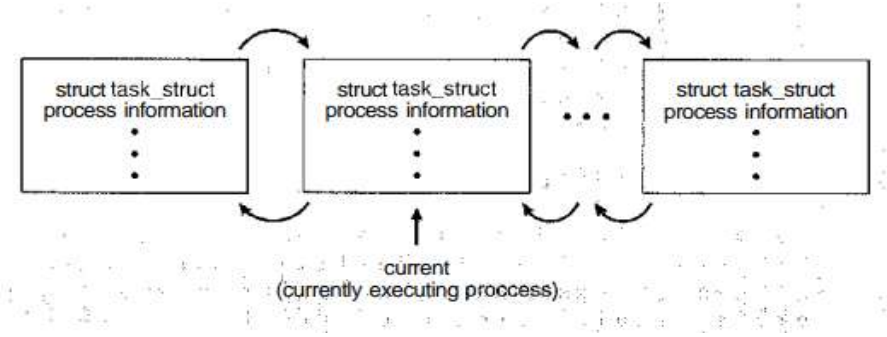


Figure 3.5 Active processes in Linux.

When a process requests I/O, like accessing a disk, it may need to wait if the device is busy. Each device has its own **device queue** for processes waiting for access.

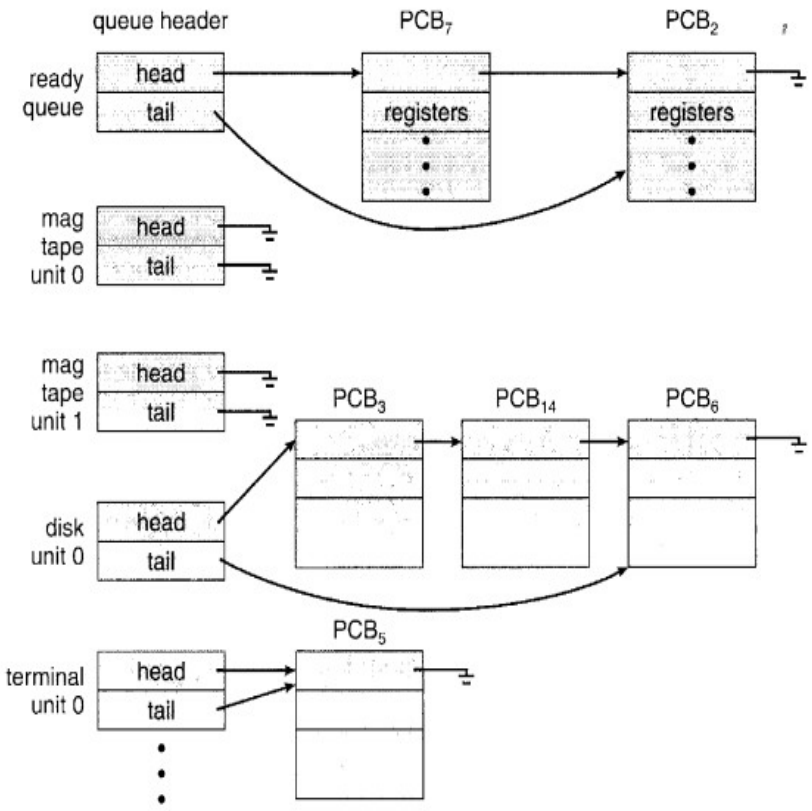


Figure 3.6 The ready queue and various I/O device queues

Queueing diagrams represent process scheduling, with **rectangular boxes** as queues (e.g., ready and device queues) and **circles** as resources. Arrows show process flow.

A new process enters the **ready queue** and waits to be dispatched. Once executing, it may:

1. Issue an I/O request and move to a device queue.
2. Create a subprocess and wait for its termination.
3. Be interrupted and return to the ready queue.

The process cycles through states until termination, after which it is removed from all queues and its PCB and resources are freed.

3.2.2 Schedulers

Throughout its lifetime, a process moves between scheduling queues, and the **scheduler** selects processes for execution.

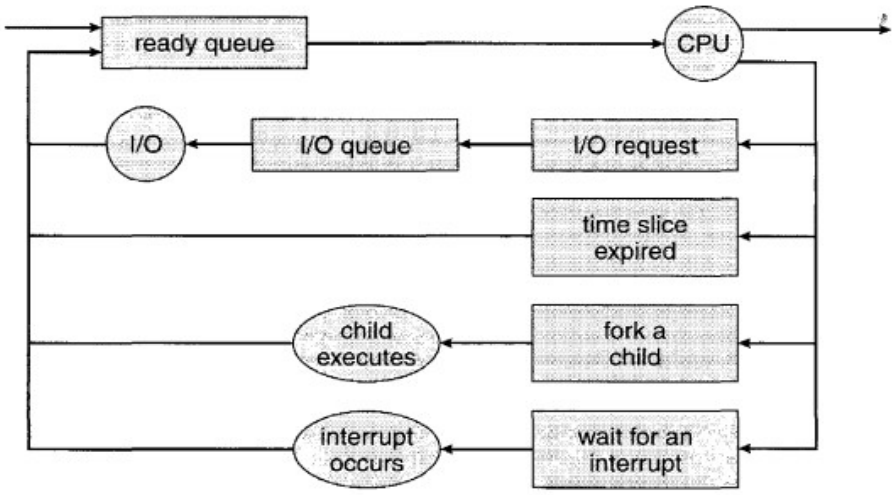


Figure 3.7 Queueing-diagram representation of process scheduling.

- **Long-term scheduler (job scheduler):**
  - Selects processes from the pool of jobs on storage and loads them into memory.
  - Executes infrequently and controls the degree of multiprogramming.
  - Balances **I/O-bound** (spend more time on I/O) and **CPU-bound** (spend more time on computation) processes for system efficiency.
- **Short-term scheduler (CPU scheduler):**
  - Selects processes from the ready queue for CPU allocation.
  - Executes frequently, often every 100 ms, requiring high speed to avoid CPU time wastage.

A balanced mix of I/O-bound and CPU-bound processes ensures optimal system performance by keeping both the CPU and I/O devices utilized effectively.

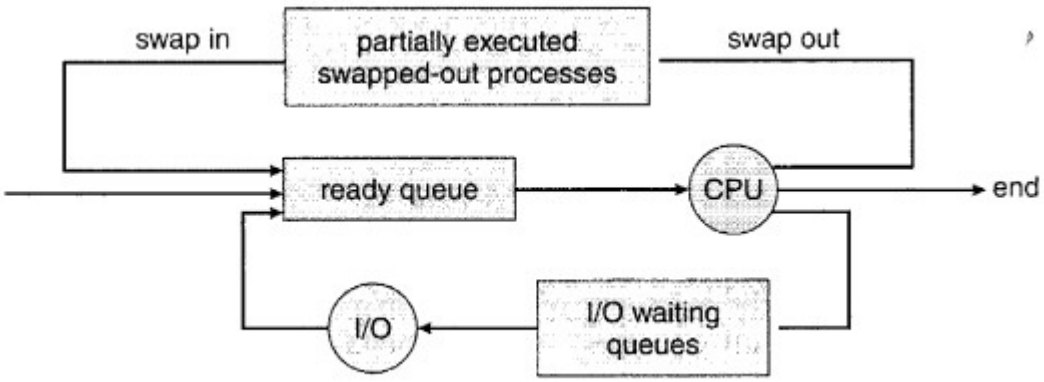


Figure 3.8 Addition of medium-term scheduling to the queueing diagram.

In some systems, like UNIX and Windows, the **long-term scheduler** may be minimal or absent, as new processes are directly handled by the short-term scheduler. System stability in such cases relies on physical limits (e.g., number of terminals) or user behavior.

Medium-term scheduler:

- Used in time-sharing systems to optimize performance.
- Temporarily removes processes from memory (swapping) to reduce multiprogramming.
- Reintroduces processes later, resuming execution from where they stopped.
- Helps improve process mix or manage memory overcommitment.

3.2.3 Context Switch

Interrupts prompt the operating system to pause the current process, save its **context** (CPU registers, process state, memory info), and switch to a kernel routine. This process, called a **context switch**, involves:

1. **State Save**: Saving the current process's context in its PCB.
2. **State Restore**: Loading the context of the next scheduled process.

Key points about context switching:

- It is **overhead** with no useful work done during the switch.
- Speed depends on hardware, memory, and OS complexity.
- Advanced systems may need additional work, like switching memory address spaces.
- Some processors, like Sun UltraSPARC, optimize this with multiple register sets, reducing overhead.

3.3 Operations on Processes

Most systems allow concurrent process execution and dynamic creation and termination of processes. Therefore, these systems need mechanisms for process creation and termination. This section covers the process creation mechanisms, with examples from UNIX and Windows systems.

3.3.1 Process Creation

A process can create multiple new processes during execution using a create-process system call. The creating process is called the **parent process**, and the newly created ones are its **children**. These children can further create other processes, forming a **process tree**. Most operating systems, like UNIX and Windows, assign each process a **unique process identifier (PID)**, which is typically an integer.

In Solaris, the root of the process tree is the **sched** process (PID 0), which creates child processes like **pageout** and **fsflush** to manage memory and file systems. It also spawns the **init** process, which acts as the root parent for all user processes.

The **init** process creates processes such as **inetd** (responsible for networking services like telnet and FTP) and **dtlogin** (providing the user login interface). When a user logs in, **dtlogin** initiates an **Xsession**, which creates a command-line shell process (e.g., **csh**). Users can then execute commands, generating child processes like **ls** and **cat**.

For instance, a **csh** process (PID 7778) may represent a user logged in via telnet. This user might start applications like a **Netscape browser** (PID 7785) or the **Emacs editor** (PID 8105), which appear as child processes in the tree.

In UNIX, processes can be listed using the ps command. For example, the command ps -e l displays detailed information about all active processes. By recursively tracing parent processes back to the **init** process, a process tree similar to Figure 3.9 can be constructed.

A process requires resources like **CPU time, memory, files, and I/O devices** to complete its tasks. When creating a subprocess, the child process may either obtain resources directly from the operating system or inherit a portion of the parent process's resources. The parent can either partition its resources among its children or share some resources, such as memory or files, across multiple children. Limiting a child process to a subset of its parent's resources helps prevent system overload from excessive subprocess creation.

Additionally, child processes may receive **initialization data** from the parent. For instance, a process tasked with displaying an image file (e.g., img.jpg) might be passed the filename as input by the parent process. The child process can use this input to open the file and send its contents to the output device, such as a terminal. In some operating systems, resources are directly passed to child processes, allowing them to seamlessly transfer data between allocated resources like open files and output devices.

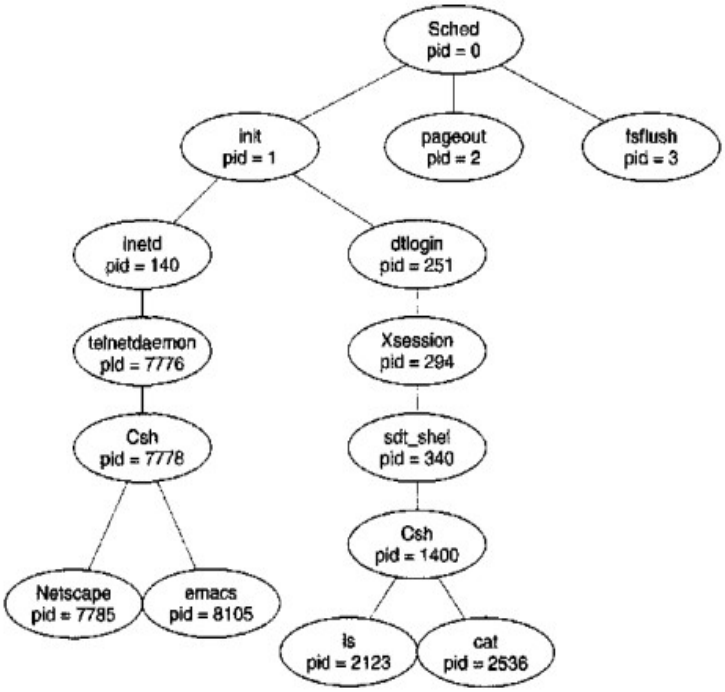


Figure 3.9 A tree of processes on a typical Solaris system.

When a process creates a new process, two execution scenarios are possible:

1. The parent continues running concurrently with its children.
2. The parent waits for some or all of its children to terminate.

For the child process's address space, two possibilities exist:

1. The child is a duplicate of the parent (same program and data).
2. The child loads a new program into its address space.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
        pid-t pid;
        /* fork a child process */
        pid = fork();
        if (pid < 0)
        {
```

```
        /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit (-1) ;
        }
        else if (pid == 0)
         {
        /* child process */
        execlp("/bin/Is","Is",NULL);
        }
        else
        {
        /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit (0) ;
        }
}
```

Figure 3.10 C program forking a separate process

In UNIX, processes are identified by unique process IDs. A new process is created using the fork() system call, which duplicates the parent's address space. Both the parent and child continue execution after fork(), but with different return values: the child receives a return value of 0, while the parent gets the child's PID.

The exec() system call is typically used after fork() to load a new program into the child's memory, replacing the previous program. This allows the parent and child to communicate initially and then operate independently. The parent can create additional children or use the wait() system call to pause execution until a child process terminates.

The C program in Figure 3.10 demonstrates UNIX system calls for process creation and management. After a fork() system call, two processes run the same program: the child process (pid = 0) and the parent process (pid > 0). The child replaces its address space with the /bin/ls command using execlp() (a version of exec()), while the

parent waits for the child to complete using the wait() system call. Once the child terminates, the parent resumes and eventually exits.

In Windows, process creation is handled by the CreateProcess() function. Unlike fork(), this function loads a specified program into the child's address space during creation. It requires ten parameters to specify process details. The program in Figure 3.12 demonstrates CreateProcess() by creating a child process to run mspaint.exe. Further details on Win32 process management can be found in the chapter's bibliographical notes.

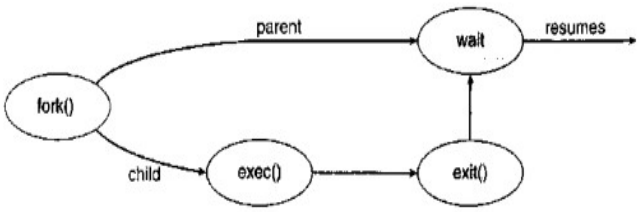


Figure 3.11 Process creation.

```
#include <stdio.h>
#include <windows.h>
int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
        // allocate memory
        ZeroMemory(&si, sizeof (si)) ;
        si.cb = sizeof (si) ;
        ZeroMemory(&pi, sizeof(pi));
        // create child process
        if (!CreateProcess(NULL, // use command line
        "C:\\WINDOWS\\system32\\mspaint.exe", // command line
        NULL, // don't inherit process handle
        NULL, // don't inherit thread handle
        FALSE, // disable handle inheritance
        0, //no creation flags
        NULL, // use parent's environment block
```

```
        NULL, // use parent's existing directory
        &si,
        &pi))
        {
                fprintf(stderr, "Create Process Failed");
                return -1;
        }
        // parent will wait for the child to complete
        WaitForSingleObject(pi.hProcess, INFINITE);
        printf("Child Complete");
        // close handles
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
}
```

Figure 3.12 Creating a separate process using the Win32 API.

In Windows, CreateProcess() uses two structures: **STARTUPINFO** and **PROCESS_INFORMATION**. STARTUPINFO defines properties like window size, appearance, and I/O file handles, while PROCESS_INFORMATION holds the handle and IDs of the new process and its thread. Memory for these structures is allocated using ZeroMemory() before calling CreateProcess().

The first two parameters of CreateProcess() are the application name and command line parameters. If the application name is NULL, the command line specifies the application to load (e.g., mspaint.exe). Default settings are used for inheriting handles, creation flags, environment block, and starting directory. The pointers to the STARTUPINFO and PROCESS_INFORMATION structures are also passed.

The parent process waits for the child to complete using WaitForSingleObject(), which monitors the child's handle (pi.hProcess). Once the child process exits, control returns to the parent.

3.3.2 Process Termination

A process terminates when it finishes execution and calls the exit() system call. It may return a status to its parent process via wait(). The operating system deallocates all resources of the process. Termination can also occur if one process terminates another

using system calls, usually by the parent process. A parent can terminate a child for reasons such as resource overuse, task completion, or the parent's own termination. Some systems, like VMS, enforce cascading termination, where if a parent process terminates, all its children are also terminated.

In UNIX, a process can be terminated using the exit() system call. The parent can wait for the termination of a child using wait(), which returns the child's process ID. If the parent terminates, its children are reassigned to the init process, ensuring they still have a parent to collect their status and execution data.

3.4 SUMMARY

A process is a program in execution that changes state during execution. Its states include **new**, **ready**, **running**, **waiting**, and **terminated**. Each process is managed by a **process control block (PCB)**, which links processes in queues.

Processes not executing reside in **I/O request queues** or the **ready queue**, which holds processes awaiting CPU time. **Long-term scheduling** selects processes to compete for the CPU, influenced by resource allocation like memory. **Short-term scheduling** chooses a process from the ready queue for execution.

Parent processes can create child processes, executing either sequentially or concurrently. Concurrent execution supports **information sharing**, **speedup**, **modularity**, and **convenience**.

3.5 KEY TERMS

Process, State, Control Block, Scheduling, Ready Queue, Child, Concurrency, Context Switching, Thread, PCB, Process Scheduler, Program counter, Process Identifier.

3.6 SELF-ASSESSMENT QUESTIONS

1. What is a process, and how is it different from a program?
2. Explain the various states of a process with a suitable diagram.
3. What is a Process Control Block (PCB), and what information does it contain?
4. Compare long-term, short-term, and medium-term schedulers in process scheduling.
5. Describe the different operations performed on processes, including process creation and termination.
6. What is context switching, and why is it essential in process scheduling?

3.7 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”, Seventh Edition, Wiley.
2. William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3. Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4. Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5. Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR
Dr. Neelima Guntupalli

CHAPTER 4

INTERPROCESS COMMUNICATION

Aim and Objectives:

- Understand the basics of interprocess communication (IPC).
- Learn examples of IPC systems and their uses.
- Explore methods like message passing, shared memory, and sockets.
- Understand communication in client-server systems.
- Study IPC design in client-server architectures.

Structure:

4.1 INTERPROCESS COMMUNICATION

Processes in an operating system can be **independent** or **cooperating**. Independent processes neither affect nor are affected by others, while cooperating processes can interact and share data.

Reasons for process cooperation:

1. **Information Sharing**: Enables concurrent access to shared data like files.
2. **Computation Speedup**: Tasks can be divided into subtasks to run in parallel on systems with multiple processing units.

3. **Modularity**: Supports system design by dividing functions into separate processes or threads.
4. **Convenience**: Allows users to perform multiple tasks, like editing, printing, and compiling, simultaneously.

Cooperating processes require **Inter Process Communication (IPC)**, which can be achieved through two models:

1. **Shared Memory**: A shared memory region is used for processes to exchange data by reading and writing to it. It is faster as communication occurs at memory speeds, with minimal kernel intervention after setup.
2. **Message Passing**: Processes exchange messages. It is simpler to implement, especially for small data or inter computer communication, as it avoids conflicts.

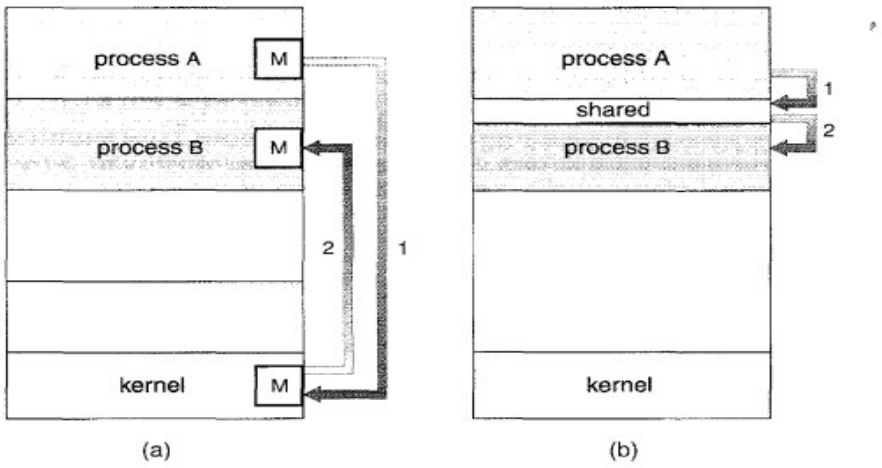


Figure 4.1 Communications models. (a) Message passing. (b) Shared memory.

Comparison:

- **Message Passing**: Easier for small data exchanges, requiring kernel calls for each operation.
- **Shared Memory**: Faster for large data transfers, requiring kernel calls only during setup.

Both models are commonly used in operating systems, often complementing each other.

4.1.1 Shared-Memory Systems

Shared Memory in IPC:

Processes establish a shared memory region to exchange data. This requires attaching the shared region to their address spaces, bypassing the usual OS restrictions on memory access. Processes decide the data format and location and must prevent simultaneous writes to the same location.

Producer-Consumer Problem:

This illustrates process cooperation where a producer generates data consumed by a consumer. Examples include a compiler producing assembly code for an assembler or a web server providing resources to a browser.

Shared Memory Solution:

- A shared buffer allows concurrent execution of producer and consumer processes.
- **Unbounded Buffer**: No size limit; producer always produces, consumer may wait.
- **Bounded Buffer**: Fixed size; producer waits if full, consumer waits if empty.

Proper synchronization ensures data is produced before consumption.

Bounded Buffer in Shared Memory:
A **circular buffer** with size BUFFER_SIZE is shared between producer and consumer processes.

```
#define BUFFER_SIZE 10

typedefstruct {

}

item;

item buffer [BUFFER_SIZE] ;

int in = 0;

int out = 0;
```

- **Variables**:
  - in: Points to the next free buffer position.
  - out: Points to the first full buffer position.
- **Conditions**:
  - Buffer is empty if in == out.
  - Buffer is full if ((in + 1) % BUFFER_SIZE) == out.

The producer stores items in nextProduced and writes to the buffer, while the consumer retrieves items into nextConsumed. At most, BUFFER_SIZE - 1 items can be stored in the buffer. A solution to allow storing BUFFER_SIZE items is left as an exercise.

This example does not address concurrent access issues, which require synchronization mechanisms discussed in next chapters.

```
itemnextProduced;
while (true)
{
        /* produce an item in next Produced */
        while (((in + 1) % BUFFER_SIZE) == out);
        /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Figure 4.2 The producer process

```
itemnextConsumed;
while (true)
{
        while (in == out);
        // do nothing
        nextConsumed = buffer [out] ;
        out = (out + 1) % BUFFER_SIZE;
        /* consume the item in nextConsumed */
}
```

Figure 4.3 The consumer process.

4.1.2 Message-Passing Systems

Unlike shared memory, message passing allows processes to communicate and synchronize without sharing an address space. This is especially useful in distributed systems where processes may reside on different computers, such as in a chat program exchanging messages over a network.

Key Features:

- **Operations**: send (message) and receive (message).
- **Message Sizes**:
  - Fixed-size: Simpler implementation but harder for programmers.
  - Variable-size: Easier for programmers but more complex to implement.

Processes must exchange messages over a logical communication link, implemented through methods like:

1. **Direct or Indirect Communication**
2. **Synchronous or Asynchronous Communication**
3. **Automatic or Explicit Buffering**

These methods are explored further in the subsequent discussion.

4.1.2.1 Naming

Processes communicate through **direct** or **indirect communication**:

Direct Communication

- Processes explicitly name the sender or receiver.
- Primitives:
  - **send(P, message):** Sends a message to process P.
  - **receive(Q, message):** Receives a message from process Q.
- **Properties:**
  - Links are automatically established between two processes.
  - Each link is unique to one process pair.
  - Communication requires both sender and receiver to name each other (symmetric) or only the sender to name the receiver (asymmetric).

In **asymmetric communication**, the receiver accepts messages from any process, with the sender's identifier stored in a variable.

Limitations

- Limited modularity: Changing process identifiers requires updates across all associated references.
- Hard-coding identifier is less flexible than indirect communication techniques.

Indirect Communication involves using **mailboxes** (or ports) for message exchange.

Key Features

- Messages are sent to and received from mailboxes, each with a unique identifier (e.g., POSIX message queues use integers).
- Processes communicate only if they share a mailbox.

Primitives

- **send(A, message):** Sends a message to mailbox A.
- **receive(A, message):** Receives a message from mailbox A.

Properties of Communication Links

1. A link is established only between processes with a shared mailbox.
2. A link may involve multiple processes.
3. Multiple links can exist between processes through different mailboxes.

Message Handling

When multiple processes share a mailbox (e.g., P1 sends to A; P2 and P3 receive from A):

- The receiver can be determined by:
    - Limiting the link to two processes.
    - Allowing only one process to receive at a time.
    - Using an arbitrary or defined algorithm (e.g., round-robin) for selecting the receiver.

Mailbox Ownership

1. **Process-owned Mailboxes**:
    - Only the owner can receive messages.
    - Mailbox disappears when the owner terminates.
    - Other processes are notified if the mailbox no longer exists.
2. **OS-owned Mailboxes**:
    - Independent of processes.
    - System supports creation, message exchange, and deletion of mailboxes.
    - Ownership and receiving privileges can be reassigned to other processes.

This approach allows flexible communication but requires careful management to avoid conflicts.

4.1.2.2 Synchronization

Communication between processes uses **send()** and **receive()** primitives, implemented as either **blocking** (synchronous) or **nonblocking** (asynchronous):

- **Blocking send:** The sender waits until the message is received.
- **Nonblocking send:** The sender resumes operation after sending the message.
- **Blocking receive:** The receiver waits for a message to arrive.
- **Nonblocking receive:** The receiver retrieves a message or returns null if none is available.

When both send() and receive() are blocking, a **rendezvous** occurs, simplifying problems like the **producer-consumer problem**. For instance, the producer blocks after sending a message until it is delivered, while the consumer blocks until a message is available.

These synchronous and asynchronous concepts are widely applied in operating system I/O algorithms.

4.1.2.3 Buffering

Messages exchanged between processes are stored in a temporary queue, which can be implemented in three ways:

1. **Zero Capacity:** The queue cannot hold messages. The sender blocks until the recipient receives the message. *(No buffering)*
2. **Bounded Capacity:** The queue has a finite length. If not full, messages are added, and the sender continues. If full, the sender blocks until space becomes available. *(Automatic buffering)*
3. **Unbounded Capacity:** The queue has infinite length, allowing any number of messages. The sender never blocks. *(Automatic buffering)*

The capacity determines whether the sender must wait during communication.

4.2 EXAMPLES OF IPC SYSTEMS

This section explores three IPC systems: POSIX API for shared memory, message passing in the Mach OS, and Windows XP, which uses shared memory for certain message-passing mechanisms.

4.2.1 An Example: POSIX Shared Memory

The POSIX API provides IPC mechanisms like shared memory. To create shared memory, the shmget() system call is used, specifying a key (IPC_PRIVATE for new memory), size, and mode (read/write permissions). It returns an integer identifier.

To access shared memory, processes attach it to their address space using shmat(), which requires the memory ID, an optional memory location, and a flag for read/write access. The function returns a pointer to the memory region. Data can then be read or written using this pointer, e.g., sprintf(shared_memory, "Message").

To detach shared memory, use shmdt(pointer). To remove it, use shmctl(id, IPC_RMID). For example, a process can create a shared segment, write data, and later detach and delete it.

```
#include <sys/shm.h>

#include <sys/stat. h>

int main()

{

        /* the identifier for the shared memory segment */

        int segment_id;

        /* a pointer to the shared memory segment */

        char* shared_memory;

        /* the size (in bytes) of the shared memory segment */

        const int size = 4096;

        /* allocate a shared memory segment */

        segmented = shmget (IPC_PRIVATE, size, S_IRUSR | S_IWUSR)

        /* attach the shared memory segment */

        shared.memory = (char *) shmat (segment_id, NULL, 0) ;

        /* write a message to the shared memory segment */

        sprintf (shared-memory, "Hi there!");

        /* now print out the string from shared memory */

        printf ("*%s\n" , shared_memory) ,•

        /* now detach the shared memory segment */

        shmdt (sharecLmemory) ;

        /* now remove the shared memory segment */

        shmctl (segment-Id, IPC_RMID, NULL);
```

```
        return 0;
}
```

Figure 4.4 C program illustrating POSIX shared-memory API.

4.2.2 An Example: Mach

The Mach operating system, part of Mac OS X, uses message-based IPC. Tasks, similar to processes with multiple threads, communicate via messages sent to and received from *ports* (mailboxes). System calls, including msg_send() (sending), msg_receive() (receiving), and msg_rpc() (remote procedure calls), operate through messages.

Each task has a Kernel port for kernel communication and a Notify port for event notifications. New mailboxes are created with port_allocate(), allowing only the owner to send and receive messages, though ownership can be transferred. Mailbox queues hold messages in FIFO order for a single sender but may vary with multiple senders.

Messages include a fixed-length header (with destination and return mailbox names) and a variable-length data portion containing typed data. These messages can carry objects like memory segments and task states, enabling efficient inter-task communication.

Mach's send and receive operations for IPC are flexible:

Sending Options:

1. Wait indefinitely if the mailbox is full.
2. Wait up to n milliseconds.
3. Return immediately without sending.
4. Temporarily cache one message until space is available.

Receiving Options:

A task can receive from a specific mailbox or a mailbox set. If no message is present, the thread can wait for n milliseconds or not wait.

Key Features:

- Mailbox sets group multiple mailboxes for simplified access.
- The port_status() system call checks the number of messages in a mailbox.

Mach, designed for distributed systems but suitable for single-processor environments (e.g., Mac OS X), uses virtual memory to map sender messages directly to receiver address spaces, avoiding performance issues caused by double-copying.

4.2.3 An Example: Windows XP

Windows XP exemplifies modular design, supporting multiple operating environments (subsystems) via a message-passing mechanism. Applications act as clients, communicating with subsystem servers through the Local Procedure Call (LPC) facility, optimized for same-machine communication.

LPC uses *port objects* to establish client-server connections. Two types of ports are used:

1. **Connection Ports**: Public, used to set up communication channels.
2. **Communication Ports**: Private, for exchanging messages.

Communication Process:

- The client opens a connection port.
- Sends a connection request.
- The server creates two private communication ports and shares one with the client.
- Both use their respective port handles for message exchange.

Windows XP supports two message-passing techniques for communication over a port.

1. **Small Messages**: For messages up to 256 bytes, the port's message queue is used as intermediate storage, with messages copied between processes.
2. **Large Messages**: For larger messages, a *section object* (shared memory region) is used. The client or server decides whether to use a section object when setting up the channel, passing a small message containing a pointer and size information. This method avoids data copying.

Both methods support asynchronous handling via a callback mechanism when immediate responses aren't possible. The LPC system is not part of the Win32 API but is used indirectly through standard remote procedure calls (RPCs) in the Win32 API.

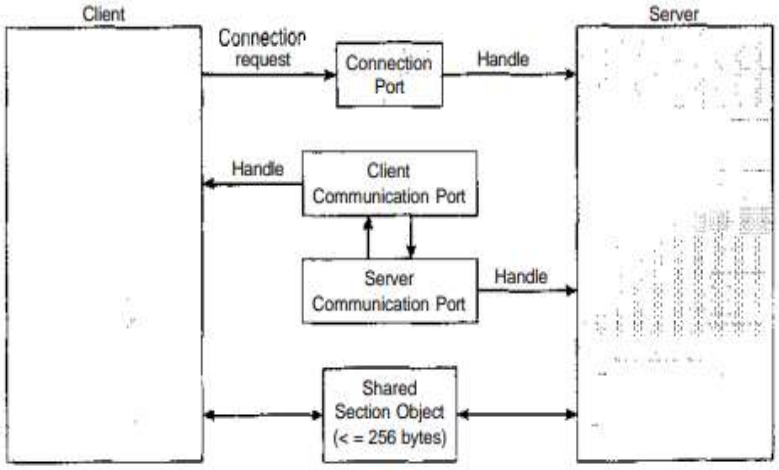


Figure 4.5 Local procedure calls in Windows XP.

4.3 COMMUNICATION IN CLIENT-SERVER SYSTEMS

In Section 4.1, we discussed shared memory and message passing for process communication. This section explores three additional communication strategies for client-server systems: sockets, remote procedure calls (RPCs), and Java's remote method invocation (RMI).

4.3.1 Sockets

A socket is an endpoint for communication, with each process in a network communication using a pair of sockets. Sockets are identified by an IP address and port number.

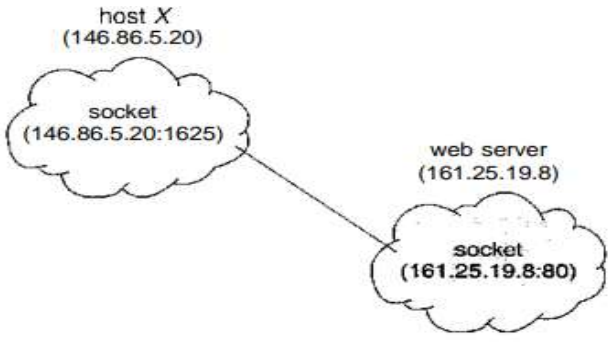


Figure 4.6 Communication using sockets.

In a client-server model, the server listens to a specific port for incoming requests, while the client connects to the server using an arbitrary port number greater than 1024. For example, a web server listens on port 80, and a client connects with a unique port number. All connections must be unique, ensuring different client connections are assigned different port numbers.

Java provides three types of sockets: TCP sockets (using the Socket class), UDP sockets (using the DatagramSocket class), and multicast sockets (a subclass of DatagramSocket), with multicast sockets allowing data to be sent to multiple recipients.

```
import java.net.*;
import java.io.*;
public class DateServer
{
        public static void main(String [] args)
        {
                try
                {
                        ServerSocket sock = new ServerSocket(6013);
                        // now listen for connections
                        while (true)
                        {
                                Socket client = sock.accept();
                                PrintWriter pout = new
                                PrintWriter(client.getOutputStream(), true);
                                // write the Date to the socket
                                pout.println(new java.util.Date() .toString());
                                // close the socket and resume
                                // listening for connections
                                client.close();
                        }
                }
                catch (IOException ioe)
                {
                        System.err.println(ioe);
                }
        }
}
```

Figure 4.7 Date server.

In this example, a date server uses connection-oriented TCP sockets. The server listens on port 6013 for client requests. Upon receiving a connection request, the server returns the current date and time. It creates a ServerSocket on port 6013 and uses the accept() method to block and wait for a client connection. Once a connection is received, the server creates a PrintWriter object to send the date to the client using println(). After sending the date, the server closes the socket and resumes listening for new requests.

```
import java.net.*;
import java.io.*;
public class DateClient
{
        public static void main(String[] args)
        {
                try
                 {
                        //make connection to server socket
                        Socket sock = new Socket("127.0 . 0.1",6013) ;
                        InputStream in = sock.getlnputStream();
                        BufferedReader bin = new
                        BufferedReader(new InputStreamReader(in));
                        // read the date from the socket
                        String line;
                        while ( (line = bin.readLine()) != null)
                        System.out.println(line);
                        II close the socket connection
                        sock.close () ;
                }
                catch (IOException ioe)
                 {
                        System.err.println(ioe);
                }
        }
}
```

Figure 4.8 Date client

A client communicates with the server by creating a socket and connecting to the server's listening port, as shown in Figure 4.8. The client connects to the server at IP address 127.0.0.1 on port 6013. After establishing a connection, the client reads data from the socket using standard I/O statements. Once it receives the date from the server, the client closes the socket and exits.

The IP address 127.0.0.1, known as the loopback address, allows communication on the same host. Alternatively, the IP address of a remote host or a host name (e.g., www.example.com) can be used. While socket communication is efficient, it is low-level, requiring the client and server to define a data structure. Higher-level communication methods, such as remote procedure calls (RPCs) and remote method invocation (RMI), provide more structured approaches.

4.3.2 Remote Procedure Calls

Remote Procedure Call (RPC) abstracts the procedure-call mechanism for networked systems, allowing processes on separate systems to communicate via structured, message-based communication. Unlike simple IPC, RPC messages include the function to execute, its parameters, and the port number of an RPC daemon on the remote system. For example, a server daemon on port 3027 could provide a list of current users to any requesting system.

RPCs mimic local procedure calls by using client-side stubs that handle communication details. When a client invokes a remote procedure, the stub packages parameters (parameter marshalling), identifies the server port, and sends a message. On the server side, another stub receives the message, executes the procedure, and sends the result back to the client. This approach simplifies network communication while hiding its complexity from the user.

One issue in RPC communication is handling differences in data representation between client and server systems (e.g., big-endian vs. little-endian). To address this, many RPC systems use an external data representation (XDR), converting machine-dependent data into XDR on the client side and converting it back on the server side.

Another challenge is ensuring the correct semantics for RPC calls. Unlike local procedure calls, RPCs can fail or be duplicated due to network errors. To ensure a message is processed at most once, a timestamp is attached to each message. The server tracks timestamps to avoid processing duplicates.

For "exactly once" semantics, the server uses the "at most once" protocol and acknowledges receipt of the RPC call. The client then resends the RPC periodically until it receives this acknowledgment, ensuring the call is executed only once.

In RPC communication, binding between the client and server is necessary but challenging since neither has full information about the other. Two common approaches to binding are:

1. **Fixed Port Addresses**: The RPC call is associated with a fixed port number at compile time. The server's port cannot be changed after compilation.
2. **Dynamic Binding**: A rendezvous (matchmaker) daemon on a fixed port helps the client find the server's port dynamically. The client sends a request to the daemon, which returns the appropriate port number.

While fixed binding is simpler, dynamic binding is more flexible, though it requires the extra overhead of an initial request.

RPC is useful in distributed file systems, where clients send messages to the server's RPC daemon for file operations (read, write, delete, etc.). The server executes the operation and returns the result.

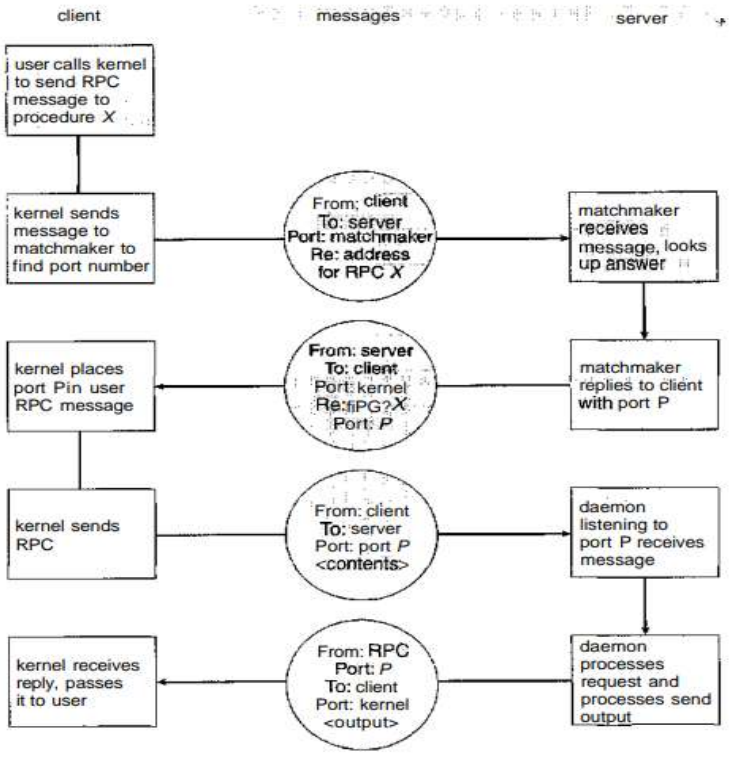


Figure 4.9 Execution of a remote procedure call (RPC).

4.3.2 Remote Method Invocation

Remote Method Invocation (RMI) is a Java feature that allows invoking methods on remote objects, which reside in a different Java Virtual Machine (JVM), either on the same machine or over a network. RMI differs from RPC in two key ways: RPC is procedural, supporting only remote functions, while RMI is object-based, allowing

method invocation on remote objects. Additionally, RMI can pass objects as parameters, unlike RPC, which uses simple data structures.

RMI makes remote method calls transparent by using **stubs** and **skeletons**. The **stub** on the client side acts as a proxy for the remote object and sends the method name and parameters to the server. The **skeleton** on the server side unmarshals the data, invokes the method, and sends the result back to the client through the stub.

For example, when a client calls server.someMethod(A, B), the stub packages the method call and parameters, sends them to the skeleton, which processes the request and returns the result.

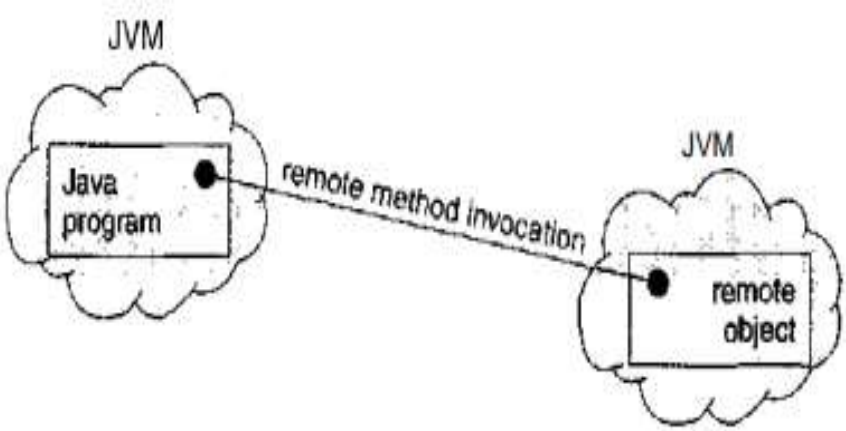


Figure 4.10 Remote method invocation.

When a client invokes someMethod(A, B), the stub marshals the parameters and method name into a parcel and sends it to the server. The server's skeleton unmarshals the parameters and invokes the method on the server. After execution, the skeleton marshals the return value and sends it back to the client, where the stub unmarshals it for the client to use.

RMI abstracts the use of stubs and skeletons, allowing developers to invoke remote methods as if they were local. However, there are important rules for parameter passing:

- **Local objects** are serialized and passed by value (copy).
- **Remote objects** are passed by reference.
- For local objects to be passed to remote objects, they must implement java.io.Serializable to support object serialization, which writes an object's state to a byte stream.

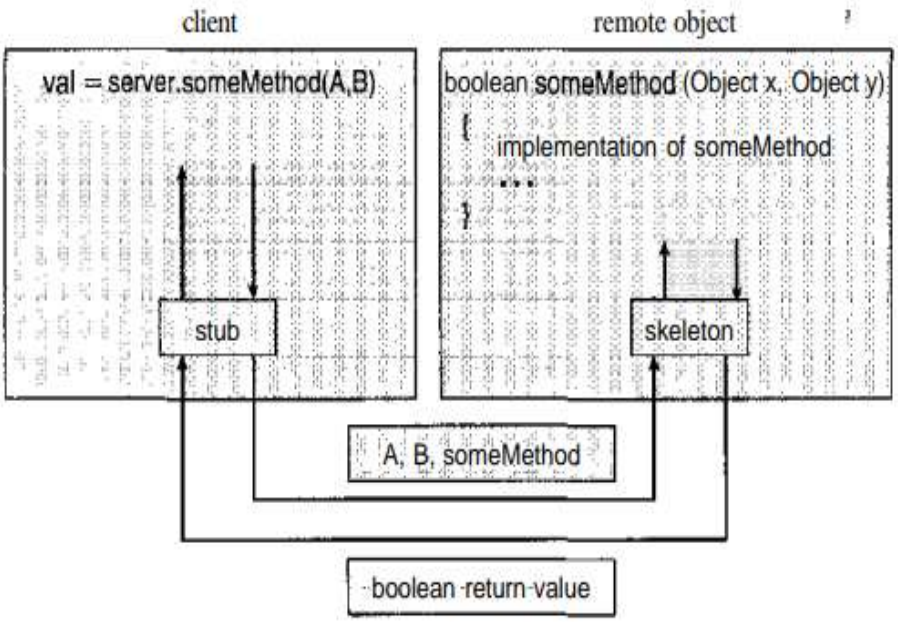


Figure 4.11 Marshalling parameters.

4.4 Summary

A process is a program in execution, and it can be in one of several states: new, ready, running, waiting, or terminated. The process is managed by its Process Control Block (PCB). When not executing, processes are placed in queues, such as the ready queue (for processes waiting for CPU) and I/O request queues. Long-term scheduling selects processes for CPU contention, while short-term scheduling picks processes from the ready queue.

Processes can be independent or cooperating. Cooperating processes use inter-process communication (IPC) methods, including shared memory (where processes communicate through shared variables) and message passing (where processes exchange messages). Both methods can be used together.

In client-server systems, communication can occur through sockets, remote procedure calls (RPCs), or Java's remote method invocation (RMI). A socket is an endpoint for communication, and RPC allows calling remote procedures. RMI is similar to RPC but allows remote method invocation on objects, with the key difference being that RMI supports passing objects, while RPC only passes data structures.

4.5 KEY TERMS

Process, PCB (Process Control Block), Ready Queue, Scheduling, Cooperating Processes, Shared Memory, Message Passing, Socket, RPC (Remote Procedure Call), RMI (Remote Method Invocation).

4.6 SELF ASSESSMENT QUESTIONS

1. What are the states of a process and how does the OS schedule processes?

2. What is the difference between long-term and short-term scheduling?

3. What is Interprocess Communication (IPC) and its two main methods?

4. Explain shared memory and message passing in IPC. How are they different?

5. How does client-server communication work using sockets, RPC, and RMI?

6. What are the benefits and challenges of using RPC and RMI in distributed systems?

4.7 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. "Operating System Principles", Seventh Edition, Wiley.
2. William Stallings, "Operating Systems – Internals and Design Principles", Fifth Edition, Pearson Education (2007)
3. Achyut S Godbole, "Operating Systems", Second Edition, TMH (2007).
4. Flynn/McHoes, "Operating Systems", Cengage Learning (2008).
5. Deitel&Deitel, "Operating Systems",Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Dr. Neelima Guntupalli

CHAPTER 5

MULTITHREADED PROGRAMMING

Aim and Objectives:

- The main aim of Multithreaded Programming is to enable concurrent execution of tasks within a program to improve performance and resource utilization.
- This includes understanding multithreading models, libraries that support thread management, and addressing common threading issues.
- The topic also covers how operating systems implement and manage multithreading for efficient process handling.

Structure:

5.1 INTRODUCTION

A *thread* is the fundamental unit of CPU utilization, consisting of a thread ID, program counter, register set, and stack. Threads within the same process share the code section, data section, and other operating system resources like open files and signals. A traditional, or *heavyweight*, process contains a single thread of control, while a process with multiple threads can execute multiple tasks concurrently. Figure 5.1 shows the contrast between a single-threaded process and a multithreaded process.

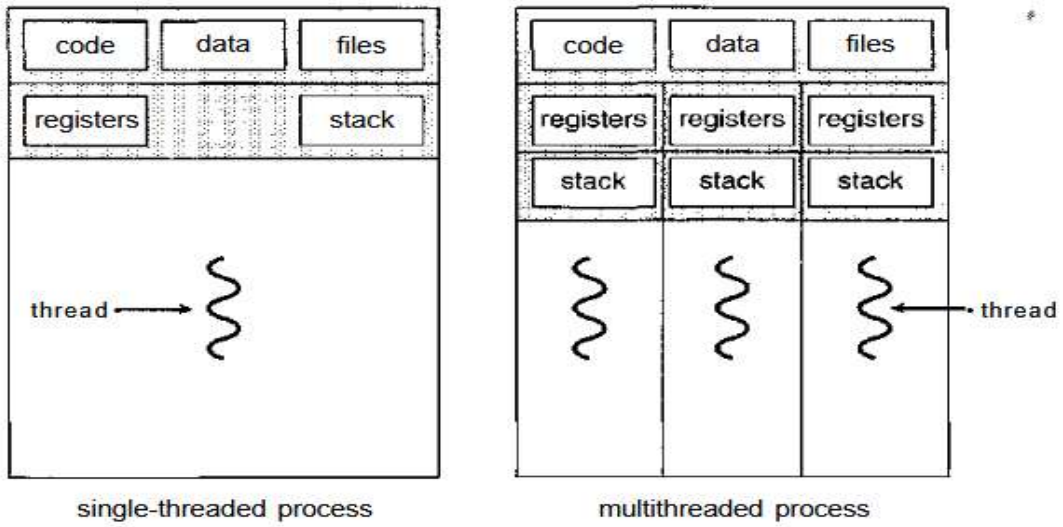


Figure 5.1 Single-threaded and multithreaded processes.

Motivation

Many software packages on modern desktops are multithreaded, using separate threads within a single process to handle multiple tasks. For example, a web browser might have one thread displaying content while another retrieves data from the network. Similarly, a word processor may have threads for displaying graphics, processing keystrokes, and running spelling and grammar checks in the background.

For applications like web servers, which need to handle numerous client requests at once, multithreading is essential. A single-threaded server can only serve one client at a time, causing delays during high demand. Multithreading enables the server to create a new thread for each request, avoiding the overhead of creating separate processes. This makes handling thousands of simultaneous client requests efficient.

Multithreading also benefits remote procedure call (RPC) systems, allowing servers to respond to multiple requests concurrently, with each request handled by a separate thread. Additionally, modern operating system kernels, such as those in Solaris and Linux, are often multithreaded, dedicating threads to tasks like device management and interrupt handling, which boosts overall system performance.

Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1 **Responsiveness**: Multithreading helps keep applications responsive. For example, a multithreaded web browser can still allow user interactions in one thread while loading an image in another, preventing delays during lengthy tasks.
2 **Resource Sharing**: Threads within the same process share memory and resources, allowing them to work within the same address space and improving resource utilization.
3 **Economy**: Threads are less resource-intensive than processes, making them more economical to create and manage. For example, in Solaris, creating a process is about 30 times slower than creating a thread.
4 **Utilization of multiprocessor architectures**: On multiprocessor systems, multithreading allows threads to run on multiple CPUs, boosting performance by enabling parallel execution.

5.2 MULTITHREADING MODELS

Threads can be implemented at either the user level, as *user threads*, or at the kernel level, as *kernel threads*. User threads operate above the kernel and are managed without its support, while kernel threads are directly supported and managed by the operating system. Most modern operating systems, including Windows XP, Linux, macOS, Solaris, and Tru64 UNIX, support kernel threads. A relationship between user and kernel threads must be established, and there are three common models for this integration.

5.2.1 Many-to-One Model

The many-to-one model (Figure 5.2) maps multiple user-level threads to a single kernel thread, managed by a user-space thread library. While efficient, this model has limitations: if one thread makes a blocking system call, the entire process is blocked, and only one thread can access the kernel at a time, preventing parallel execution on multiprocessor systems. Examples include the Green Threads library in Solaris and GNU Portable Threads

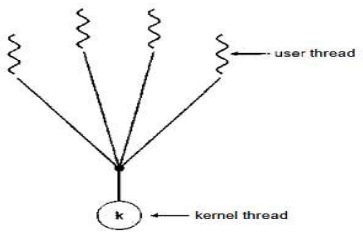


Figure 5.2 Many-to-one model

5.2.2 One-to-One Model

The one-to-one model (Figure 5.3) maps each user thread to a separate kernel thread, enabling high concurrency by allowing other threads to run if one is blocked and supporting parallel execution on multiprocessors. However, the model requires creating a kernel thread for each user thread, which can lead to overhead. To manage performance, implementations of this model often limit the number of threads. This model is used by Linux and the Windows family, including Windows 95, 98, NT, 2000, and XP.

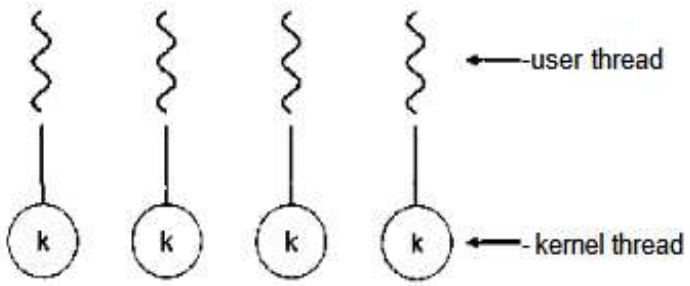


Figure 5.3 One-to-one mode

5.2.3 Many-to-Many Model

The many-to-many model (Figure 5.4) multiplexes many user-level threads onto a smaller or equal number of kernel threads, with the number of kernel threads varying by application or system. This model balances flexibility and concurrency, as it allows multiple threads to run simultaneously on multiprocessors. While the many-to-one model offers unlimited user threads, it lacks true concurrency, and the one-to-one model offers better concurrency but may limit the number of threads. The many-to-many model seeks to combine the benefits of both, allowing efficient thread management without overloading the system.

The *many-to-many model* overcomes the limitations of other models by allowing developers to create numerous user threads, which can run in parallel on multiprocessors. When a thread makes a blocking system call, the kernel can schedule another thread. A variation of this model, known as the *two-level mode (Figure 5.4) l*, multiplexes user threads to kernel threads but allows binding specific user threads to kernel threads. This model, used by systems like IRIX, HP-UX, and Tru64 UNIX, provides more flexibility. Solaris used the two-level model before version 9 but now uses the one-to-one model.

5.3 THREAD LIBRARIES

A thread library provides an API for creating and managing threads. It can be implemented in two ways:

1. **User-space library**: All code and data structures for the library exist in user space, and invoking a function results in a local function call rather than a system call.
2. **Kernel-level library**: Code and data structures reside in kernel space, and invoking a function typically results in a system call to the kernel.

The three main thread libraries in use today are:

- **POSIX Pthreads**: Can be implemented as either a user-space or kernel-level library.
- **Win32**: A kernel-level library available on Windows systems.
- **Java**: The Java thread API, which allows thread management directly within Java programs. However, it typically relies on the underlying operating system's thread library (e.g., Win32 on Windows, Pthreads on UNIX/Linux).

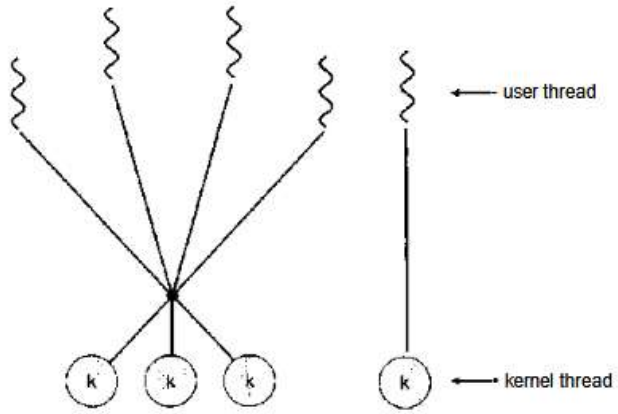


Figure5.4 Two-level model

The next section discusses basic thread creation using these libraries, illustrated through a multi-threaded program for summing non-negative integers.

$$sum = \sum_{i=0}^{N} i$$

For example, if N is 5, the function will compute the summation from 0 to 5, resulting in 15. The three programs will accept the upper bound of the summation as input via the command line. For instance, if the user enters 8, the program will output the summation of integers from 0 to 8.

5.3.1 Pthreads

Pthreads refers to the POSIX standard (IEEE 1003.1c), which defines an API for thread creation and synchronization. It is a specification for thread behavior, allowing operating system designers to implement it in various ways. Many systems, including

Solaris, Linux, Mac OS X, and Tru64 UNIX, implement the Pthreads specification. Shareware versions are also available for Windows.

The C program in Figure 5.5 illustrates the basic Pthreads API, which creates a multithreaded program to calculate the summation of a non-negative integer in a separate thread. In Pthreads, threads start execution in a specified function, such as the runner() function shown in the example. Initially, the program starts with a single thread of control.

```
#include <pthread.h> •>

#include <stdio.h>

int sum; /* this data is shared by the thread(s) */

void *runner(void *param); /* the thread */

int main(int argc, char *argv[])

{

pthread_t tid; /* the thread identifier */

pthread.attr_t attr; /* set of thread attributes */

if (argc != 2) {

fprintf(stderr,"usage: a.out <integer value>\n");

return -1;

}

if (atoi(argv[1]) < 0) {

fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));

return -1;

/* get the default attributes */

pthread.attr.init (&attr) ;

/* create the thread */

pthread^create(&tid,&attr,runner,argv[1]);
```

```
/* wait for the thread to exit */

pthread_join (tid, NULL) ;

printf("sum = %d\n",sum);

/* The thread will begin control in this function */

void *runner(void *param)

{

int i, upper = atoi(param);

sum = 0;

for (i = 1; i <= upper; i

sum += i;

pthread_exit (0) ;

}
```

Figure 5.5 Multithreaded C program using the Pthreads API.

In this Pthreads program, after some initial setup, the main() function creates a second thread that begins execution in the runner() function. Both threads share the global variable sum.

Key elements of the program include:

- **pthread.h**: This header file is essential for all Pthreads programs.
- **pthread_t tid**: This declares a thread identifier for the thread to be created.
- Each thread has a set of attributes, such as stack size and scheduling information, stored in **pthread_attr_t attr**. These attributes can be customized before creating the thread.

This structure allows for thread management and synchronization in a multithreaded environment.

In a Pthreads program, the main() function creates a second thread that begins execution in the runner() function. Both threads share global data, like the sum variable. The program includes the **pthread.h** header, and the **pthread_t tid** declares a thread identifier. Threads also have attributes, such as stack size and scheduling

information, managed using **pthread_attr_t attr**. These components enable thread creation and synchronization within a multithreaded application.

In Pthreads, thread attributes are represented using **pthread_attr_t attr**. These attributes are initialized with **pthread_attr_init(&attr)**, and by default, no specific attributes are set. The **pthread_create()** function is used to create a new thread, passing the thread identifier, attributes, the function where the thread will begin execution (like runner()), and any necessary parameters (e.g., argv[1]).

The program then has two threads: the main (parent) thread and the summation (child) thread. The parent thread waits for the child thread to complete using **pthread_join()**. Once the child thread finishes and calls **pthread_exit(0)**, the parent thread outputs the value of the shared variable sum.

5.3.2 Win32 Threads

The Win32 thread library follows a similar approach to Pthreads for creating and managing threads. In a Win32 program, the **windows.h** header file is included, and shared data between threads (such as Sum) is declared globally. Threads are created using the **CreateThread()** function, which accepts attributes like security information, stack size, and thread startup flags. By default, the thread is not started in a suspended state.

The **Summation()** function, executed by the new thread, calculates the summation and updates the global Sum. The parent thread waits for the child thread to complete using **WaitForSingleObject()**, which blocks the parent until the child thread finishes. This is similar to using **pthread_join()** in Pthreads.

5.3.3 Java Threads

Threads are the core execution model in Java, enabling concurrent task handling. The Java API offers extensive features for creating and managing threads, starting with at least one main thread in every program.

```
#inciude <windows.h>

 #include <stdio.h>

 DWORD Sum; /* data is shared by the thread(s) */

 /* the thread runs in this separate function */

 DWORD WINAPI Summation(LPVOID Param)

 {

 DWORD Upper = *(DWORD*)Param;
```

```
for (DWORD i = 0; i <= Upper; i++)
Sum += i;
return 0;
int main(int argc, char *argv[])
{
DWORD Threadld;
HANDLE ThreadHandle;
int Param;
/* perform some basic error checking */
if (argc != 2) {
fprintf(stderr,"An integer parameter is required\n");
return -1;
}
Param = atoi(argv[l]);
if (Param < 0) {
fprintf(stderr,"An integer >= 0 is required\n");
return -1;
// create the thread
ThreadHandle = CreateThread(
NULL, // default security attributes
0, // default stack size
Summation, // thread function
&Param, // parameter to thread function
0, // default creation flags
SThreadld); // returns the thread identifier
if (ThreadHandle != NULL) {
// now wait for the thread to finish
```

```
WaitForSingleObject(ThreadHandle,INFINITE);

// close the thread handle

CloseHandle(ThreadHandle);

printfC'sum = %d\n",Sum);

}
```

Figure 5.6 Multithreaded C program using the Win32 API.

In Java, even a simple program with only a main method runs as a single thread within the JVM. To create additional threads, there are two common techniques. One approach is to create a class that extends the Thread class and overrides its run() method. Alternatively, and more commonly, a class can implement the Runnable interface, allowing for greater flexibility in thread management.

```
public interface Runnable

{

public abstract void run();

}
```

When a class implements Runnable, it must define a run() method, where the code executes as a separate thread. In a multithreaded program like the example Summation class, this class implements Runnable, and a Thread object is created by passing a Runnable instance to its constructor. However, calling start() on this Thread object is what truly initiates the new thread, allocating memory and invoking the run() method.

In the summation example, two threads are created: the main thread starts in main(), and the second (child) thread begins when start() is called on the Thread object, running within the Summation class's run() method. Since Java has no global data, data is shared between threads by passing references to shared objects. Here, the main and summation threads share an instance of the Sum class, accessible via getter and setter methods.

```
class Sura

{

private int sum;

public int getSumO {

return sum;
```

```
public void setSum(ir.t sum) {
this.sum = sum;
class Summation implements Runnable
{
private int upper;
private SUIT. sumValue;
public Summation(int upper, Sum sumValue)
this.upper = upper;
this.sumValue = sumValue;
public void run() {
int sum = 0;
for (int i = 0; i <= upper,- i
sum += i ,•
sumValue.setSum(sum);
public class Driver
{
public static void main(String[] args) {
if {args.length > 0) {
if (Integer.parseint(args[0]) < 0)
System.err.println(args [0] + " must be >= 0.") ;
else {
// create the object to be shared
Sum sumObject = new Sum();
int upper = Integer.parseint(args [0]) ;
Thread thrd = new Thread(new Summation(upper, sumObject)
thrd.start();
try {
```

```
thrd.join();
System.out.println
("The sum of "+upper+" is "+sumObject.getSum()
} catch (InterruptedException ie) { }
else
System.err.println("Usage: Summation <integer value>")
```

Figure 5.7 Java program for the summation of a non-negative integer.

The JVM is implemented on a host operating system, allowing it to abstract OS-specific details and provide a consistent environment for Java programs on any platform with a JVM. The JVM specification doesn't dictate how Java threads map to the underlying OS, leaving this to the JVM implementation. For example, on Windows XP, each Java thread maps to a kernel thread following a one-to-one model, while systems like Tru64 UNIX use a many-to-many model. Solaris initially used a many-to-one model but later adopted a one-to-one model. Java thread libraries may also integrate with native OS thread libraries, such as Win32 on Windows or Pthreads on Linux.

5.4 THREADING ISSUES

In this section, we discuss some of the issues to consider with multithreaded programs.

5.4.1 The fork() and exec() System Calls

In a multithreaded program, the behavior of the fork() system call changes. When fork() is called, some UNIX systems offer two versions: one duplicates all threads, and the other duplicates only the calling thread. If exec() is called immediately after fork(), duplicating only the calling thread is efficient, as exec() replaces the entire process. However, if exec() is not used, duplicating all threads may be necessary for the separate process to maintain the program's functionality.

5.4.2 Cancellation

Thread cancellation involves stopping a thread before it completes its task. For instance, if multiple threads are searching a database and one finds the result, the others can be canceled. Cancellation can be either:

1. **Asynchronous cancellation**: The target thread is terminated immediately by another thread.
2. **Deferred cancellation**: The target thread periodically checks if it should terminate, allowing it to exit safely.

Asynchronous cancellation can lead to resource issues, as canceled threads may not release all resources properly. Deferred cancellation, by contrast, allows threads to stop at designated safe points known as ***cancellation points*** in Pthreads, ensuring resource safety and consistency.

5.4.3 Signal Handling

In UNIX systems, signals are used to notify processes of specific events. Signals can be received either synchronously or asynchronously, depending on the event's source. The signal handling process follows a consistent pattern:

1. A signal is triggered by an event.
2. The signal is delivered to the relevant process.
3. The signal must then be handled by the process.

Synchronous signals, such as illegal memory access or division by zero, are generated by operations within the process and are delivered to the same process. Asynchronous signals, such as those triggered by external events like keystrokes or timers, are delivered to processes other than the one causing the event. Each signal can be handled by a default handler or a user-defined handler.

In a multithreaded environment, signals generated by external events are delivered asynchronously. Examples include terminating a process with keystrokes like <Control><C> or when a timer expires. These signals can be handled by either a default handler or a user-defined handler. The default handler is invoked by the kernel, but users can override it with a custom handler.

In single-threaded programs, signals are straightforward to handle, as they are delivered to the process. However, in multithreaded programs, where multiple threads exist, deciding where to deliver the signal becomes more complex. Signals can be delivered to a specific thread, all threads, selected threads, or even a dedicated thread. Synchronous signals are delivered to the thread that caused them, while asynchronous signals, like process termination, are often sent to all threads. UNIX systems allow threads to specify which signals to accept or block, and signals are typically delivered to the first thread that is not blocking them. In contrast, Windows doesn't have built-in signal support, but emulates them using Asynchronous Procedure Calls (APCs), which are delivered to specific threads.

5.4.4 Thread Pools

In a multithreaded web server, each incoming request is typically handled by creating a separate thread. While this is more efficient than creating a new process for each request, there are potential issues. First, the overhead of creating and discarding threads for each request can be time-consuming. More critically, if there is no limit to

the number of threads created, it could exhaust system resources like CPU time or memory.

To address this, a **thread pool** can be used. A thread pool creates a fixed number of threads at the start and keeps them ready to handle requests. When a request arrives, an available thread is assigned to handle it, and once the task is done, the thread returns to the pool for future use. Thread pools improve performance by reusing threads and limiting the total number of active threads, which helps manage system resources more efficiently. The size of the thread pool can be dynamically adjusted based on system load, optimizing both memory usage and processing capacity.

```
DWORD WINAPI PoolFunction(AVOID Param) {

 /**

 * this function runs as a separate thread.

 **/
```

In the thread pool API, a pointer to PoolFunction() is passed to one of the API functions, which then directs a thread from the pool to execute the specified function. The function that will run as a separate thread is defined by three parameters:

1. **LPTHREAD_START_ROUTINE Function**: A pointer to the function to be executed in the new thread.
2. **PVOID Param**: A parameter that is passed to the function.
3. **ULONG Flags**: Flags that define how the thread pool should create and manage the execution of the thread. These flags control various aspects of thread handling, such as priorities or scheduling behavior.

An example of an invocation is: QueueUserWorkltemC&PoolFunction, NULL, 0);

A thread from the thread pool invokes PoolFunction() on behalf of the programmer, with no parameters and no special instructions (flag set to 0). The Win32 thread pool API also includes utilities for invoking functions periodically or after asynchronous I/O requests. Additionally, Java's java.util.concurrent package, introduced in Java 1.5, provides its own thread pool utility for managing threads.

5.4.5 Thread-Specific Data

Threads within a process share the process's data, which is one of the key benefits of multithreaded programming. However, in some cases, each thread may need its own copy of certain data, known as thread-specific data. For example, in a transaction-processing system, each thread might be assigned a unique transaction identifier,

which can be managed using thread-specific data, supported by thread libraries like Win32, Pthreads, and Java.

5.4.6 Scheduler Activations

A key issue in multithreaded programs is the communication between the kernel and the thread library, especially in systems using the many-to-many or two-level models. This coordination allows for dynamic adjustment of kernel threads, ensuring optimal performance. These models often introduce an intermediate structure called a **lightweight process (LWP)**, which acts as a virtual processor for the user-thread library. The LWP is linked to a kernel thread, and it is the kernel thread that is scheduled on physical processors. If a kernel thread is blocked (e.g., waiting for I/O), the associated LWP and user-level thread also become blocked.

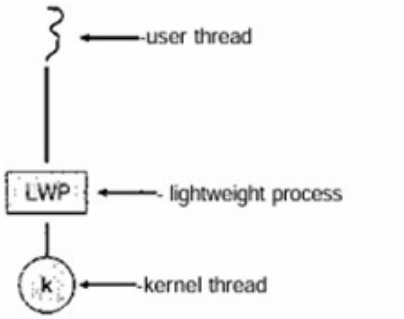


Figure 5.8 Lightweight process (LWP).

An application may require a varying number of LWPs depending on its workload. For example, a CPU-bound application on a single processor only needs one LWP, whereas an I/O-intensive application may require multiple LWPs—one for each blocking system call like simultaneous file reads. If there are fewer LWPs than required, some requests must wait. **Scheduler activation** enables the kernel to allocate a set of virtual processors (LWPs) to an application, allowing it to schedule user threads. The kernel also uses **upcall handler** to notify the application about events, like a thread about to block, triggering handlers that manage thread state and scheduling efficiently.

5.5 Operating System Examples

In this section, we explore how threads are implemented in Windows XP and Linux systems.

5.5.1 Windows XP Threads

Windows XP implements the Win32 API, which is used across Microsoft operating systems like Windows 95, 98, NT, 2000, and XP. In Windows XP, each application runs as a separate process with one or more threads, and the Win32 API is used to create these threads. Windows XP follows a one-to-one mapping model, where each user thread corresponds to a kernel thread, but it also supports a fiber library for many-to-many thread management. Key components of a thread include its unique ID, register set, user and kernel stacks, and a private storage area for libraries, collectively known as the thread's context.

ETHREAD—executive thread block

KTHREAD—kernel thread block

TEB—thread environment block

The ETHREAD structure contains key components such as a pointer to the process the thread belongs to and the address of the routine where the thread starts. It also has a pointer to the corresponding KTHREAD. The KTHREAD stores scheduling and synchronization information, along with the kernel stack and a pointer to the Thread Environment Block (TEB). Both ETHREAD and KTHREAD exist in kernel space, while the TEB resides in user space, accessed during user-mode execution. The TEB includes the thread ID, user-mode stack, and thread-specific data (or thread-local storage).

5.5.2 Linux Threads

Linux uses the fork() system call to duplicate a process and the clone() system call to create threads.

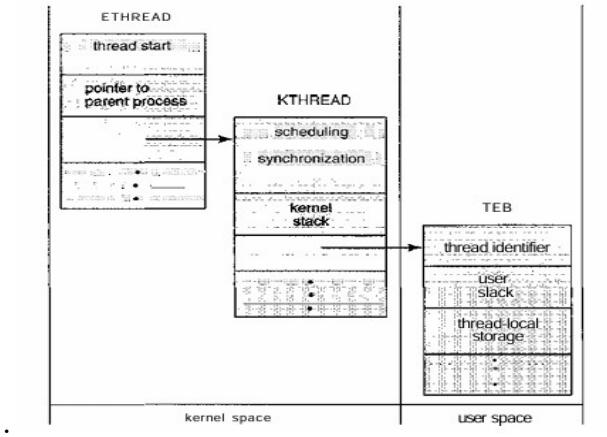


Figure 5.9 Data structures of a Windows XP thread.

However, Linux does not distinguish between processes and threads, referring to both as "tasks." The clone() system call is passed flags that define the level of resource sharing between the parent and child tasks. These flags determine how much of the process's memory and resources are shared between the two tasks.

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open fifes is shared.

In Linux, the clone() system call allows for creating tasks with varying levels of resource sharing between the parent and child. If flags like CLONE_FS, CLONE_SIGHAND, and CLONE_FILES are set, the parent and child tasks share the same file-system information, memory space, signal handlers, and open files, effectively creating a thread. Without these flags, clone() behaves similarly to fork(), creating a separate process with no resource sharing. This flexibility is made possible by the struct task_struct, a kernel data structure that represents each task and contains pointers to other data structures such as open files, signal handlers, and virtual memory.

5.6 SUMMARY

Multithreaded Programming focuses on enabling multiple threads to execute concurrently within a single application, enhancing efficiency and responsiveness. The overview introduces core concepts of multithreading and how it differs from single-threaded processing. Various *multithreading models* are explored, such as user-level and kernel-level threads, which provide different ways for threads to interact with the operating system. *Thread libraries* are tools that simplify thread management and implementation in different programming environments. Key *threading issues* like synchronization, deadlock, and race conditions are addressed to ensure thread safety. Lastly, *operating system examples* show practical implementations of multithreading in real-world OS architectures, highlighting how systems handle concurrent tasks.

5.7 KEY TERMS

Concurrency, Multithreading, Threads, Kernel, Synchronization, Deadlock, Race Condition, Thread Safety, Operating System, Process, Parallelism, Thread pool, System call.

5.8 SELF-ASSESSMENT QUESTIONS

1. What are the primary advantages of using multithreading in applications, and how does it differ from multiprocessing?
2. Explain the difference between the "many-to-one," "one-to-one," and "many-to-many" multithreading models. What are the strengths and weaknesses of each?
3. What is a thread library, and how does it assist in thread management within a program? Give an example of a popular thread library.
4. Describe a race condition and deadlock. How can these issues affect multithreaded applications, and what techniques can help prevent them?
5. How do operating systems like Linux or Windows manage threads, and what role does the OS scheduler play in thread execution?

5.9 Further Readings

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”,

Seventh Edition, Wiley.

2.William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3.Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4.Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5.Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Dr. Vasantha Rudramalla

CHAPTER 6
PROCESS SCHEDULING

Aim and Objectives:

- The goal is to introduce CPU scheduling, a fundamental concept for multiprogrammed operating systems.
- This includes explaining different CPU scheduling algorithms and discussing the criteria used to evaluate and select the most suitable CPU-scheduling algorithm for a specific system.

Structure:

6.1 INTRODUCTION

In a single-processor system, only one process can execute at any given time, while other processes must wait until the CPU is available again. The primary goal of multiprogramming is to ensure that the CPU is continuously utilized, minimizing idle time. The basic idea is straightforward: a process runs until it must wait, usually for an I/O operation to complete. Without multiprogramming, the CPU would remain idle

during this waiting period, resulting in wasted resources. However, with multiprogramming, multiple processes are loaded into memory, and when one process is blocked, the CPU is assigned to another. This cycle ensures that the CPU is always in use. Scheduling, which dictates how the CPU is allocated, is a fundamental function of the operating system and is crucial for managing resources efficiently.

6.1.1 CPU-I/O Burst Cycle

The success of CPU scheduling relies on a key observation about process behavior: process execution alternates between CPU execution and I/O waiting. Each process follows a cycle, beginning with a CPU burst, followed by an I/O burst, then another CPU burst, and so on, until the final CPU burst ends with a system request to terminate (Figure 6.1). The lengths of CPU bursts have been extensively studied, and although they can vary widely between processes and systems, their distribution often follows an exponential or hyperexponential pattern. This means that there are typically many short CPU bursts and fewer long ones. I/O-bound programs tend to have many short CPU bursts, while CPU-bound programs usually have longer CPU bursts.

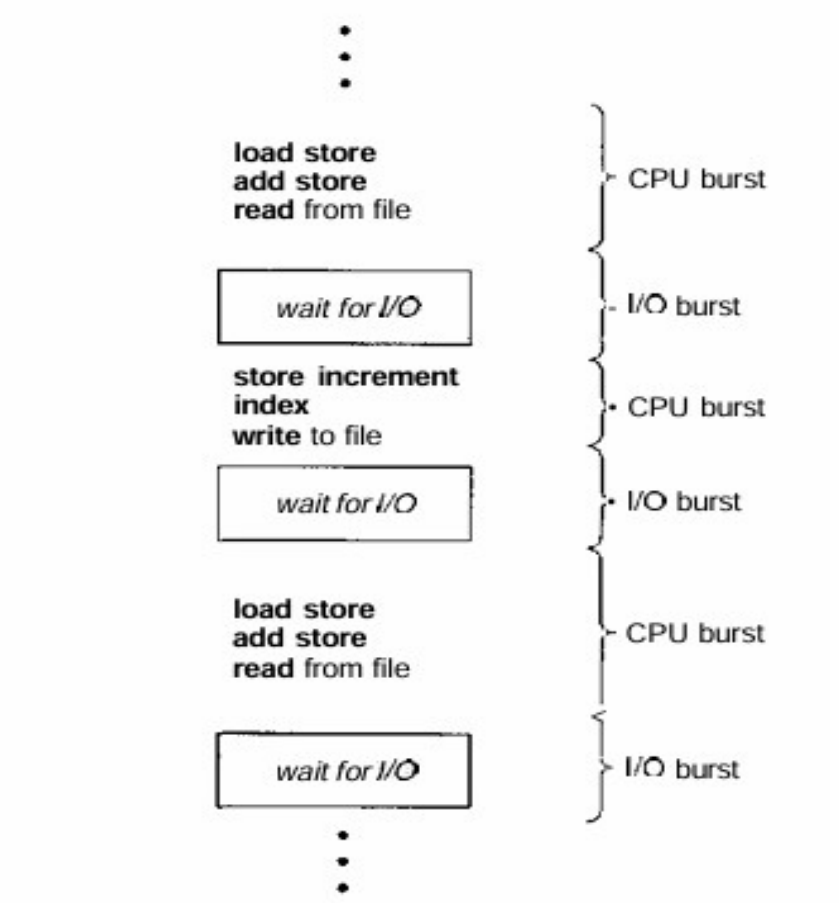


Figure 6.1 Alternating sequence of CPU and I/O bursts.

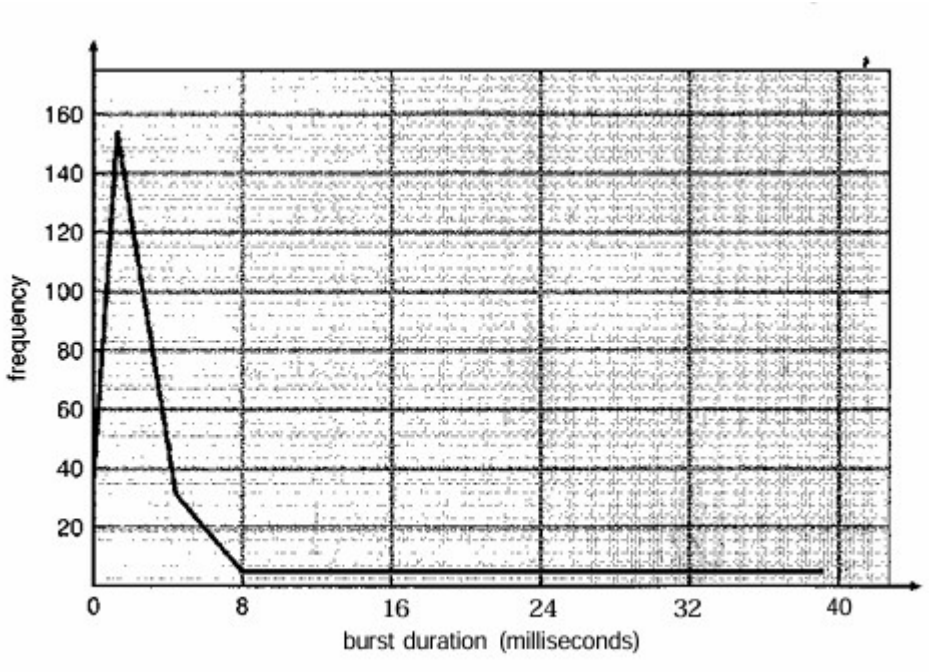


Figure 6.2 Histogram of CPU-burst durations.

A program may have a mix of short and a few long CPU bursts. This distribution is crucial when selecting the most suitable CPU-scheduling algorithm.

6.1.2 CPU Scheduler

When the CPU becomes idle, the operating system must choose one of the processes from the ready queue to execute. This decision is made by the short-term scheduler (or CPU scheduler), which selects a process from those in memory that are ready to run and assigns the CPU to that process.

The ready queue doesn't have to follow a first-in, first-out (FIFO) structure. Depending on the scheduling algorithm used, the ready queue can be implemented as a FIFO queue, priority queue, tree, or an unordered linked list. Conceptually, all the processes in the ready queue are waiting for a chance to use the CPU, and the entries in the queue are typically represented by process control blocks (PCBs) of the processes.

6.1.3 Preemptive Scheduling

CPU scheduling decisions can occur in the following four situations:

1. When a process transitions from the running state to the waiting state, typically due to an I/O request or when waiting for the termination of a child process.
2. When a process transitions from the running state to the ready state, such as when an interrupt occurs.

3. When a process moves from the waiting state to the ready state, for instance, after completing I/O.
4. When a process terminates.

In situations 1 and 4, the scheduling decision is straightforward, as a new process must be selected for execution. However, in situations 2 and 3, there is a choice to be made regarding which process to schedule next.

Scheduling is categorized as either nonpreemptive or preemptive. If scheduling occurs only during situations 1 and 4, it is nonpreemptive, meaning once a process is allocated the CPU, it holds onto it until it either terminates or moves to the waiting state. Nonpreemptive scheduling was used by older versions of Microsoft Windows (3.x) but has since been replaced with preemptive scheduling starting with Windows 95. Preemptive scheduling is also employed by macOS X and modern versions of UNIX. Nonpreemptive scheduling is still useful in environments lacking the hardware necessary for preemptive scheduling.

Preemptive scheduling introduces challenges, particularly concerning access to shared data. For instance, if one process is updating shared data and is preempted, the second process may read inconsistent data. This situation requires mechanisms to ensure data integrity, which is addressed in Chapter 6. Additionally, preemption affects kernel design, particularly when a process is preempted during the execution of system calls. In such cases, operating systems like UNIX ensure consistency by avoiding context switches during critical kernel operations, but this approach can hinder real-time and multiprocessing tasks.

Interrupts, which can occur at any time, further complicate scheduling, as the kernel must handle them promptly. To avoid concurrent access to sensitive code, the system disables interrupts during critical sections and reenables them afterward. These interrupt-disabling sections are typically brief and infrequent but are essential for maintaining system stability.

6.1.4 Dispatcher

Another key component in the CPU scheduling process is the **dispatcher**. The dispatcher is responsible for transferring control of the CPU to the process selected by the short-term scheduler. This function involves three main tasks:

1. **Switching context** – Saving the state of the currently running process and loading the state of the next process.
2. **Switching to user mode** – Changing the processor from kernel mode to user mode.
3. **Jumping to the correct location** in the user program to resume execution.

The dispatcher must be as fast as possible because it is invoked every time the operating system switches between processes. The time it takes for the dispatcher to perform these tasks is referred to as **dispatch latency**.

6.2 SCHEDULING CRITERIA

Different CPU scheduling algorithms come with various properties, and the selection of an algorithm can impact how well the system performs for different types of processes. When choosing an algorithm, it is important to consider several criteria, as each has a significant effect on the algorithm's effectiveness.

Some key criteria for comparing CPU scheduling algorithms include:

- **CPU Utilization**: The goal is to keep the CPU as busy as possible, ideally between 40% for a lightly loaded system and 90% for a heavily loaded system.
- **Throughput**: This measures how much work the system completes, i.e., the number of processes completed per unit of time. For example, this could range from one process per hour for long-running processes to ten processes per second for short tasks.
- **Turnaround Time**: The total time from when a process is submitted to when it is completed, including waiting, CPU execution, and I/O times.
- **Waiting Time**: The time a process spends in the ready queue waiting to be executed, excluding time spent executing or performing I/O.
- **Response Time**: In interactive systems, this is the time from submitting a request to receiving the first response. It focuses on how quickly the system can begin providing output, rather than completing the task.

Ideally, we want to maximize CPU utilization and throughput while minimizing turnaround time, waiting time, and response time. While the average is often optimized, in some cases, it is better to minimize the maximum or minimum values, such as ensuring a good maximum response time for all users. For interactive systems, it may be more important to minimize the variance in response times rather than the average, creating a more predictable user experience.

6.3 SCHEDULING ALGORITHMS

CPU scheduling addresses the challenge of determining which process in the ready queue should be allocated the CPU. Various CPU scheduling algorithms exist to solve this problem, each with different strategies and objectives. In this section, we will explore several of these algorithms and their characteristics.

6.3.1 First-Come, First-Served Scheduling

First-Come, First-Served (FCFS) scheduling is the simplest CPU scheduling algorithm, where processes are executed in the order they arrive in the ready queue. It is a **non-preemptive scheduling method**, meaning that once a process starts execution, it runs to completion without interruption.

Features of FCFS Scheduling

- **Execution Order**: Processes are scheduled based on their arrival time.
- **Non-Preemptive**: A running process cannot be interrupted by another process.
- **Implementation**: The ready queue is organized as a FIFO (First-In, First-Out) queue.
- **Fair but Inefficient**: Every process gets a turn, but longer processes can delay shorter ones.

Example of FCFS Scheduling

Process Details:

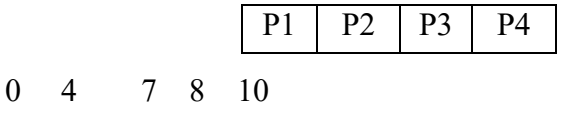
Process	Arrival Time (AT)	Burst Time (BT)
P1	0 ms	4 ms
P2	1 ms	3 ms
P3	2 ms	1 ms
P4	3 ms	2 ms

Step 1: Create the Execution Order

- Based on arrival time: P1 → P2 → P3 → P4.

Step 2: Draw the Gantt Chart

A Gantt chart illustrates the execution sequence and completion times of the processes:



Step 3: Calculate Metrics

1. **Completion Time (CT)**: Time when a process finishes execution.
   - P1: 0 + 4 = 4 ms
   - P2: 4 + 3 = 7 ms
   - P3: 7 + 1 = 8 ms
   - P4: 8 + 2 = 10 ms
2. **Turnaround Time (TAT)**:
   TAT=CT−AT
   - P1: 4 - 0 = 4 ms
   - P2: 7 - 1 = 6 ms
   - P3: 8 - 2 = 6 ms

- P4: 10 - 3 = 7 ms

3.**Waiting Time (WT)**:
WT=TAT−BT

- P1: 4 - 4 = 0 ms
- P2: 6 - 3 = 3 ms
- P3: 6 - 1 = 5 ms
- P4: 7 - 2 = 5 ms

Table

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0 ms	4 ms	4 ms	4 ms	0 ms
P2	1 ms	3 ms	7 ms	6 ms	3 ms
P3	2 ms	1 ms	8 ms	6 ms	5 ms
P4	3 ms	2 ms	10 ms	7 ms	5 ms

Average Turnaround Time:

$$\frac{4+6+6+7}{4} = 5.75\,\text{ms}$$

Average Waiting Time:

$$\frac{0+3+5+5}{4} = 3.25\,\text{ms}$$

Advantages of FCFS

1. **Simplicity**: Easy to understand and implement.
2. **Fairness**: Every process gets executed in the order of arrival.

Disadvantages of FCFS

1. **Convoy Effect**: A long process can block the queue, causing delays for shorter processes.
2. **Poor Utilization**: Longer waiting times for processes arriving later.
3. **Not Suitable for Interactive Systems**: Response times can be too high for real-time or time-sharing systems.

FCFS is a foundational CPU scheduling algorithm suitable for batch systems but is often replaced by more efficient algorithms like Shortest Job Next (SJN) or Round Robin (RR) in modern operating systems.

6.3.2 Shortest-Job-First Scheduling

Shortest-Job-First (SJF) is a CPU scheduling algorithm that selects the process with the smallest burst time for execution. It minimizes waiting time and is widely regarded as an optimal scheduling technique for reducing average waiting and turnaround times.

SJF can be implemented in two forms:

1. **Non-Preemptive SJF**: Once a process starts, it runs to completion.
2. **Preemptive SJF** (also known as **Shortest Remaining Time First**, SRTF): If a new process arrives with a shorter burst time than the remaining time of the current process, the CPU is preempted.

Characteristics of SJF

1. **Priority**: Shorter burst time processes are given priority.
2. **Optimality**: It minimizes the average waiting time and turnaround time.
3. **Complexity**: Requires knowledge of burst times in advance, which is difficult in real-world scenarios.

Example of Non-Preemptive SJF Scheduling

Given Processes:

Process	Arrival Time (AT)	Burst Time (BT)
P1	0 ms	7 ms
P2	2 ms	4 ms
P3	4 ms	1 ms
P4	5 ms	4 ms

Step 1: Arrange Processes by Arrival Time

Initially, only P1 is available. When P2, P3, and P4 arrive, the process with the shortest burst time is selected.

Step 2: Execute Processes

1. P1 executes first (no other process available at 0 ms).
2. At 7 ms, P2, P3, and P4 are ready. P3 (1 ms) executes next.
3. After P3, the next shortest burst time is P2 (4 ms).
4. Finally, P4 executes.

Gantt Chart

P1	P3	P2	P4

0 7 8 12 16

Step 3: Calculate Metrics

1. **Completion Time (CT)**:
   - P1: 0 + 7 = 7 ms
   - P3: 7 + 1 = 8 ms
   - P2: 8 + 4 = 12 ms
   - P4: 12 + 4 = 16 ms
   - **Turnaround Time (TAT)**:
     TAT=CT−
   - P1: 7 - 0 = 7 ms
   - P2: 12 - 2 = 10 ms
   - P3: 8 - 4 = 4 ms
   - P4: 16 - 5 = 11 ms
2. **Waiting Time (WT)**:
   WT=TAT−BT
   - P1: 7 - 7 = 0 ms
   - P2: 10 - 4 = 6 ms
   - P3: 4 - 1 = 3 ms
   - P4: 11 - 4 = 7 ms

Table

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0 ms	7 ms	7 ms	7 ms	0 ms
P2	2 ms	4 ms	12 ms	10 ms	6 ms
P3	4 ms	1 ms	8 ms	4 ms	3 ms
P4	5 ms	4 ms	16 ms	11 ms	7 ms

Average Turnaround Time:

$$\frac{7+10+4+11}{4} = 8\,\text{ms}$$

Average Waiting Time:

$$\frac{0+6+3+7}{4} = 4\,\text{ms}$$

Advantages of SJF

1. **Optimal for Waiting Time**: SJF minimizes the average waiting time.
2. **Efficient**: Reduces turnaround time for shorter processes.

Disadvantages of SJF

1. **Starvation**: Longer processes may experience indefinite delays.
2. **Uncertainty**: Exact burst times are often difficult to predict.
3. **Not Suitable for Time-Sharing**: Non-preemptive SJF cannot handle interactive systems effectively.

SJF scheduling is a highly efficient algorithm for batch systems where burst times are predictable. However, its practical implementation is limited due to the challenges of estimating burst times and potential starvation of longer processes. To overcome these drawbacks, **Shortest Remaining Time First (SRTF)** or other dynamic scheduling methods are often preferred in modern systems.

6.3.3 Priority Scheduling

Priority Scheduling is a CPU scheduling algorithm that assigns a priority level to each process. The process with the highest priority is executed first. Priority can be determined based on various factors such as process type, resource requirements, or importance.

The algorithm can be implemented in two ways:

1. **Preemptive Priority Scheduling**: If a new process arrives with a higher priority than the currently executing process, it preempts the CPU.
2. **Non-Preemptive Priority Scheduling**: The currently executing process completes its execution before the CPU is allocated to a higher-priority process

Characteristics of Priority Scheduling

1. **Priority-Based**: Processes with higher priority are executed before lower-priority processes.
2. **Preemptive and Non-Preemptive Variants**: Offers flexibility in implementation.

3. **Starvation**: Lower-priority processes may wait indefinitely if higher-priority processes continuously arrive.
4. **Aging**: A technique to prevent starvation by gradually increasing the priority of waiting processes.

Example of Non-Preemptive Priority Scheduling

Given Processes:

Process	Arrival Time (AT)	Burst Time (BT)	Priority (Lower Value = Higher Priority)
P1	0 ms	7 ms	2
P2	1 ms	4 ms	1
P3	2 ms	1 ms	3
P4	3 ms	4 ms	2

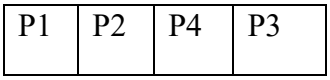
Step 1: Arrange by Priority

- Process with the lowest priority value executes first.
- If multiple processes have the same priority, the tie is resolved using arrival time.

Step 2: Execute Processes

1. P1 starts execution at time 0.
2. At 1 ms, P2 arrives with a higher priority and executes next after P1.
3. P3 and P4 execute according to priority order.

Gantt Chart



0 7 11 15 16

Step 3: Calculate Metrics

1. **Completion Time (CT)**:
   - P1: 7 ms
   - P2: 11 ms
   - P3: 16 ms
   - P4: 15 ms
2. **Turnaround Time (TAT)**:
   TAT=CT−AT
   - P1: 7 - 0 = 7 ms
   - P2: 11 - 1 = 10 ms
   - P3: 16 - 2 = 14 ms

- P4: 15 - 3 = 12 ms

3. **Waiting Time (WT)**:
   WT=TAT−BT
   - P1: 7 - 7 = 0 ms
   - P2: 10 - 4 = 6 ms
   - P3: 14 - 1 = 13 ms
   - P4: 12 - 4 = 8 ms

Table

Process	Arrival Time	Burst Time	Priority	Completion Time	Turnaround Time	Waiting Time
P1	0 ms	7 ms	2	7 ms	7 ms	0 ms
P2	1 ms	4 ms	1	11 ms	10 ms	6 ms
P3	2 ms	1 ms	3	16 ms	14 ms	13 ms
P4	3 ms	4 ms	2	15 ms	12 ms	8 ms

Average Turnaround Time:

$$\frac{7+10+14+12}{4} = 10.75\,\text{ms}$$

Average Waiting Time:

$$\frac{0+6+13+8}{4} = 6.75\,\text{ms}$$

Advantages of Priority Scheduling

1. **Flexibility**: Allows differentiation between processes based on their importance.
2. **Efficient Resource Allocation**: Critical tasks can be executed sooner.

Disadvantages of Priority Scheduling

1. **Starvation**: Low-priority processes may never execute if higher-priority processes keep arriving.
2. **Complexity**: Determining and assigning accurate priorities can be challenging.
3. **Unfairness**: May favor specific processes excessively.

Solution to Starvation: Aging

Aging is a technique that gradually increases the priority of processes waiting in the ready queue for a long time. This ensures that lower-priority processes eventually get executed.

Priority Scheduling is a versatile algorithm, particularly suitable for real-time systems and scenarios where certain processes must be prioritized. However, to avoid starvation and ensure fairness, mechanisms like aging are essential.

6.3.4 Round-Robin Scheduling

Round-Robin (RR) is a CPU scheduling algorithm designed for time-sharing systems. In this method, each process is assigned a fixed time slice or **time quantum**. The CPU cycles through all processes in the ready queue, executing each process for a maximum of one time quantum before moving to the next process. If a process does not finish within its assigned time quantum, it is placed at the end of the queue.

Characteristics of Round-Robin Scheduling

1. **Time Quantum**: Each process gets an equal time slice for execution.
2. **Preemptive**: If a process does not finish within its time quantum, it is preempted and re-queued.
3. **Fairness**: All processes are treated equally, preventing starvation.
4. **Efficient for Time-Sharing**: Ideal for interactive and real-time systems due to quick response times.

How Round-Robin Works

1. **Ready Queue Management**: The processes are maintained in a circular queue.
2. **CPU Allocation**: Each process gets the CPU for a duration up to the time quantum.
3. **Preemption**: If a process's burst time exceeds the time quantum, the remaining time is added back to the queue.

Example of Round-Robin Scheduling

Given Processes:

Process	Arrival Time (AT)	Burst Time (BT)
P1	0 ms	5 ms
P2	1 ms	4 ms
P3	2 ms	2 ms
P4	3 ms	1 ms

Time Quantum: 2 ms

Step 1: Create the Execution Order

1. Start with the first process in the ready queue.

2. Allocate the CPU for a maximum of the time quantum.
3. If a process completes within its time quantum, it exits the queue.
4. If not, it re-enters the queue with its remaining burst time.

Gantt Chart

P1	P2	P3	P4	P1	P2	P1

0 2 4 6 7 9 11 12

Step 2: Calculate Metrics

1. **Completion Time (CT)**:
   - P1: 12 ms
   - P2: 9 ms
   - P3: 6 ms
   - P4: 7 ms
2. **Turnaround Time (TAT)**:
   TAT=CT−AT
   - P1: 12 - 0 = 12 ms
   - P2: 9 - 1 = 8 ms
   - P3: 6 - 2 = 4 ms
   - P4: 7 - 3 = 4 ms
3. **Waiting Time (WT)**:
   WT=TAT−BT
   - P1: 12 - 5 = 7 ms
   - P2: 8 - 4 = 4 ms
   - P3: 4 - 2 = 2 ms
   - P4: 4 - 1 = 3 ms

Table

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0 ms	5 ms	12 ms	12 ms	7 ms
P2	1 ms	4 ms	9 ms	8 ms	4 ms
P3	2 ms	2 ms	6 ms	4 ms	2 ms
P4	3 ms	1 ms	7 ms	4 ms	3 ms

Average Turnaround Time:

$$\frac{12+8+4+4}{4} = 7\,\text{ms}$$

Average Waiting Time:

$$\frac{7+4+2+3}{4} = 4\,\text{ms}$$

Advantages of Round-Robin Scheduling

1. **Fairness**: All processes are given equal time.
2. **Prevention of Starvation**: No process waits indefinitely for the CPU.
3. **Interactive Systems**: Ideal for time-sharing systems due to regular context switches.

Disadvantages of Round-Robin Scheduling

1. **Overhead**: Frequent context switching can increase CPU overhead.
2. **Response Time Dependence**: Performance depends on the size of the time quantum.
    - A **small quantum** increases context switches and overhead.
    - A **large quantum** reduces responsiveness and can mimic First-Come, First-Served (FCFS).

Round-Robin scheduling is a widely used algorithm, particularly for interactive and time-sharing systems. It balances fairness and responsiveness but requires careful selection of the time quantum to optimize performance.

6.3.5 Multilevel Queue Scheduling

Multilevel Queue Scheduling is a CPU scheduling algorithm where the ready queue is divided into multiple distinct queues. Each queue is designed for a specific category of processes, such as system processes, interactive processes, or batch processes. Each queue can have its own scheduling algorithm, and a priority is assigned to decide the order of queue execution.

Characteristics of Multilevel Queue Scheduling

1. **Queue Classification**: Processes are categorized into different queues based on their type (e.g., system, user, background).
2. **Fixed Priority**: Queues are executed based on priority. Higher-priority queues preempt lower-priority queues.
3. **Separate Scheduling**: Each queue can use a different scheduling algorithm (e.g., Round-Robin, FCFS).
4. **Non-Preemptive or Preemptive**: Execution can either be non-preemptive (lower-priority queues wait until higher-priority queues finish) or preemptive.

Structure of Multilevel Queue Scheduling

- **System Queue**: Reserved for system-critical processes.
- **Interactive Queue**: For processes requiring user interaction.
- **Batch Queue**: For background or long-running processes.

Example of Multilevel Queue Scheduling

Given Queues:

1. **System Queue (Priority 1)**: Uses Round-Robin Scheduling.
2. **Interactive Queue (Priority 2)**: Uses First-Come, First-Served Scheduling.
3. **Batch Queue (Priority 3)**: Uses First-Come, First-Served Scheduling.

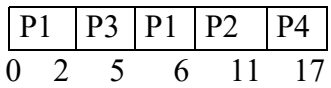
Processes:

Process	Queue	Arrival Time (AT)	Burst Time (BT)
P1	System Queue	0 ms	4 ms
P2	Interactive Queue	1 ms	5 ms
P3	System Queue	2 ms	3 ms
P4	Batch Queue	3 ms	6 ms

Step 1: Process Execution Based on Queue Priority

1. **System Queue** (Priority 1) executes P1 and P3 using Round-Robin.
2. After all System Queue processes are completed, the **Interactive Queue** (Priority 2) executes P2.
3. Finally, the **Batch Queue** (Priority 3) executes P4.

Gantt Chart



Step 2: Calculate Metrics

1. **Completion Time (CT)**:
   - P1: 6 ms
   - P2: 11 ms
   - P3: 5 ms
   - P4: 17 ms
2. **Turnaround Time (TAT)**:
   TAT=CT−AT
   - P1: 6 - 0 = 6 ms
   - P2: 11 - 1 = 10 ms
   - P3: 5 - 2 = 3 ms
   - P4: 17 - 3 = 14 ms
3. **Waiting Time (WT)**:
   WT=TAT−BT
   - P1: 6 - 4 = 2 ms
   - P2: 10 - 5 = 5 ms

- P3: 3 - 3 = 0 ms
- P4: 14 - 6 = 8 ms

Table

Process	Queue	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	System Queue	0 ms	4 ms	6 ms	6 ms	2 ms
P2	Interactive Queue	1 ms	5 ms	11 ms	10 ms	5 ms
P3	System Queue	2 ms	3 ms	5 ms	3 ms	0 ms
P4	Batch Queue	3 ms	6 ms	17 ms	14 ms	8 ms

Average Turnaround Time:

$$\frac{6 + 10 + 3 + 14}{4} = 8.25\,\text{ms}$$

Average Waiting Time:

$$\frac{2 + 5 + 0 + 8}{4} = 3.75\,\text{ms}$$

Advantages of Multilevel Queue Scheduling

1. **Efficient Process Categorization**: Processes are grouped based on their type or importance.
2. **Flexibility**: Different queues can have different scheduling algorithms to suit specific needs.
3. **Priority Control**: High-priority tasks are executed sooner.

Disadvantages of Multilevel Queue Scheduling

1. **Starvation**: Lower-priority queues may experience indefinite delays if higher-priority queues are always occupied.
2. **Complexity**: Managing multiple queues and scheduling rules can be challenging.
3. **Rigid Classification**: Once assigned, a process cannot move between queues.

Solution to Starvation: Aging

To mitigate starvation, the priority of long-waiting processes in lower-priority queues can be increased over time.

Multilevel Queue Scheduling is ideal for systems with diverse process requirements, such as interactive and batch processing systems. While it provides

flexibility and prioritization, it requires careful configuration to avoid starvation and inefficiency.

6.3.6 Multilevel Feedback-Queue Scheduling

Multilevel Feedback Queue Scheduling is an advanced CPU scheduling algorithm that allows processes to move between multiple queues based on their behavior and requirements. Unlike Multilevel Queue Scheduling, where processes are permanently assigned to a queue, the feedback mechanism in this algorithm adapts dynamically, promoting or demoting processes among queues based on their execution characteristics.

Features of Multilevel Feedback Queue Scheduling

1. **Multiple Queues**: Similar to multilevel queue scheduling, there are multiple queues, each with a specific priority and scheduling algorithm.
2. **Dynamic Adjustment**: Processes can move between queues based on their CPU burst times. Shorter processes are prioritized, while longer ones may be moved to lower-priority queues.
3. **Aging**: This prevents starvation by gradually increasing the priority of processes that have waited too long.
4. **Preemptive**: The scheduler can preempt lower-priority processes if a higher-priority process arrives.

Structure of Multilevel Feedback Queue Scheduling

1. **Top Queue (Highest Priority)**: Uses Round-Robin with a small time quantum to handle interactive processes.
2. **Middle Queue**: Uses Round-Robin with a larger time quantum to accommodate processes requiring longer execution.
3. **Bottom Queue (Lowest Priority)**: Uses First-Come, First-Served (FCFS) to handle batch or long-running processes.

Rules for Moving Between Queues

1. A process starts in the highest-priority queue.
2. If a process uses its entire time quantum without completing, it is moved to a lower-priority queue.
3. If a process waits too long in a lower-priority queue, it may be promoted to a higher-priority queue (aging).

Example of Multilevel Feedback Queue Scheduling

Assume 3 Queues:

1. **Queue 1 (Priority 1)**: Round-Robin, Time Quantum = 4 ms
2. **Queue 2 (Priority 2)**: Round-Robin, Time Quantum = 8 ms
3. **Queue 3 (Priority 3)**: FCFS

Processes:

Process	Arrival Time (AT)	Burst Time (BT)
P1	0 ms	20 ms
P2	1 ms	6 ms
P3	2 ms	4 ms
P4	3 ms	10 ms

Execution Process

1. **Queue 1 Execution**:
   - At time 0, P1 starts in Queue 1. After 4 ms, it is preempted and moved to Queue 2.
   - P2 arrives at time 1 and executes for 4 ms. It completes within its time quantum and exits.
   - P3 arrives at time 2 and executes for 4 ms. It completes and exits.
   - P4 arrives at time 3 and executes for 4 ms. It is preempted and moved to Queue 2.
2. **Queue 2 Execution**:
   - P1 resumes execution for 8 ms but still does not complete. It is preempted and moved to Queue 3.
   - P4 resumes execution for 6 ms and completes.
3. **Queue 3 Execution**:
   - P1 executes in FCFS until completion (remaining 8 ms).

Gantt Chart

P1	P2	P3	P4	P1	P4	P1

0 4 8 12 16 24 30 38

Step 2: Calculate Metrics

1. **Completion Time (CT)**:
   - P1: 38 ms
   - P2: 8 ms
   - P3: 12 ms
   - P4: 30 ms

2. **Turnaround Time (TAT)**:
   TAT=CT−AT
   - P1: 38 - 0 = 38 ms
   - P2: 8 - 1 = 7 ms
   - P3: 12 - 2 = 10 ms
   - P4: 30 - 3 = 27 ms
3. **Waiting Time (WT)**:
   WT=TAT−BT
   - P1: 38 - 20 = 18 ms
   - P2: 7 - 6 = 1 ms
   - P3: 10 - 4 = 6 ms
   - P4: 27 - 10 = 17 ms

Table

Process	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	0 ms	20 ms	38 ms	38 ms	18 ms
P2	1 ms	6 ms	8 ms	7 ms	1 ms
P3	2 ms	4 ms	12 ms	10 ms	6 ms
P4	3 ms	10 ms	30 ms	27 ms	17 ms

Average Turnaround Time:

$$\frac{38 + 7 + 10 + 27}{4} = 20.5\,\text{ms}$$

Average Waiting Time:

$$\frac{18 + 1 + 6 + 17}{4} = 10.5\,\text{ms}$$

Advantages of Multilevel Feedback Queue Scheduling

1. **Adaptability**: Dynamically adjusts process priorities based on behavior.
2. **Fairness**: Shorter processes get higher priority, ensuring quick execution.
3. **Prevention of Starvation**: Aging mechanism prevents indefinite delays for lower-priority processes.

Disadvantages of Multilevel Feedback Queue Scheduling

1. **Complexity**: Managing multiple queues and dynamic process movement is challenging.
2. **Overhead**: Frequent context switches increase CPU overhead.
3. **Parameter Sensitivity**: Performance depends on the number of queues, time quanta, and aging mechanisms.

Multilevel Feedback Queue Scheduling is a versatile and efficient algorithm suitable for systems with diverse process requirements. Its adaptability and fairness make it ideal for multitasking environments, though careful configuration is necessary to balance performance and overhead.

6.4 SUMMARY

Scheduling in operating systems involves managing the execution of processes to optimize CPU utilization and system performance. The **CPU-I/O Burst Cycle** describes how processes alternate between CPU bursts (computation) and I/O bursts (waiting for input/output). The **CPU Scheduler** decides which process gets CPU time next, while **Preemptive Scheduling** allows the OS to interrupt running processes to prioritize others. The **Dispatcher** handles context switching, transferring control between processes. **Scheduling Criteria** such as CPU utilization, throughput, waiting time, and response time help evaluate scheduling effectiveness. Various **Scheduling Algorithms** like FCFS, SJF, RR, and Priority Scheduling optimize performance by using different strategies for process execution.

6.5 KEY TERMS

Switching context, dispatch latency**,** CPU Utilization, Throughput, Turnaround Time,Waiting Time, Response Time,Execution Order, Non-Preemptive, Implementation, Fair but Inefficient, Priority, Optimality, Complexity, Starvation, Aging, CPU Allocation.

6.6 SELF-ASSESSMENT QUESTIONS

1. What is the CPU-I/O burst cycle, and why is it important for process scheduling?
2. What is the role of the CPU scheduler in an operating system?
3. How does preemptive scheduling differ from non-preemptive scheduling?
4. What is the function of the dispatcher in process scheduling?
4. What are the main factors considered in scheduling criteria for CPU scheduling algorithms?

6.7 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”, Seventh Edition, Wiley.
2. William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3. Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4. Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5. Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR
Dr. Vasantha Rudramalla

LESSON 7
MULTIPLE-PROCESSOR SCHEDULING

Aim and Objectives:

- Efficiently manage task allocation, processor utilization, and thread execution across multiple CPUs using advanced scheduling strategies, load balancing, and processor affinity.
- Enhance performance through symmetric multithreading and optimize system responsiveness using effective thread scheduling techniques.
- Analyze and validate scheduling algorithms with queueing models and simulations to ensure fairness, scalability, and high throughput.

Structure:

7.1 Introduction

In earlier discussions, we examined the challenges of CPU scheduling in single-processor systems. The introduction of multiple CPUs enables load sharing, offering potential improvements in performance and efficiency. However, this also adds complexity to the scheduling process. Just as in single-processor systems, no universally optimal solution exists, and various strategies have been explored. This section focuses on the key issues in multiprocessor scheduling, emphasizing systems with homogeneous processors—where all CPUs are functionally identical and can

execute any process in the queue. Despite this homogeneity, certain constraints may arise, such as when specific I/O devices are tied to a particular processor, necessitating process scheduling on that specific CPU.

7.1.1 Approaches to Multiple-Processor Scheduling

CPU scheduling in multiprocessor systems can follow two main approaches. In **asymmetric multiprocessing**, a single processor (master server) handles all scheduling, I/O processing, and system tasks, while other processors execute only user code. This simplifies system design by reducing shared data access. Alternatively, in **symmetric multiprocessing (SMP)**, each processor is self-scheduling, using either a shared ready queue or private queues. In SMP, careful programming is required to ensure proper synchronization and prevent issues like process duplication or loss from queues. Most modern operating systems, including Windows, Solaris, Linux, and macOS, support SMP. This section delves deeper into SMP-related considerations.

7.1.2 Processor Affinity

When a process runs on a specific processor, its recently accessed data is stored in the processor's cache, allowing subsequent memory accesses to be quickly satisfied. However, if the process migrates to another processor, the cache of the original processor must be invalidated, and the new processor's cache must be repopulated, resulting in a costly overhead. To minimize this expense, most SMP systems aim to keep a process on the same processor whenever possible. This strategy is known as **processor affinity**, where a process favors staying on the processor it is currently running on.

Processor affinity is classified into two types:

1. **Soft Affinity**: The operating system tries to keep a process on the same processor but does not strictly enforce this, allowing migrations when necessary.
2. **Hard Affinity**: Supported by some systems, like Linux, through system calls, it allows processes to specify strict rules preventing migration to other processors.

Processor affinity enhances performance by maintaining cache efficiency and reducing migration costs.

7.1.3 Load Balancing

In SMP (Symmetric Multiprocessing) systems, it is crucial to balance the workload across all processors to maximize the benefits of multiple processors. Without load balancing, some processors may remain idle while others become overloaded with tasks, causing inefficiency. **Load balancing** ensures that the workload is distributed evenly among all processors, especially in systems where each processor has its own private queue of processes. In systems with a common run queue, load balancing may not be necessary, as

idle processors can immediately fetch tasks from the shared queue. However, most modern operating systems with SMP do employ private queues for each processor.

There are two main approaches to load balancing:

1. **Push Migration**: A task periodically checks the load on each processor and, if an imbalance is found, moves processes from overloaded processors to less-busy or idle ones.
2. **Pull Migration**: An idle processor pulls tasks from a busy processor to balance the load.

These approaches can be used together, and many modern operating systems, like Linux and FreeBSD, implement both methods. For example, Linux runs its load-balancing algorithm every 200 milliseconds (push migration) and when a processor's run queue is empty (pull migration).

However, load balancing can conflict with **processor affinity** (discussed earlier), which benefits from keeping a process on the same processor to take advantage of cached data. Moving a process between processors through load balancing can invalidate this benefit. In practice, there is no one-size-fits-all approach; some systems prioritize load balancing by always moving processes when an imbalance occurs, while others only perform migration when the load imbalance exceeds a certain threshold.

7.1.4 Symmetric Multithreading

SMP systems allow multiple threads to run concurrently using several physical processors. An alternative approach is **Symmetric Multithreading (SMT)**, also known as **hyper threading** on Intel processors. SMT enables multiple logical processors to share a single physical processor, presenting several logical processors to the operating system, even on a single physical processor. Each logical processor has its own architecture state, including registers and interrupt handling, but shares resources like cache and buses with other logical processors on the same physical core.

SMT is implemented in hardware, not software, and operating systems typically don't need major changes to run on SMT systems. However, performance can be improved if the OS is designed to optimize the use of logical processors. For example, when two physical processors are idle, the OS should prefer scheduling threads on separate physical processors rather than on separate logical processors of the same physical core to avoid overloading a single processor.

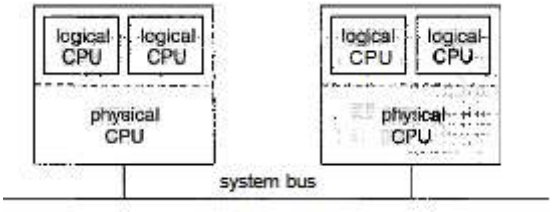


Figure 7.1 A typical SMT architecture

7.2 THREAD SCHEDULING

We introduced threads into the process model, distinguishing between **user-level threads** and **kernel-level threads**. In operating systems that support both, it is the kernel-level threads that are scheduled by the operating system, not the processes. User-level threads, on the other hand, are managed by a thread library, and the kernel is unaware of them. For user-level threads to run on a CPU, they must eventually be mapped to a corresponding kernel-level thread, often using a lightweight process (LWP) as an intermediary. This section delves into the scheduling issues related to both user-level and kernel-level threads, with specific examples for **Pthreads** scheduling.

7.2.1 Contention Scope

A key difference between **user-level** and **kernel-level threads** is how they are scheduled. In systems using the **many-to-one** and **many-to-many** models, the thread library schedules user-level threads onto available **lightweight processes (LWPs)**, which is called **process-contention scope (PCS)**. Here, competition for the CPU happens among threads within the same process. However, the operating system is responsible for scheduling the associated **kernel-level thread** onto a physical CPU using **system-contention scope (SCS)**, where competition for the CPU occurs among all threads in the system.

In **one-to-one** models (e.g., Windows XP, Solaris 9, Linux), only SCS is used. **PCS** typically schedules threads based on priority, selecting the highest-priority runnable thread, though **user-level thread priorities** are set by the programmer and are not adjusted by the thread library. PCS may preempt running threads if a higher-priority thread becomes available, but there is no guarantee of **time slicing** for threads with equal priority.

7.2.2 Pthread Scheduling

Processor scheduling is a crucial component of operating systems, ensuring efficient allocation of CPU time among competing tasks. POSIX Threads (Pthreads) enhance this by allowing thread-level scheduling, offering flexibility and control over how

threads are prioritized and executed. The Pthread library provides tools to manage thread scheduling policies and priorities.

Key Concepts in Pthread Scheduling

1. **Thread Scheduling Policies**
   Pthreads support several scheduling policies that define how threads are prioritized and executed:
   - **SCHED_FIFO (First-In, First-Out):** Real-time threads are scheduled in the order they are ready to run, and they execute until they either finish or are preempted by a higher-priority thread.
   - **SCHED_RR (Round-Robin):** Similar to SCHED_FIFO, but with a time quantum. After the quantum expires, the thread is placed at the end of the queue if it is still ready to run.
   - **SCHED_OTHER:** Default policy for non-real-time threads, usually time-sharing based.
2. **Thread Priority**

   Each thread has an associated priority. Higher-priority threads are given preference by the scheduler, but priority ranges depend on the specific policy.

3. **Setting and Getting Scheduling Parameters**
   - **pthread_setschedparam()**: Sets a thread's scheduling policy and priority.
   - **pthread_getschedparam()**: Retrieves a thread's current scheduling policy and priority.

Example: Pthread Scheduling in Processor Scheduling

Below is an example of a Pthread program that demonstrates thread scheduling using the **SCHED_RR**policy:

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#include<sched.h>
#include<unistd.h>

// Function executed by threads
void* thread_function(void* arg) {
intthread_num = *((int*)arg);
printf("Thread %d is running\n", thread_num);
sleep(2);  // Simulate work
printf("Thread %d is finishing\n", thread_num);
returnNULL;
}
```

```
intmain() {
constint NUM_THREADS = 3;
pthread_t threads[NUM_THREADS];
pthread_attr_tattr;
structsched_paramsched_param;
int ret;

// Initialize thread attributes
pthread_attr_init(&attr);

// Set scheduling policy to SCHED_RR
pthread_attr_setschedpolicy(&attr, SCHED_RR);

// Set thread priority
sched_param.sched_priority = 10;
pthread_attr_setschedparam(&attr, &sched_param);

// Enable explicit scheduling
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);

// Create threads
for (inti = 0; i< NUM_THREADS; i++) {
int* thread_num = malloc(sizeof(int));
      *thread_num = i + 1;
ret = pthread_create(&threads[i], &attr, thread_function, thread_num);
if (ret) {
perror("pthread_create");
exit(EXIT_FAILURE);
      }
   }

// Wait for threads to finish
for (inti = 0; i< NUM_THREADS; i++) {
pthread_join(threads[i], NULL);
   }

// Destroy thread attributes
pthread_attr_destroy(&attr);

printf("All threads have completed.\n");
return0;
}
```

Explanation of the Program

1. **Thread Attributes:** The pthread_attr_t structure is initialized, and the scheduling policy is set to **SCHED_RR** using pthread_attr_setschedpolicy(). The priority is specified via the sched_param structure.
2. **Explicit Scheduling:** By default, threads inherit their scheduling policy from the process. Using pthread_attr_setinheritsched(), we enable explicit scheduling, ensuring threads use the specified policy and priority.
3. **Thread Execution**: The thread_function() simulates work by sleeping for a few seconds. Threads are scheduled based on the **round-robin** policy with the specified priority.
4. **Thread Synchronization:**The pthread_join() function ensures the main thread waits for all worker threads to complete before exiting.

- **Real-Time Scheduling:**Policies like SCHED_FIFO and SCHED_RR are often used in real-time systems where predictable execution is critical. However, these require root privileges on most systems.
- **System-Specific Behavior:**The availability and behavior of scheduling policies and priorities can vary between operating systems. For example, Linux supports real-time policies, but they are limited to processes with elevated privileges.
- **Error Handling**:Always check the return values of Pthread functions to handle errors effectively, such as permission issues or invalid attribute settings.

This example illustrates how Pthread scheduling integrates with processor scheduling to optimize thread execution based on specific policies and priorities

7.3 OPERATING SYSTEM EXAMPLES

Next, we will describe the scheduling policies of the **Solaris**, **Windows XP**, and **Linux** operating systems. It's important to note that we are focusing on the scheduling of kernel threads for both **Solaris** and **Linux**. Since **Linux** does not distinguish between processes and threads, we will use the term **task** when discussing the **Linux scheduler**.
Example: Solaris Scheduling

Solaris employs priority-based thread scheduling and defines four scheduling classes, ranked by priority as follows:

1. Real-time
2. System
3. Time-sharing
4. Interactive

Each class has its own set of priorities and scheduling algorithms. The scheduling structure in **Solaris** is illustrated in Figure 7.3

```
#include <pthread.h> ?
tinclude<stdio.h>
#define NUM.THREADS 5
i n t main(intargc, char *argv[])
{
i n t i , scope;
pthread_ttid [NUMJTHREADS] ;
pthread_attr_tattr;
/* get the default attributes */
pthread_attr_init (&attr) ;
/* first inquire on the current scope */
if (pthread_attr_getscope(fcattr, kscope) != 0)
fprintf(stderr, "Unable to get scheduling scope\n");
else {
if (scope == PTHREAD.SCOPE.PROCESS)
printf ( "PTHREAD_SCOPE_PROCESS" ) ;
else if (scope == PTHREAD.SCOPE.SYSTEM)
printf ( " PTHREAD_SCOPE_SYSTEM") ;
else
fprintf(stderr, "Illegal scope value.\n");
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope (&attr, PTHREAD^SCOPE.SYSTEM)
/* create the threads */
for (i = 0; i< NUM_THREADS; i++)
pthread^create (&tid [i] , &attr, runner,NULL) ;
/* now join on each thread */
for (i = 0; i< NUMJTHREADS; i++)
pthread^join (tid [i] , NULL);
/* Each thread will begin control in this function */
void *runner(void *param)
{ /* do some work ... */
Dthread_exitfO) ;
```

Figure 7.2 Pthread scheduling API.

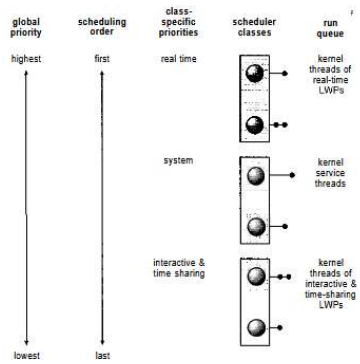


Figure 7.3 Solaris scheduling

The default scheduling class in **Solaris** is **time-sharing**. This policy dynamically adjusts priorities and assigns time slices of varying lengths using a **multilevel feedback queue**. There is an inverse relationship between priorities and time slices: higher priorities get shorter time slices, while lower priorities get longer ones. **Interactive processes** typically have higher priority, while **CPU-bound processes** have lower priority. This ensures good response times for interactive processes and high throughput for CPU-bound processes. The **interactive class** uses the same policy as the time-sharing class but gives windowing applications higher priority for better performance. The dispatch table in **Figure 7.4** shows the scheduling for both classes, with 60 priority levels (only a few are displayed for simplicity).

Priority: The class-dependent priority for the time-sharing and interactive classes. A higher number indicates a higher priority.
Time Quantum: The time quantum for the associated priority. This illustrates the inverse relationship between priorities and time quanta, where higher priorities are allocated shorter time quanta, and lower priorities receive longer time quanta.

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figure 7.4 Solaris dispatch table for interactive and time-sharing threads.

The Lowest Priority (priority 0)has the highest time quantum (200 milliseconds), while the **highest priority (priority 59)** has the lowest time quantum (20 milliseconds).
Time Quantum Expired: Threads that use their entire time quantum without blocking (CPU-intensive threads) have their priorities lowered.
Return from Sleep: When a thread returns from sleep (e.g., after waiting for I/O), its priority is boosted to a range between 50 and 59 to support good response times for interactive processes.

Solaris 9 introduced two new scheduling classes: fixed priority and fair share. Fixed priority threads have static priorities, while fair-share scheduling uses CPU shares for resource allocation. Solaris also uses the system class for kernel processes, with priority fixed once

established. Real-time threads have the highest priority, guaranteeing a bounded response time. The scheduler assigns global priorities by converting class-specific priorities, selecting the highest for execution. If priorities match, round-robin scheduling is used. Solaris 9 shifted from the many-to-many model to the one-to-one model for thread management.

Example: Windows XP Scheduling

Windows XP uses a priority-based, preemptive scheduling algorithm, managed by the dispatcher. Threads are scheduled based on a 32-level priority scheme, divided into two classes: variable (priorities 1-15) and real-time (priorities 16-31). A thread runs until preempted by a higher-priority thread, its time quantum expires, it terminates, or it calls a blocking system call. Real-time threads have preferential access to the CPU. The dispatcher uses queues for each priority level and executes the idle thread if no ready threads are found. Windows XP also links kernel priorities with the Win32 API, which includes several

REALTIME-PRIORITY_CLASS
- HIGH-PRIORITY-CLASS
- ABOVE_NORMAL.PRIORITY_CLASS
- NORMAL-PRIORITY-CLASS
- BELOW.NORMAL_PRIORITY-CLASS
- IDLE-PRIORITY-CLASS

In all classes except the REALTIME-PRIORITY-CLASS, thread priorities are variable, allowing their priority to be adjusted dynamically. This enables the system to respond to changing workload conditions and scheduling needs.

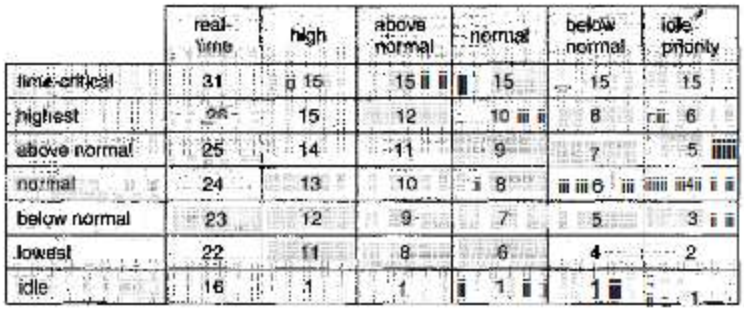


Figure 7.5 Windows XP priorities

Within each priority class, there is a relative priority, with values that include different levels of priority within the class.

- TIMEJZRITICAL
- HIGHEST
- ABOVE-NORMAL
- NORMAL
- BELOW-NORMAL
- LOWEST
- IDLE

The priority of a thread is determined by both the priority class it belongs to and its relative priority within that class. Each class has a range of priorities, and the thread's numeric priority is based on its relative priority within the class. The default base priority for a thread is set to the normal relative priority of its class, which varies for each priority class.

REALTIME-PRIORTTY-CLASS—24
• HIGH_PRIORITY-CLASS—13
• ABOVE-NORM AL.PRIORJTY-CLASS—10
• NORMAL-PRIORITY.CLASS—8
• BELOW.NORMAL_PRIORITY_CLASS—6
• IDLE-PRIORITY-CLASS—4

Processes usually belong to the NORMAL_PRIORITY_CLASS unless specified otherwise during creation or if the parent process was in the IDLE_PRIORITY_CLASS. A thread's initial priority typically matches the base priority of its process. When a thread's time quantum expires, its priority may be lowered (but not below the base priority), limiting CPU usage for compute-bound threads. Upon being released from waiting, the thread's priority may be boosted, depending on the type of wait operation, to improve response times for interactive or I/O-bound tasks. In Windows XP, the foreground process receives a priority boost, increasing its time quantum for better performance

Example: Linux Scheduling

Prior to version 2.5, the Linux kernel used a traditional UNIX scheduling algorithm, which struggled with scalability and SMP support. Version 2.5 introduced a new O(1) scheduler, running in constant time regardless of tasks and enhancing SMP support with processor affinity and load balancing. Linux uses a preemptive, priority-based algorithm with two ranges: real-time (0–99) and nice values (100–140). Higher-priority tasks get longer time quanta, unlike Solaris and Windows XP.

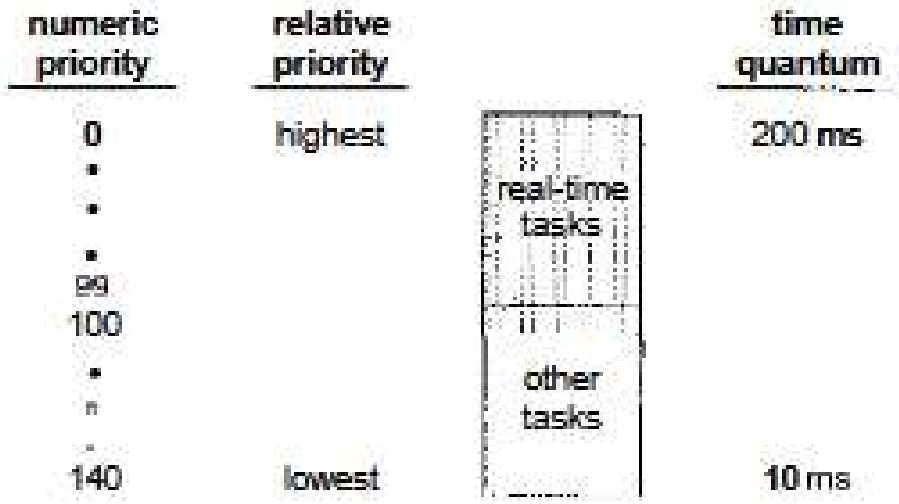
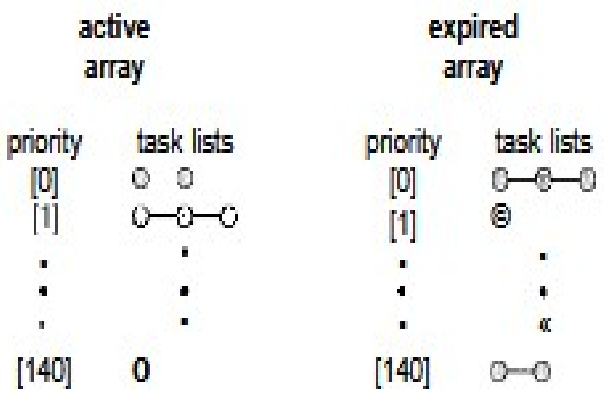


Figure 7.6 The relationship between priorities and time-slice length



In Linux, runnable tasks are eligible for CPU execution until their time slice expires. When a task's time slice is exhausted, it moves to an "expired" priority array and waits until all other tasks exhaust their slices. Each CPU maintains its own runqueue with two arrays: active (tasks with remaining time) and expired. Once the active array is empty, the arrays are swapped. Real-time tasks have static priorities, while other tasks have dynamic priorities adjusted based on interactivity, favoring tasks with longer I/O waits. Priority recalculations occur when tasks move to the expired array.

Algorithm Evaluation:

Selecting a CPU scheduling algorithm requires defining criteria like CPU utilization, response time, or throughput. Examples include maximizing CPU utilization with a 1-second response time limit or ensuring throughput aligns with proportional turnaround times. Once criteria are set, the algorithms are evaluated using various methods.

7.4.1 Deterministic Modeling

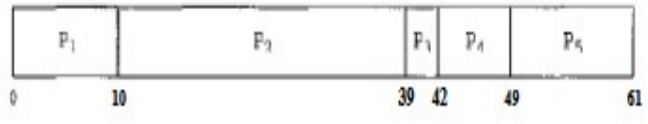
Analytic evaluation assesses an algorithm's performance using the system workload, often through deterministic modeling. This approach analyzes a predefined workload and calculates performance metrics for each algorithm under consideration. For instance, processes with specific arrival times and CPU bursts can be evaluated to determine efficiency.

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_i	7
P_5	12

Consider the FCFS, SJF, and RR (quantum = 10 ms) scheduling algorithms for this set of processes. To determine which algorithm provides the minimum average waiting time:

- **FCFS (First-Come, First-Served):** Processes are executed in the order of arrival.

Further analysis of execution order and average waiting times would clarify which algorithm performs best.



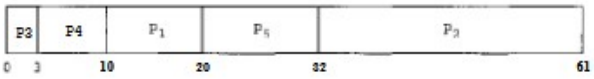
For the given processes using the **FCFS algorithm**, the waiting times are:

- **P1**: 0 ms
- **P2**: 10 ms
- **P3**: 39 ms
- **P4**: 42 ms
- **P5**: 49 ms

The average waiting time is calculated as:

Average waiting time=(0+10+39+42+49)5=28 ms

Using **non-preemptive SJF scheduling**, the processes would be executed differently.



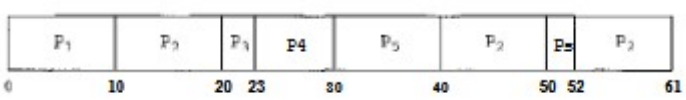
Using **non-preemptive SJF scheduling**, the waiting times for the processes are:

- **P1**: 10 ms
- **P2**: 32 ms
- **P3**: 0 ms
- **P4**: 3 ms
- **P5**: 20 ms

The average waiting time is calculated as:

Average waiting time=(10+32+0+3+20)5=13 ms.

Using the **RR algorithm (quantum = 10 ms)**, the execution order and waiting times would differ.



The waiting times using the **RR algorithm (quantum = 10 ms)** are:

- **P1**: 0 ms
- **P2**: 32 ms
- **P3**: 20 ms
- **P4**: 23 ms
- **P5**: 40 ms

The average waiting time is calculated as:

Average waiting time=(0+32+20+23+40)5=23 ms

In this example:

- The **SJF policy** results in the lowest average waiting time.
- The **FCFS scheduling** gives the highest average waiting time.
- The **RR algorithm** provides an intermediate average waiting time.

Deterministic modeling is useful for comparing algorithms because it produces exact results for given inputs. However, it requires precise inputs and is only applicable to specific scenarios. It is most valuable for explaining algorithms, analyzing trends, and optimizing repeatable workloads. For instance, in cases like this, SJF always minimizes waiting time when all process details are available upfront.

7.4.2 Queueing Models

When processes vary daily, deterministic modeling becomes less applicable. Instead, CPU and I/O burst distributions can be measured or estimated, typically using an exponential distribution defined by its mean. Similarly, arrival-time distributions describe when processes enter the system. Using these distributions, we can calculate metrics like average throughput, utilization, and waiting time for scheduling algorithms.

In queueing-network analysis, the system is modeled as servers with queues (e.g., the CPU's ready queue and I/O device queues). Metrics like average queue length, waiting time, and utilization can be derived from arrival and service rates. For steady-state systems, the number of processes entering and leaving a queue must be equal.

$$n = \lambda \times W.$$

Little's formula, valid for any scheduling algorithm, relates the average arrival rate (X), average queue length (n), and average waiting time (W) in the form n=X×W. For instance, if 7 processes arrive per second and there are 14 in the queue, the average waiting time is 2 seconds.

Queueing analysis helps compare scheduling algorithms but has limitations due to simplified assumptions and unrealistic distributions, making results approximate and sometimes less accurate.

7.4.3 Simulations

Simulations offer a more accurate evaluation of scheduling algorithms by modeling the system and tracking its state over time. These models use software data structures and a clock variable to simulate system activities, gathering performance statistics during execution. The simulation can be driven by randomly generated data based on probability distributions or real-world system measurements.

However, using trace tapes (which record actual system events) can enhance accuracy by capturing event order, and these simulations provide a reliable method to compare algorithms under real-world conditions. Despite their precision, simulations are computationally expensive, time-consuming, and may require significant storage, along with considerable effort to design and debug.

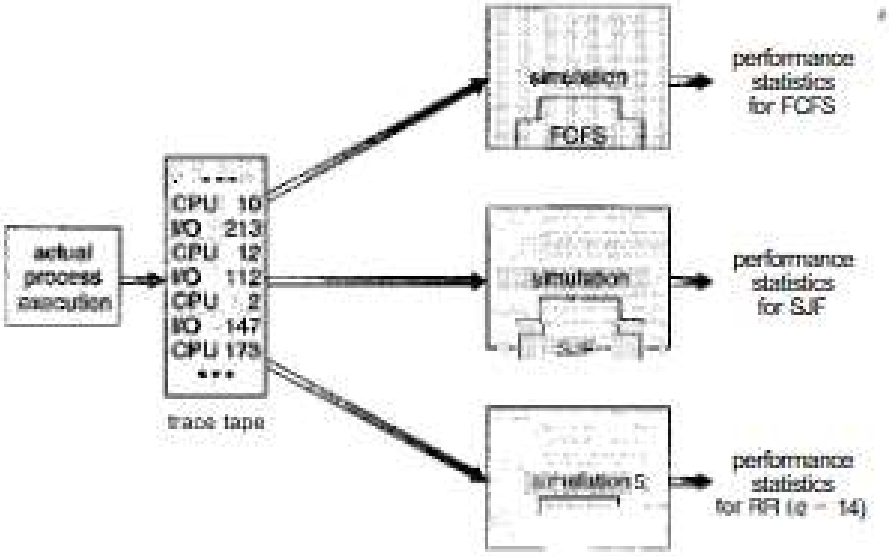


Figure 7.7 Evaluation of CPU schedulers by simulation

7.4.4 Implementation

The most accurate way to evaluate a scheduling algorithm is to implement it in the operating system and observe its performance under real conditions. However, this approach is costly, requiring extensive coding and modification of the OS, which may disrupt users. Additionally, the environment can change dynamically in response to the algorithm's behavior, as users adapt their processes to benefit from specific scheduling priorities.Some systems allow flexibility by enabling users or administrators to fine-tune scheduling algorithms for particular applications, such as graphic-intensive workstations or servers. However, such performance tuning often doesn't improve general system performance and may only benefit specific use cases.

7.5 SUMMARY

Multiple-processor scheduling involves strategies like processor affinity and load balancing to efficiently distribute tasks across CPUs. Symmetric multithreading improves performance by enabling parallel execution on shared resources.

Thread scheduling manages execution priorities, while contention scope determines whether threads contend within one processor or across multiple. Pthread scheduling handles multi-threaded processes in POSIX systems. OS examples like Solaris, Windows XP, and Linux have unique scheduling models tailored for their architectures. Evaluation methods like deterministic modeling, queueing models, and simulations help assess algorithm performance, though implementation in real systems offers the most accurate results.

7.6 KEY TERMS

User, kernel, process-contention scope,Thread Priority, Thread Execution, priority, time quantum

7.7 SELF-ASSESSMENT QUESTIONS

1. How do different approaches to multiple-processor scheduling improve system performance and efficiency?
2. What is processor affinity, and how does it affect the performance of a multi-core system?
3. How does load balancing in scheduling ensure optimal resource utilization across multiple processors?
4. What is the role of contention scope in thread scheduling, and how does it influence performance in a multi-threaded environment?
5. How do deterministic modeling, queueing models, and simulations differ in evaluating the performance of scheduling algorithms?

7.8 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”, Seventh Edition, Wiley.
2.William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3.Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4.Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5. Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR
Dr. Vasantha Rudramalla

LESSON 8
SYNCHRONIZATION

Aim and Objectives:

- To introduce the critical-section problem, whose solutions are essential for ensuring the consistency of shared data in a concurrent environment.
- To present both software and hardware solutions to the critical-section problem, highlighting their advantages and limitations.
- To introduce the concept of atomic transactions and describe mechanisms designed to ensure atomicity, preventing inconsistency in data when multiple processes access shared resources.

Structure:

8.1 Introduction

Synchronization in operating systems is the coordination of concurrent processes or threads to ensure that shared resources are accessed and modified in a consistent and orderly manner. As modern systems often involve multitasking and multithreading, processes frequently interact and share data or resources, making synchronization a critical aspect of system reliability and performance.

Importance of Synchronization

In a multitasking environment, processes or threads may execute simultaneously, leading to potential conflicts when accessing shared resources. Synchronization ensures:

1. **Data Consistency**:

   Prevents data corruption and inconsistencies by regulating access to shared resources.

2. **Mutual Exclusion**:

   Ensures that only one process can access a critical section of code at a time.

3. **Coordination**:

   Allows processes to work together efficiently, avoiding deadlocks and race conditions.

4. **Fairness**:

   Prevents indefinite blocking or starvation of any process while accessing shared resources.

Challenges in Synchronization

1. **Race Conditions**:
   Occur when the outcome of a computation depends on the timing or sequence of execution of processes.
2. **Deadlocks**:
   Arise when two or more processes are stuck, each waiting for the other to release a resource.
3. **Starvation**:
   Happens when a process waits indefinitely due to lack of resource access, often caused by priority issues.
4. **Priority Inversion**:
   Occurs when a higher-priority process is waiting for a lower-priority process holding a resource.

Synchronization Mechanisms

Operating systems use several mechanisms to manage synchronization effectively:

1. **Locks and Mutexes**:
   Used to ensure mutual exclusion by allowing only one thread or process to access a critical section at a time.
2. **Semaphores**:
   Counting variables that help manage access to limited resources by multiple processes.
3. **Monitors**:
   High-level constructs combining mutual exclusion and condition variables for process coordination.

4. **Message Passing**:
   Enables processes to synchronize by communicating through messages rather than shared memory.

Real-World Relevance

Synchronization is vital for applications like databases, operating systems, and multi-user environments, where concurrent processes must maintain data integrity while working together. Understanding synchronization ensures efficient system design and helps resolve challenges in concurrent programming.

8.2 The Critical-Section Problem

The **Critical-Section Problem** is a fundamental concept in operating systems and concurrent programming. It arises when multiple processes or threads need to access shared resources or data simultaneously, requiring mechanisms to ensure consistency and prevent conflicts. Synchronization techniques are used to solve this problem by controlling access to shared resources.

Components of the Critical-Section Problem

1. **Critical Section**
   The portion of a program where shared resources (like variables, files, or memory) are accessed. Only one process or thread should execute its critical section at any given time.
2. **Entry Section**
   The part of the program where a process or thread requests permission to enter the critical section.
3. **Exit Section**
   The code executed after leaving the critical section to release the resource.
4. **Remainder Section**
   The remaining part of the program that does not interact with the shared resource.

Requirements for a Solution
To solve the critical-section problem, any synchronization solution must satisfy these conditions:

Mutual Exclusion:
Only one process can execute in the critical section at a time.

1. **Progress:**
   If no process is in the critical section, other processes waiting to enter must be allowed to do so without indefinite delay.
2. **Bounded Waiting:**
   A process attempting to enter the critical section should not wait indefinitely. There must be a bound on how many other processes can access the critical section before it is granted access.

Example: Peterson's Algorithm for Two Processes

Peterson's Algorithm is a classic software-based synchronization solution for two processes.

Peterson's Algorithm Code

```
#include <stdio.h>
#include <stdbool.h>
#include <pthread.h>
#include <unistd.h>

// Shared variables
bool flag[2] = {false, false}; // Indicates if a process wants to enter critical section
int turn = 0; // Indicates whose turn it is to enter

// Critical section simulation
void critical_section(int id) {
   printf("Process %d: Entering critical section.\n", id);
   sleep(1); // Simulate critical section work
   printf("Process %d: Leaving critical section.\n", id);
}

// Process 0 function
void* process0(void* arg) {
   while (1) {
      flag[0] = true; // Process 0 wants to enter
      turn = 1;       // Give turn to process 1
      while (flag[1] && turn == 1); // Wait if process 1 wants to enter

      // Critical section
      critical_section(0);

      // Exit section
      flag[0] = false; // Process 0 is done
      sleep(1);        // Simulate remainder section
   }
}

// Process 1 function
void* process1(void* arg) {
   while (1) {
      flag[1] = true; // Process 1 wants to enter
      turn = 0;       // Give turn to process 0
```

```
        while (flag[0] && turn == 0); // Wait if process 0 wants to enter

        // Critical section
        critical_section(1);

        // Exit section
        flag[1] = false; // Process 1 is done
        sleep(1);       // Simulate remainder section
    }
}

int main() {
    pthread_t t0, t1;

    // Create two threads for the processes
    pthread_create(&t0, NULL, process0, NULL);
    pthread_create(&t1, NULL, process1, NULL);

    // Join threads (infinite loop in this example)
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);

    return 0;
}
```

Explanation

1. **Flags and Turn Variable:**
   - flag[0] and flag[1] indicate if Process 0 and Process 1 want to enter the critical section.
   - Turn ensures fairness, giving one process priority over the other.
2. **Entry Section:**
   A process sets its flag to true and checks the other process's flag and the turn variable. It waits if the other process has priority.
3. **Critical Section:**
   Only one process enters at a time due to mutual exclusion.
4. **Exit Section:**
   The process resets its flag to false to indicate it no longer needs the critical section.

Example: Mutex for Critical Section

For modern applications, using **mutexes** is a practical approach to solve the critical-section problem.

Mutex Example Code

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

// Mutex variable
pthread_mutex_t mutex;

// Critical section simulation
void critical_section(int id) {
   printf("Thread %d: Entering critical section.\n", id);
   sleep(1); // Simulate critical section work
   printf("Thread %d: Leaving critical section.\n", id);
}

// Thread function
void* process(void* arg) {
   int id = *((int*)arg);

   while (1) {
      // Entry section: Lock the mutex
      pthread_mutex_lock(&mutex);

      // Critical section
      critical_section(id);

      // Exit section: Unlock the mutex
      pthread_mutex_unlock(&mutex);

      sleep(1); // Simulate remainder section
   }
}

int main() {
   pthread_t t0, t1;
   int id0 = 0, id1 = 1;

   // Initialize the mutex
   pthread_mutex_init(&mutex, NULL);

   // Create threads
   pthread_create(&t0, NULL, process, &id0);
```

```
    pthread_create(&t1, NULL, process, &id1);

    // Join threads (infinite loop in this example)
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);

    // Destroy the mutex
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Explanation

1. **Mutex Initialization:**
   A mutex is initialized using pthread_mutex_init().
2. **Lock and Unlock:**
   - pthread_mutex_lock() ensures mutual exclusion by locking the critical section.
   - pthread_mutex_unlock() releases the lock, allowing other threads to enter.
3. **Critical Section Execution:**
   Only one thread can enter the critical section at a time.

The **Critical-Section Problem** is central to synchronization in operating systems. Solutions like Peterson's Algorithm and mutexes ensure mutual exclusion, progress, and bounded waiting. While software-based solutions like Peterson's Algorithm are conceptually important, mutexes and other hardware-supported mechanisms are preferred in real-world applications for efficiency and scalability.

8.3 Peterson's Solution

Peterson's Solution is a classical software-based protocol for achieving mutual exclusion in critical sections of a program. It ensures that no two processes are simultaneously executing their critical sections, addressing synchronization challenges in multi-threaded or multi-process environments.
Peterson's solution is particularly significant because it demonstrates mutual exclusion, progress, and bounded waiting using simple variables without relying on hardware-based atomic operations like test-and-set or semaphores.

Features of Peterson's Solution

1. **Two Processes**
   The original solution applies to two processes, typically denoted as P0 and P1. It can be extended to more processes with additional complexity.
2. **Shared Variables**

- **flag[]**: An array where each element indicates if a process is ready to enter the critical section.
  - flag[0] = true indicates P0 wants to enter.
  - flag[1] = true indicates P1 wants to enter.
- **turn**: Indicates whose turn it is to enter the critical section if both processes want to enter at the same time.

3. **Mutual Exclusion**
   The solution ensures that only one process can access the critical section at a time.
4. **Progress**
   A process that does not want to enter the critical section does not prevent other processes from entering.
5. **Bounded Waiting**
   A process attempting to enter the critical section will eventually be allowed to enter.

Algorithm

Entry Section

Each process:

1. Sets its flag to true to indicate its intention to enter the critical section.
2. Sets the turn variable to the other process.
3. Waits while the other process has its flag set to true and it's the other process's turn.

Critical Section

The process executes its critical section code.

Exit Section

After finishing the critical section, the process sets its flag to false, indicating it no longer needs access.

Example: Peterson's Solution for Two Processes

Below is a simple implementation of Peterson's solution for two threads in C:

```
#include <stdio.h>
#include <pthread.h>
#include <stdbool.h>
#include <unistd.h>

// Shared variables
```

```
bool flag[2] = {false, false};
int turn = 0;

// Critical section simulation
void critical_section(int id) {
   printf("Thread %d is in the critical section.\n", id);
   sleep(1); // Simulate work
   printf("Thread %d is leaving the critical section.\n", id);
}

// Thread function for process 0
void* process0(void* arg) {
   while (1) {
      flag[0] = true; // Indicate process 0 wants to enter
      turn = 1;       // Allow process 1 to go first if there's a conflict
      while (flag[1] && turn == 1); // Wait while process 1 is in its critical section

      // Critical section
      critical_section(0);

      // Exit section
      flag[0] = false; // Indicate process 0 is done
      sleep(1); // Simulate work outside the critical section
   }
}

// Thread function for process 1
void* process1(void* arg) {
   while (1) {
      flag[1] = true; // Indicate process 1 wants to enter
      turn = 0;       // Allow process 0 to go first if there's a conflict
      while (flag[0] && turn == 0); // Wait while process 0 is in its critical section

      // Critical section
      critical_section(1);

      // Exit section
      flag[1] = false; // Indicate process 1 is done
      sleep(1); // Simulate work outside the critical section
   }
}

int main() {
```

```
    pthread_t t0, t1;

    // Create two threads
    pthread_create(&t0, NULL, process0, NULL);
    pthread_create(&t1, NULL, process1, NULL);

    // Wait for threads to finish (they won't in this infinite loop example)
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);

    return 0;
}
```

Explanation of the Example

1. **Initialization**

   The flag array is initialized to false, and turn is set arbitrarily.

2. **Thread Behavior**
   - Each thread represents a process (P0 and P1) attempting to access a shared critical section.
   - A thread sets its flag to true and gives the other thread the "turn" to prevent deadlock.
   - The while loop in the entry section ensures that the thread waits until the other thread is not in its critical section or it is not its turn.
3. **Critical Section**: Only one thread can execute the critical section at a time due to the mutual exclusion logic.
4. **Exit Section:** After finishing, the thread sets its flag to false to indicate it no longer needs the critical section.

Advantages of Peterson's Solution

- **Simplicity:** Requires only shared variables and basic constructs like loops and conditions.
- **No Special Hardware:** Unlike hardware locks, it works in pure software.
- **Compliance:** Ensures mutual exclusion, progress, and bounded waiting.

Limitations

1. **Restricted to Two Processes:** Extending to more processes is possible but significantly increases complexity.
2. **Busy Waiting:** Processes continually check the flag and turn variables, consuming CPU time.

3. **Not Suitable for Modern Architectures:** Assumes sequential consistency, which may not hold on modern multi-core CPUs due to optimizations like instruction reordering.

Peterson's solution is an elegant and educational algorithm for achieving mutual exclusion in a simple environment. Although it is rarely used in practice due to its limitations, it remains a foundational concept in understanding synchronization mechanisms in operating systems.

8.4 SYNCHRONIZATION HARDWARE

Synchronization hardware in operating systems refers to specialized instructions or mechanisms provided by modern computer architectures to ensure synchronization among processes or threads. These mechanisms help resolve issues in the critical-section problem by supporting **atomic operations**—operations that cannot be interrupted once started.

Key Features

1. **Atomicity:** Synchronization hardware ensures that certain operations are performed atomically (completely or not at all), preventing race conditions.
2. **Efficiency:** Hardware mechanisms are faster and more reliable compared to software-only solutions, especially in multiprocessor systems.

Synchronization hardware refers to special machine-level instructions used to control access to shared resources in multi-threaded or multi-processor systems. These instructions help prevent issues like race conditions and ensure proper synchronization between processes or threads.

Key Types of Synchronization Hardware

1. **Test-and-Set (TAS)**
   - **Description**: The test-and-set instruction is an atomic operation that tests a value and sets it to a specific value (usually true). It helps in locking mechanisms by setting a flag and checking if it is already set, allowing only one process to access the critical section.
   - **Example**:
     - test-and-set(flag):
       - If the flag is false, the instruction sets it to true and returns the previous value.
       - If the flag is already true, the process has to wait.
     - This is often used to implement spinlocks where processes repeatedly check the flag until they can set it.
2. **Compare-and-Swap (CAS)**

- **Description**: The compare-and-swap operation compares the value of a memory location with an expected value and, if they match, swaps it with a new value. This operation is atomic, meaning it executes as a single, indivisible operation.
- **Example**:
  - compare-and-swap(memory_location, expected_value, new_value)
  - If memory_location == expected_value, it is replaced with new_value. Otherwise, no change occurs.
  - CAS is used in lock-free data structures like queues and stacks, where threads can try to update a value without blocking other threads.

3. **Load-Link / Store-Conditional (LL/SC)**
   - **Description**: This pair of operations is used for atomic updates. Load-Link reads a value from memory, while Store-Conditional writes to the memory location only if no other process has modified it since the Load-Link operation.
   - **Example**:
     - **Load-Link**: Loads the value from the memory location.
     - **Store-Conditional**: Attempts to write a new value to the memory location if it has not been modified since the Load-Link.
     - Used in implementing software transactional memory or lock-free data structures.
4. **Memory Barriers (Fences)**
   - **Description**: Memory barriers ensure that certain memory operations are completed before others can start. They prevent the processor from reordering instructions around the barrier.
   - **Example**:
     - A write memory barrier will ensure that all writes (e.g., updating variables or flags) are completed before proceeding with further operations, avoiding out-of-order execution issues in concurrent systems.

Example Scenario: Implementing a Lock with Test-and-Set

A classic example of using hardware synchronization is implementing a spinlock using the test-and-set instruction:

```
boolean flag = false;

void lock() {
   while (test-and-set(&flag)) { // While flag is true, keep spinning
      // Wait (busy-wait loop)
   }
}
```

```
void unlock() {
   flag = false; // Release the lock
}
```

In this example:

- The test-and-set instruction atomically checks the value of flag and sets it to true if it's not already.
- If another process has already set flag to true, the process has to spin (busy-wait) until it becomes false (the lock is released).

Synchronization hardware provides efficient, low-level mechanisms for managing concurrency and ensuring mutual exclusion in multi-threaded or multi-processor environments. These hardware primitives help implement high-level synchronization techniques like locks and condition variables.

Advantages of Synchronization Hardware

1. **Performance:** Reduces overhead compared to complex software-based algorithms.
2. **Simplicity:** Simplifies the implementation of mutual exclusion.
3. **Scalability:** Particularly beneficial in multi-core or multiprocessor systems.

8.5 SUMMARY

Synchronization in operating systems is essential for managing the interaction of concurrent processes or threads that share resources. It ensures orderly execution, maintains data consistency, and prevents issues like race conditions, deadlocks, and starvation.

Key components of synchronization include **mutual exclusion**, which restricts simultaneous access to critical sections, and **process coordination**, which organizes the execution order among processes.

Mechanisms such as **locks**, **mutexes**, **semaphores**, and **monitors** are used to implement synchronization, alongside **message passing** for communication between processes.

Synchronization is crucial for system stability, ensuring fairness, preventing resource conflicts, and enabling efficient multitasking in modern operating systems.

8.6 KEY TERMS

Synchronization, Data Consistency, Mutual Exclusion, Deadlock, Priority Inversion, Critical Section, Entry Section, Exit Section, Progress,

8.7 SELF-ASSESSMENT QUESTIONS

1. What is the purpose of synchronization in operating systems?
2. Define a "critical section" in the context of concurrent processes.
3. What are the three requirements that any solution to the critical-section problem must satisfy?
4. Explain why mutual exclusion is essential in solving the critical-section problem.
5. What is Peterson's solution, and how does it ensure mutual exclusion?
6. Describe the role of the flag and turn variables in Peterson's solution.
7. What are the limitations of Peterson's solution in modern multiprocessor systems?
8. How does the test-and-set instruction work in synchronization?
9. What are the advantages of using hardware-based synchronization mechanisms?

8.8 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”, Seventh Edition, Wiley.
2.William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3.Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4.Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5. Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Dr. Vasantha Rudramalla

CHAPTER- 9
SEMAPHORES

Aim and Objectives:

- Semaphores' are synchronization tools used to manage access to shared resources in concurrent programming, ensuring that multiple processes or threads do not conflict.
- They are used to implement mutual exclusion, process synchronization, and signal management between processes or threads.
- The primary goal is to prevent issues such as race conditions, deadlocks, and starvation, by controlling the execution flow of processes in a multitasking environment.

Structure:

9.1 Semaphores

Semaphores are commonly used to solve synchronization issues without requiring busy waiting. Furthermore, operating systems provide synchronization mechanisms like semaphores, mutexes, and condition variables to manage shared data access. In the context of transactions, ensuring atomicity and serializability is vital, often achieved through write-ahead logs, undo/redo operations, and concurrency-control schemes

A **semaphore** is a synchronization primitive used to control access to shared resources in a concurrent system. It is a variable or abstract data type that is manipulated using two atomic operations: **wait (P)** and **signal (V)**.

1. **Wait (P)**: This operation decreases the value of the semaphore. If the value becomes negative, the process is blocked until the semaphore is positive again.
2. **Signal (V)**: This operation increases the value of the semaphore, potentially unblocking a waiting process.

Types of Semaphores:

- **Binary Semaphore (Mutex)**: Takes only values 0 or 1, used for mutual exclusion.
- **Counting Semaphore**: Used for controlling access to a resource pool, with a range of values beyond just 0 or 1.

Example:

Consider two processes trying to access a shared resource:

```
semaphore mutex = 1; // Initially, the semaphore is available (1).

void process() {
   wait(mutex); // Enter critical section (decrement semaphore)
   // Access shared resource
   signal(mutex); // Leave critical section (increment semaphore)
}
```

In this example, only one process can access the critical section at a time. If the semaphore is 0, other processes must wait until it becomes positive.

9.1.1 Usage

 Counting Semaphores: These semaphores can take any non-negative integer value and are used to manage a resource pool with a finite number of instances. Example: If a printer pool has 3 printers, the semaphore is initialized to 3.

 Binary Semaphores (Mutex Locks): These have values of either 0 or 1, used to enforce mutual exclusion, ensuring only one process enters the critical section at a time. Example: A process can enter a critical section only if mutex = 1. After entering, it sets mutex = 0 to block others.

 Critical Section Control with Binary Semaphores: Multiple processes share a binary semaphore (mutex) initialized to 1. Each process calls wait(mutex) before entering its critical section and signal(mutex) when leaving, ensuring mutual exclusion.

 Resource Management with Counting Semaphores: Counting semaphores control access to a limited number of resources. If the semaphore value is greater than 0, a process can decrement it (use a resource); if 0, processes block until a resource is freed. Example: A system with 5 printers uses a counting semaphore initialized to 5.

 Semaphore Example for Synchronization: Suppose process P1 executes S1 and process P2 executes S2, where S2 should only execute after S1. This can be achieved by using a shared semaphore synch, initialized to 0.

 Synchronization using Semaphores:

- Process P1 executes S1 and then signals synch with signal(synch).
- Process P2 waits on the semaphore with wait(synch) before executing S2. This ensures that P2 runs S2 only after P1 has completed S1.

 Blocking with Counting Semaphores: When a process tries to use a resource but the semaphore value is 0, it blocks until the resource becomes available. For example, a resource with 3 available units would block processes trying to access it once the semaphore reaches 0.

 Semaphore Operations:

- **Wait()**: Decrements the semaphore value. If the value is 0, the process is blocked.
- **Signal()**: Increments the semaphore value, potentially waking up a blocked process.

 Mutex for Mutual Exclusion: Binary semaphores (mutexes) are commonly used to protect critical sections in multi-threaded environments, ensuring that only one thread can access shared resources at a time.

 Semaphore Use in Producer-Consumer Problem: In a producer-consumer system, a counting semaphore can be used to track the number of available slots in a buffer, ensuring that producers don't overwrite the buffer and consumers don't attempt to consume from an empty buffer.

9.3.2 Implementation

The main disadvantage of semaphores is the need for busy waiting, where a process continuously checks if it can enter the critical section. This wastes CPU cycles that could be used by other processes. In systems with a single CPU, busy waiting can be inefficient. Spinlocks, a type of semaphore, avoid context switches by "spinning" while waiting for a lock. However, they are only useful for short lock durations. To overcome busy waiting, we can modify the wait() and signal() operations to block processes when the semaphore value is negative, adding them to a waiting list. A process is awakened when a signal() operation occurs. This blocking method, which avoids busy waiting, helps in better managing system resources and CPU time. A semaphore can be defined as a structure containing a value and a process list. While this reduces CPU waste, care must be taken to ensure atomic execution of wait() and signal() operations to avoid race conditions..

9.4.3 Deadlocks and Starvation

A **deadlock** occurs when two or more processes are blocked indefinitely, each waiting for a resource held by another. In this situation, no process can proceed, resulting in a "circular wait."

Example of Deadlock:

Consider two processes, **P1** and **P2**, each needing two semaphores to complete their tasks. If **P1** acquires **semaphore1** and waits for **semaphore2**, and **P2** acquires **semaphore2** and waits for **semaphore1**, a deadlock occurs because both processes are stuck, each waiting for the semaphore held by the other.

```
sem_t semaphore1, semaphore2;

void* process1(void* arg) {
   sem_wait(&semaphore1); // Acquires semaphore1
   sem_wait(&semaphore2); // Waits for semaphore2 (Deadlock)
   // Critical section
}
void* process2(void* arg) {
   sem_wait(&semaphore2); // Acquires semaphore2
   sem_wait(&semaphore1); // Waits for semaphore1 (Deadlock)
   // Critical section
}
```

Starvation

Starvation occurs when a process is continually denied access to a resource because other processes keep taking the resources first. This is often caused by unfair scheduling or prioritization.

Example of Starvation:

In a system where a high-priority process continuously acquires the semaphore before a low-priority process, the low-priority process may never get a chance to execute.

```
sem_t semaphore;

void* high_priority_process(void* arg) {
   sem_wait(&semaphore); // Acquires semaphore
   // Executes
   sem_post(&semaphore);
}

void* low_priority_process(void* arg) {
   sem_wait(&semaphore); // Waits for semaphore (Starvation risk)
   // Executes
   sem_post(&semaphore);
}
```

Here, the **high-priority process** might always acquire the semaphore first, leading to starvation for the **low-priority process**.

- **Deadlock**: A situation where processes wait indefinitely for resources, creating a circular dependency.
- **Starvation**: A situation where a process is perpetually denied resources due to unfair scheduling or resource allocation.

In both cases, the correct use of semaphores and careful scheduling policies can help prevent these issues. For example, avoiding circular waits can prevent deadlocks, while fair scheduling algorithms can mitigate starvation.

9.2 CLASSIC PROBLEMS OF SYNCHRONIZATION

This section discusses synchronization problems that serve as examples for testing concurrency-control schemes. We use semaphores to solve these problems. For instance, in the producer-consumer problem, the producer process produces an item, waits for an empty slot, and then adds the item to the buffer. The process involves using semaphores to control access to the shared buffer and ensure synchronization between the producer and consumer, as shown in the provided structure.

9.2.1 The Bounded-Buffer Problem

The bounded-buffer problem involves a shared buffer pool with limited capacity. Semaphores are used to manage synchronization between the producer and consumer processes. The mutex semaphore ensures mutual exclusion, while empty and full semaphores track available and used buffers. The producer adds items to the buffer, and the consumer removes them, with both processes ensuring proper synchronization to prevent race conditions. For example, the producer waits for an empty slot (empty), then accesses the buffer (mutex), and signals the consumer once an item is produced.

9.2.2 The Readers-Writers Problem

The readers-writers problem involves synchronizing processes that either read from or write to a shared database. Readers can access the database simultaneously, but writers need exclusive access. The problem has variations, including the first and second readers-writers problems. In the first, readers wait only if a writer is active, while in the second, writers should get immediate access when ready. To solve this, semaphores like mutex and wrt manage access to shared resources. The readcount variable tracks active readers, and wrt ensures mutual exclusion for writers. This problem is commonly used to test synchronization techniques, especially in systems with more readers than writers.

9.2.3 The Dining-Philosophers Problem

The dining philosophers problem involves five philosophers sitting around a circular table with chopsticks. Each philosopher alternates between thinking and eating, requiring two chopsticks to eat. The challenge arises when philosophers attempt to pick up chopsticks simultaneously, potentially causing deadlock or starvation.

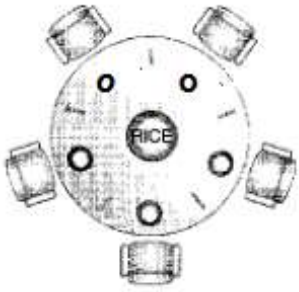


Figure 9.1The situation of the dining philosophers.

A basic solution uses semaphores to control access to the chopsticks, but a deadlock can occur if each philosopher picks up one chopstick and waits for the other. To solve this, we can limit the number of philosophers at the table, ensure a philosopher picks both chopsticks in a critical section, or use an asymmetric solution where odd philosophers pick up chopsticks in a different order than even philosophers. These solutions aim to prevent deadlock and starvation.

9.3 MONITORS

Semaphores are an effective synchronization tool but can lead to timing errors when misused. These errors may occur only under certain execution sequences, making them difficult to detect. For example, in the producer-consumer problem, a timing issue arose from incorrect use of counters, where the counter appeared reasonable but was still faulty. In critical-section problems, if semaphores like mutex are used incorrectly, such as by reversing the wait and signal order, multiple processes could enter their critical sections simultaneously, violating mutual exclusion. Other errors, like omitting wait or signal, can lead to deadlock or mutual exclusion violations. These issues highlight the challenges in using semaphores correctly, prompting the development of high-level synchronization constructs like monitors to address such errors.

9.3.1 Usage

A monitor is an abstract data type that encapsulates private data and provides public methods to operate on that data with mutual exclusion. It contains shared variable declarations and procedure definitions that manipulate these variables. Processes cannot directly access the monitor's data, ensuring encapsulation and synchronization. The monitor construct guarantees that only one process can be active at a time, eliminating the need for explicit synchronization code. For more complex synchronization, condition variables are introduced, allowing processes to wait and signal other processes. A signal operation only resumes one suspended process, unlike semaphores, where signal() always affects the semaphore's state. When a signal is invoked, the signaling process may either wait or continue, depending on the design choice. The language Concurrent Pascal uses a compromise where the signaling process leaves the monitor immediately, allowing the suspended process to resume.

9.3.3 Implementing a Monitor Using Semaphores

The Dining Philosophers Problem involves five philosophers sitting at a circular table, each needing two chopsticks to eat. They alternate between thinking and eating. Using monitors ensures safe synchronization by encapsulating shared resources and their operations.

A monitor provides mutual exclusion within its procedures, ensuring that only one philosopher can access the critical section (eating) at a time.

```
monitor DiningPhilosophers {
   condition chopstick[5];

   procedure pickup(i) {
      wait(chopstick[i]);
      wait(chopstick[(i+1)%5]);
   }
```

```
    procedure putdown(i) {
        signal(chopstick[i]);
        signal(chopstick[(i+1)%5]);
    }
}
```

Philosopher Process:

```
do {
    think();
    pickup(i);   // Pick up chopsticks
    eat();
    putdown(i);  // Put down chopsticks
} while(true);
```

Example Workflow:

1. Philosopher 0 picks chopsticks 0 and 1, then eats.
2. Philosopher 1 waits for chopstick 1, eats when available.
3. After eating, each philosopher puts down chopsticks, signaling others.

Philosopher 0 → (eating) ← Philosopher 1

↑ ↓

Philosopher 4 ← (thinking) → Philosopher 2

↑ ↓

Philosopher 3 ← (waiting) → Philosopher 4

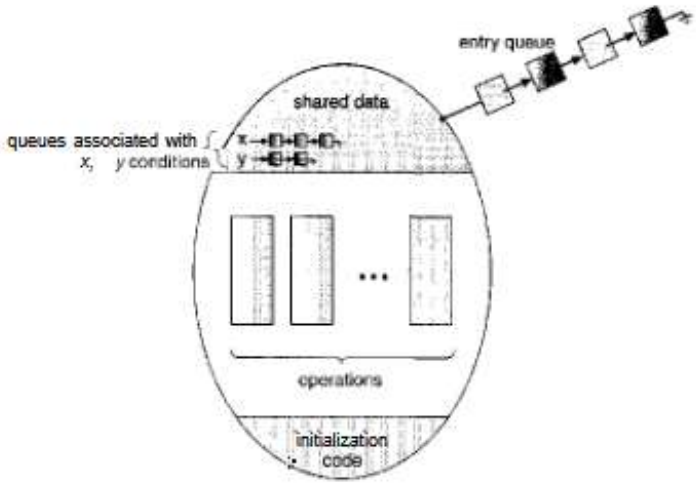


Figure 9.2 Monitor with condition variable

Benefits:

- **Deadlock-free**: Only one philosopher can access the chopsticks at a time.
- **Starvation-free**: Every philosopher will eventually get a chance to eat.

9.3.3 Implementing a Monitor Using Semaphores

A **monitor** in operating systems provides synchronization by encapsulating shared data and operations that manipulate this data. It ensures mutual exclusion, meaning only one process can execute within the monitor at a time. To implement a monitor using semaphores, we use semaphores for mutual exclusion and condition variables for synchronization.

- **Semaphore**: Used for mutual exclusion within the monitor.
- **Condition Variables**: Used to block and wake up processes based on certain conditions.

Example: Bounded-Buffer Problem (Producer-Consumer)

The goal is to synchronize a producer and a consumer accessing a bounded buffer. The producer inserts items into the buffer, while the consumer removes them. To implement a monitor using semaphores, we need semaphores for mutual exclusion and for counting full/empty slots in the buffer.

Monitor with Semaphores:

```
semaphore mutex = 1;     // Semaphore for mutual exclusion
semaphore empty = n;     // Semaphore for empty buffer slots
semaphore full = 0;      // Semaphore for full buffer slots

int buffer[n];           // Buffer of size n
int in = 0;              // Index for inserting an item
int out = 0;             // Index for removing an item

monitor BoundedBuffer {
   procedure producer() {
      while (true) {
         item = produceItem();      // Produce an item
         wait(empty);               // Wait if buffer is full
         wait(mutex);               // Enter critical section
         buffer[in] = item;         // Insert item into buffer
         in = (in + 1) % n;         // Update index
```

```
        signal(mutex);          // Exit critical section
        signal(full);           // Notify consumer
```

9.4.4 Resuming Processes Within a Monitor

In a monitor, processes can be blocked on a condition variable using the wait() operation, and they can be resumed using the signal() operation. The wait() operation suspends the process until a specific condition is met, while signal() wakes up one of the processes waiting on the condition.

Example: Producer-Consumer Monitor

```
monitor Buffer {
   condition full, empty;
   int buffer[10];
   int in = 0, out = 0;

   procedure producer() {
      while (true) {
         wait(empty);        // Wait if buffer is full
         buffer[in] = item;   // Produce item
         in = (in + 1) % 10;  // Update index
         signal(full);       // Notify consumer
      }
   }
      procedure consumer() {
      while (true) {
         wait(full);         // Wait if buffer is empty
         item = buffer[out];  // Consume item
         out = (out + 1) % 10; // Update index
         signal(empty);      // Notify producer
      }
   }
}
```

In this example, the consumer waits for the full condition, and the producer waits for the empty condition. When the producer or consumer performs a signal(), it resumes the blocked process.

9.4 SYNCHRONIZATION EXAMPLES

This section discusses synchronization mechanisms in Solaris, Windows XP, Linux, and the Pthreads API, highlighting the diverse approaches each system uses for kernel synchronization. These systems offer unique methods, with Pthreads being

particularly common for thread management in UNIX and Linux environments. The variations in their synchronization techniques are both subtle and significant.

9.4.1 Synchronization in Solaris

In Solaris, synchronization is achieved using several mechanisms, including mutexes, condition variables, semaphores, and reader-writer locks. These mechanisms ensure that multiple threads or processes do not interfere with shared resources, preventing race conditions and ensuring consistency.

Example:

One common synchronization primitive in Solaris is the **mutex**. A mutex is used to enforce mutual exclusion, ensuring that only one thread can access a critical section at a time.

```
mutex_t mutex;  // Declare a mutex

// Initialize the mutex

mutex_init(&mutex, USYNC_THREAD, NULL);

// Acquire the mutex before entering the critical section

mutex_lock(&mutex);

// Critical section code

printf("Thread is inside the critical section\n");

// Release the mutex after exiting the critical section

mutex_unlock(&mutex);
```

In this example, a thread locks the mutex before entering the critical section. Once done, it unlocks the mutex, allowing other threads to enter the critical section.

Solaris also supports condition variables for more complex synchronization needs, where threads may need to wait for a specific condition to be met before proceeding. Using semaphores and reader-writer locks, Solaris provides flexibility for process synchronization in multithreaded environments.

9.4.2 Synchronization in Windows XP

In Windows XP, synchronization is achieved using various primitives like mutexes, semaphores, critical sections, and events. These tools are crucial in managing access to shared resources among multiple threads, ensuring mutual exclusion, and avoiding race conditions.

Example: Mutex in Windows XP

A mutex in Windows XP is used to ensure that only one thread can access a critical section at a time.

```
#include <windows.h>

#include <stdio.h>

HANDLE hMutex;  // Declare a mutex

int main() {

   // Create a mutex

   hMutex = CreateMutex(NULL, FALSE, NULL);

   // Wait for the mutex (acquire it)

   WaitForSingleObject(hMutex, INFINITE);

   // Critical section: only one thread can execute this at a time

   printf("Thread is inside the critical section\n");

   // Release the mutex

   ReleaseMutex(hMutex);

   // Close the mutex handle

   CloseHandle(hMutex);

   return 0;

}
```

Explanation:

1. **CreateMutex**: Creates a mutex that will be used for synchronization.
2. **WaitForSingleObject**: The thread waits until it acquires the mutex before entering the critical section.
3. **ReleaseMutex**: After finishing the critical section, the mutex is released so other threads can acquire it.

Windows XP also supports **critical sections** for synchronizing threads within the same process, **semaphores** for controlling the count of resource access, and **events** for signaling between threads or processes. These synchronization mechanisms help ensure efficient and safe multithreading operations.

9.4.3 Synchronization in Linux

In Linux, synchronization between processes and threads is achieved through several mechanisms, including mutexes, semaphores, read-write locks, and condition variables. The Pthreads library (POSIX threads) provides the standard set of synchronization primitives for thread management in Linux.

Example: Mutex in Linux (Pthreads)

A mutex in Linux is used to ensure mutual exclusion, allowing only one thread to access a critical section at a time

```
#include <pthread.h>
#include <stdio.h>
pthread_mutex_t mutex;  // Declare a mutex
void* thread_func(void* arg) {
   pthread_mutex_lock(&mutex);  // Lock the mutex
   // Critical section
   printf("Thread is inside the critical section\n");
   pthread_mutex_unlock(&mutex);  // Unlock the mutex
   return NULL;
}
int main() {
   pthread_t thread1, thread2;
   // Initialize the mutex
   pthread_mutex_init(&mutex, NULL);
   // Create threads
   pthread_create(&thread1, NULL, thread_func, NULL);
   pthread_create(&thread2, NULL, thread_func, NULL);
   // Wait for threads to finish
   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);
   // Destroy the mutex
   pthread_mutex_destroy(&mutex);
   return 0;
```

}

1. **pthread_mutex_init**: Initializes a mutex object.
2. **pthread_mutex_lock**: The thread locks the mutex before entering the critical section to ensure that only one thread can access the shared resource at a time.
3. **pthread_mutex_unlock**: After finishing the critical section, the mutex is unlocked to allow other threads to acquire it.
4. **pthread_create**: Creates threads that will run the thread_func function.
5. **pthread_join**: Waits for the threads to complete their execution.

Other Synchronization Primitives in Linux:

- **Semaphores**: Used for signaling and controlling access to shared resources.
- **Condition Variables**: Used to synchronize threads by allowing them to wait for certain conditions to be met.
- **Read-Write Locks**: Allow multiple threads to read shared data simultaneously, but provide exclusive access for writers.

These synchronization mechanisms help manage concurrency and ensure thread safety in Linux systems.

9.4.4 Synchronization in Pthreads

In **Pthreads** (POSIX Threads), synchronization is crucial for managing concurrent threads safely. Pthreads provides several synchronization mechanisms, including **mutexes**, **condition variables**, **read-write locks**, and **semaphores**.

Key Synchronization Primitives in Pthreads:

1. **Mutexes**: Provide mutual exclusion, ensuring that only one thread can access the critical section at a time.
2. **Condition Variables**: Allow threads to wait for certain conditions to be true before proceeding, often used in conjunction with mutexes.
3. **Read-Write Locks**: Allow multiple threads to read shared data simultaneously but give exclusive access to a writer thread.

Example: Mutex in Pthreads

Here's an example of using a **mutex** to synchronize access to a critical section in a multithreaded environment.

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t mutex;  // Declare a mutex
```

```
void* thread_func(void* arg) {
    pthread_mutex_lock(&mutex);  // Lock the mutex

    // Critical section
    printf("Thread is inside the critical section\n");

    pthread_mutex_unlock(&mutex);  // Unlock the mutex
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // Initialize the mutex
    pthread_mutex_init(&mutex, NULL);

    // Create two threads
    pthread_create(&thread1, NULL, thread_func, NULL);
    pthread_create(&thread2, NULL, thread_func, NULL);

    // Wait for both threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Destroy the mutex
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

9.5 ATOMIC TRANSACTIONS

Mutual exclusion ensures that critical sections are executed atomically, meaning they execute as if in a single, sequential order. In many cases, we need critical sections to be atomic in a way that they either complete entirely or not at all, such as in funds transfer, where both debit and credit must occur together. Data consistency is essential in operations like these, often addressed in database systems. Operating systems, being data manipulators, can benefit from database techniques to improve flexibility and power in file management.

9.5.1 System Model

A transaction is a sequence of operations that performs a single logical function, often involving reading and writing data stored on disk. The main challenge in processing transactions is maintaining atomicity, which ensures that a transaction is either fully completed or not executed at all, despite potential system failures. A transaction ends with either a commit (successful completion) or an abort (due to errors or failures). If a transaction is aborted, any changes it made to data must be rolled back to its previous state, ensuring atomicity.

To achieve atomicity, systems must manage various types of storage:

- **Volatile storage** (e.g., memory), which is fast but loses data on crashes.
- **Nonvolatile storage** (e.g., disks), which is slower but survives crashes.
- **Stable storage**, which ensures data is never lost by replicating data across multiple nonvolatile devices.

The system must ensure atomicity by handling these storage types properly, especially in environments where failures may cause data loss in volatile storage.

9.5.2 Log-Based Recovery

One way to ensure atomicity in transactions is by using write-ahead logging, which records changes made by a transaction in a stable storage log. Each log entry includes the transaction name, data item name, old value, and new value. Key log records track the start, commit, or abort of transactions. Before any write operation, the corresponding log record must be written to stable storage. This ensures that updates are not made to data before they are logged, although it incurs a performance cost by requiring two writes for every update.

The recovery process, using the log, consists of two operations:

- **undo(T)**: Restores the data to its original state if a transaction T aborts.
- **redo(T)**: Reapplies updates from a committed transaction T if a system failure occurs.

Transactions are classified for recovery:

- A transaction must be undone if the log shows only a start record and no commit.
- A transaction must be redone if both start and commit records exist.

9.5.3 Checkpoints

When a system failure occurs, we need to examine the log to determine which transactions should be redone and which should be undone. However, this process has two key drawbacks: first, it can be time-consuming to search the entire log, and second, many transactions that need to be redone have already modified the data, causing unnecessary delays in recovery. To address these issues, **checkpoints** are introduced.

During execution, the system periodically creates checkpoints, which involve three main actions:

1. Writing all log records in volatile storage to stable storage.
2. Writing all modified data in volatile storage to stable storage.
3. Adding a <checkpoint> record to the log.

A <checkpoint> record helps streamline recovery. If a transaction (e.g., Tj) committed before the checkpoint, its modifications are already stored in stable storage, and no redo is necessary. The recovery procedure thus becomes more efficient.

After a failure, the recovery system identifies the most recent transaction (Tj) that started before the checkpoint. It then performs redo and undo operations only on transactions that started after Tj. Specifically:

- **Redo** is performed for transactions in which the <Tj, commits> record exists.
- **Undo** is performed for transactions without the <Tj, commits> record.

This reduces the amount of log that needs to be examined and speeds up recovery.

9.5.4 Concurrent Atomic Transactions

In a system with multiple transactions, serializability ensures that their concurrent execution is equivalent to some serial execution. One way to achieve this is by using a critical section, where transactions share a common semaphore mutex. Each transaction must acquire the mutex before execution and release it after committing or aborting. However, this approach is too restrictive, as it prevents overlapping execution. More flexible concurrency-control algorithms can be used to maintain serializability while allowing transactions to overlap.

9.5.6 Serializability

Consider a system with two data items, A and B, accessed by two transactions, T_0 and T_1. If these transactions execute atomically in a specific order (T_0 followed by T_1), it forms a serial schedule. A serial schedule ensures that the instructions of each transaction are executed sequentially. If transactions overlap, the schedule becomes nonserial, but this doesn't necessarily mean the execution is incorrect.

Figure 9.3 Schedule 1: A serial schedule in which To is followed by Ti.

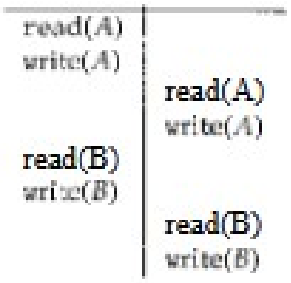


Figure 9.4 Schedule 2: A concurrent serializable schedule

Conflicting operations are those where two operations access the same data item, and at least one of them is a write operation. In nonserial schedules, operations that don't conflict can be swapped without affecting the final result. A schedule is conflict serializable if it can be transformed into a serial schedule by swapping nonconflicting operations. In the example, a nonserial schedule can be rearranged to produce an equivalent serial schedule, demonstrating conflict serializability.

9.5.7 Locking Protocol

To ensure serializability in a database, we can use a locking protocol that governs how locks are acquired and released. There are two main types of locks:

1. **Shared lock (S)**: Allows a transaction to read a data item but not write to it.
2. **Exclusive lock (X)**: Allows a transaction to both read and write a data item.

When a transaction requests a lock on a data item, it must wait if the item is already locked by another transaction. If the item is locked in exclusive mode, the requesting transaction must wait for the lock to be released. If it's locked in shared mode, the transaction can obtain a shared lock and proceed.

The **two-phase locking protocol** ensures serializability by requiring transactions to follow two phases:

1. **Growing phase**: The transaction can acquire locks but cannot release any.
2. **Shrinking phase**: The transaction can release locks but cannot acquire new ones.

While the two-phase locking protocol ensures conflict serializability, it does not guarantee freedom from deadlock. Additionally, some conflict-serializable schedules may not be achievable using this protocol. However, with additional information or by imposing structure on data, performance can be optimized beyond two-phase locking.

9.5.8. Timestamp-Based Protocols

In timestamp ordering, each transaction is assigned a unique fixed timestamp before execution, determining the serializability order. The system ensures that transactions execute in timestamp order by rejecting operations that violate this order. Two main methods for generating timestamps are:

1. **System clock**: Assigning a timestamp based on the system's clock.
2. **Logical counter**: Using a counter that increments with each new transaction.

For each data item, two timestamps are maintained:

- **W-timestamp(Q)**: The largest timestamp of a transaction that wrote to Q.
- **R-timestamp(Q)**: The largest timestamp of a transaction that read Q.

The protocol works as follows:

- **Read(Q)**: If the transaction's timestamp is less than the write timestamp, the read is rejected and the transaction is rolled back.
- **Write(Q)**: If the transaction's timestamp is less than the read or write timestamps, the write is rejected and the transaction is rolled back.

This protocol ensures **conflict serializability** and **freedom from deadlock**, as no transaction waits for another. However, some schedules may be possible under one protocol but not the other.

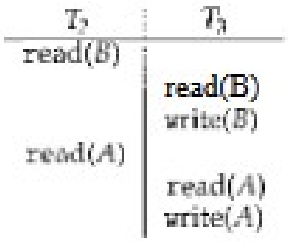


Figure 9.5 Schedule 3: A schedule possible under the timestamp protocol.

- **Summary**

Operating systems provide tools like semaphores, mutexes, and condition variables for controlling access to shared data. In transaction processing, atomicity is ensured using a write-ahead log to record updates. After a system crash, the log helps restore data through undo and redo operations. To improve log recovery, checkpoints are used. Concurrency-control schemes like locking protocols and timestamp ordering ensure serializability in overlapping transactions.

The bounded-buffer, readers-writers, and dining-philosophers problems test synchronization methods. Operating systems like Solaris, Windows XP, and Linux

use semaphores, mutexes, and condition variables to manage shared data access. Transactions, which require atomic execution, are protected using write-ahead logs and rollback mechanisms. Concurrency-control schemes such as locking protocols and timestamp ordering ensure serializability when multiple transactions overlap.

Key Terms

Semaphores**,** Synchronization**,** Monitors ,Deadlock Starvation, monitor, event, checkpoint

9.8. SELF-ASSESSMENT QUESTIONS

1. How do semaphores help in solving synchronization problems compared to user-coded solutions?
2. What are some classic synchronization problems used to test synchronization schemes?
3. How do operating systems like Solaris, Windows XP, and Linux handle shared data synchronization?
4. What is the role of a write-ahead log in ensuring atomicity and recovering from system failures?

9.9 Further Readings

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”,

Seventh Edition, Wiley.

2.William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3.Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4.Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5.Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Mrs. Appikatla Pushpa Latha

CHAPTER 10
DEADLOCKS

Aim and Objectives:

- To understand the concept of deadlocks and their impact on concurrent processes in computer systems.
- To identify the conditions necessary for a deadlock to occur.
- To explore and describe various strategies for preventing or avoiding deadlocks.
- To examine algorithms and techniques for detecting and resolving deadlocks in a system.
- To evaluate the trade-offs and effectiveness of different deadlock management approaches.

Structure:

10.1 INTRODUCTION

Deadlocks in operating systems occur when processes are unable to proceed because they are waiting for resources held by one another, leading to a standstill. The **system model** defines how processes interact with resources, often using resource allocation graphs to analyze potential deadlocks. **Deadlock characterization** identifies the four necessary conditions for deadlock: mutual exclusion, hold-and-wait, no preemption, and circular wait.

To handle deadlocks, operating systems use various strategies, including **prevention**, which ensures that at least one of the conditions for deadlock cannot occur; **avoidance**, which dynamically keeps the system in a safe state (e.g., using the Banker's Algorithm); **detection**, which identifies deadlocks when they occur using tools like wait-for graphs; and **recovery**, which resolves deadlocks through process termination or resource preemption. These methods together help ensure efficient resource management and prevent system inefficiencies.

10.2 SYSTEM MODEL

A system consists of a finite number of resources distributed among competing processes. These resources are grouped into types, each with a certain number of identical instances.

Examples include memory space, CPU cycles, files, and I/O devices like printers. For instance, a system with two CPUs considers the CPU resource type to have two instances. Processes request resources before use and release them afterward, following a sequence of **request, use, and release**.

Deadlocks can arise when processes hold and wait for resources in a cycle, leading to a situation where none can proceed. Deadlocks may involve the same or different resource types, as seen in examples like competing printers or a printer and DVD drive.

To manage resources effectively, the system ensures proper allocation, tracks usage in system tables, and uses mechanisms like semaphores or mutex locks to handle resource requests and releases. Multithreaded applications are particularly prone to deadlocks due to shared resource competition, requiring careful programming to prevent them.

10.3 DEADLOCK CHARACTERIZATION

Deadlocks in operating systems occur when a set of processes becomes indefinitely blocked because each process in the set is waiting for a resource held by another process. Four conditions must hold simultaneously for a deadlock to occur, collectively known as the necessary conditions for deadlock.

1. Mutual Exclusion

At least one resource involved must be held in a non-shareable mode, meaning only one process can use the resource at a time.
Example:

A printer being accessed by one process cannot be accessed by another process until it is released.

2. Hold and Wait

A process holding one or more resources can request additional resources without releasing the ones it already holds.
Example:

A process holding a printer requests access to a scanner while still holding the printer.

3. No Preemption

Resources cannot be forcibly taken away from a process; they must be released voluntarily by the process holding them.

Example:

A process using memory cannot be forced to release it for another process; it must finish its task and release it willingly.

4. Circular Wait

A set of processes exists in a circular chain where each process is waiting for a resource held by the next process in the chain.

Example:

- Process P1 holds Resource R1 and requests Resource R2.
- Process P2 holds Resource R2 and requests Resource R3.
- Process P3 holds Resource R3 and requests Resource R1.

This cycle causes a deadlock as no process can proceed.

Example of Deadlock in an Operating System

Consider a system with two resources: a printer and a DVD drive.

- Process P1 is holding the printer and requests the DVD drive.
- Process P2 is holding the DVD drive and requests the printer.

The processes are now in a deadlock state:

- P1 cannot proceed because it is waiting for the DVD drive held by P2.
- P2 cannot proceed because it is waiting for the printer held by P1.

This scenario satisfies all four necessary conditions for a deadlock, illustrating how resource contention can lead to system-wide blocking if not managed properly.

To mitigate deadlocks, operating systems implement techniques such as deadlock prevention, avoidance (e.g., Banker's Algorithm), detection, and recovery, depending on the system's design and requirements.

10.4 Methods for Handling Deadlocks

Deadlocks in systems can be managed in three main ways: prevention/avoidance, detection/recovery, or ignoring the problem. Deadlock prevention ensures that the system will never enter a deadlock state by eliminating at least one of the necessary conditions for deadlock. Deadlock avoidance requires advance knowledge of resource usage, allowing the system to decide whether a process should proceed based on available resources and potential future requests.

Alternatively, a system may allow deadlocks to occur, then detect and recover from them. This involves detecting deadlocks when they occur and applying recovery methods, such as process termination or resource preemption.

Many operating systems, including Linux and Windows, choose to ignore deadlocks, relying on application developers to handle them. T

his approach is cost-effective but can result in system performance degradation if deadlocks do occur. Some systems combine these methods to address various resource allocation problems and select the optimal strategy for different types of resources.

10.5 Deadlock Prevention

Deadlock is a situation in an operating system where a set of processes is stuck because each process is waiting for a resource that another process in the set holds. Deadlock prevention is one of the strategies to handle deadlocks by ensuring that at least one of the necessary conditions for deadlock does not hold.

Necessary Conditions for Deadlock

For a deadlock to occur, the following four conditions must hold simultaneously:

1. **Mutual Exclusion**:
   At least one resource must be held in a non-shareable mode.
2. **Hold and Wait**:
   A process holding at least one resource is waiting for additional resources held by other processes.
3. **No Preemption**:
   Resources cannot be forcibly taken from a process; they can only be released voluntarily.
4. **Circular Wait**:
   A set of processes is waiting in a circular chain, where each process waits for a resource held by the next process.

Deadlock Prevention Techniques

To prevent deadlocks, the system ensures that at least one of the necessary conditions never holds. Here are the strategies:

1. Mutual Exclusion Prevention

- Avoid making resources exclusively available.

- For example, using *spooling* for resources like printers. Instead of giving the printer directly to a process, the output is written to a spool and printed sequentially.
  Limitation:
  Not all resources can be shared (e.g., semaphores or mutex locks).

2. Hold and Wait Prevention

- Ensure that a process requests all required resources at once, or it releases all held resources before requesting additional resources.
- Example: In a banking application, a process may request access to multiple accounts simultaneously rather than one at a time.

  Drawback:

  This can lead to low resource utilization, as processes may hold resources they do not need immediately.

3. No Preemption Prevention

- Allow resources to be forcibly taken away if the requesting process cannot proceed otherwise.
- Example: In CPU scheduling, if a higher-priority process arrives, the CPU may be preempted from the currently running process.

  Limitation:

  Implementing preemption may not be feasible for all types of resources (e.g., files being modified).

4. Circular Wait Prevention

- Impose a total ordering on resources and require that processes request resources in increasing order.
- Example: If resources R1, R2, R3are ordered, a process holding R1can only request R2 orR3, and not R0.

  Drawback:

  This may lead to inefficiencies and may require significant changes in resource allocation.

Example of Deadlock Prevention

Consider a system with two processes P1 and P2, and two resources R1 and R2:

- **Deadlock Scenario**:
  P1 holds R1 and requests R2, while P2 holds R2 and requests R1. A circular wait occurs, leading to deadlock.
- **Circular Wait Prevention**:
  By assigning an order to resources, say R1<R2 the processes must request resources in that order. If P1 requests R2, it must first release R1. This breaks the circular wait condition.

Deadlock prevention involves compromising on one or more of the four necessary conditions for deadlock. While effective, these strategies often lead to reduced resource utilization or increased complexity. Hence, a balance between prevention and performance must be achieved.

10.6 DEADLOCK AVOIDANCE

Deadlock avoidance is a strategy for handling deadlocks in an operating system. It ensures that the system will never enter an unsafe state, where deadlock becomes possible. Unlike deadlock prevention, which restricts resource allocation policies, avoidance requires additional runtime information to make decisions about resource allocation.

Safe and Unsafe States

- **Safe State**:
  A state where the system can allocate resources to all processes in some order without entering a deadlock.
- **Unsafe State**:
  A state where the system cannot guarantee that all processes will complete without deadlock, even though no deadlock exists yet.

Necessary Conditions for Deadlock Avoidance

Deadlock avoidance works by dynamically analyzing the system's resource allocation to ensure that it remains in a safe state. The **Banker's Algorithm**, developed by EdsgerDijkstra, is a popular technique for this purpose.

The Banker's Algorithm

The Banker's Algorithm works by considering:

1. **Maximum Needs**:
   Each process declares its maximum resource requirements in advance.
2. **Current Allocation**:

The resources currently allocated to each process.

3. **Available Resources**:

   The resources currently available in the system.

The system uses this information to determine whether allocating resources to a process will leave the system in a safe state.

Steps:

1. Calculate the **Need** matrix: Need[i][j]=Maximum[i][j]−Allocation[i][j]
2. Check if the current request can be granted:
   - A request is valid if:
     - Request[i][j]≤Need[i][j]
     - Request[i][j]≤Available[j]
   - Tentatively allocate the resources and check for a safe state.
3. If a safe sequence exists after the tentative allocation, grant the request. Otherwise, deny it.

Example of Deadlock Avoidance

Problem:

Consider a system with:

- **Processes**: P1,P2,P3
- **Resources**: A,B
- **Instances of Resources**:A=10,B=5

Input:

Process	Maximum Need (A, B)	Allocation (A, B)	Need (A, B)
P_1	(7, 5)	(2, 1)	(5, 4)
P_2	(3, 2)	(2, 1)	(1, 1)
P_3	(9, 0)	(3, 0)	(6, 0)

Available: (3, 3)

Scenario:

1. P1P_1P1requests (3,2).
   - Check if the request can be granted:
     - (3,2)≤Need[P1]:(3, 2) ≤(5,4) → True

- (3,2)≤Available:(3, 2)≤(3,3) → True
- Tentative allocation:
  - Available: (3,3)−(3,2)=(0,1)
  - Allocation for P1: (2,1)+(3,2)=(5,3)
  - Need for P1: (5,4)−(3,2)=(2,2)

2. Check for a safe state:
   - Processes can execute in the orderP2,P1,P3, completing without deadlock.

Since the system remains in a safe state, the request is granted.

Advantages of Deadlock Avoidance

1. **No Restrictive Policies**:

   Resources are allocated more flexibly compared to deadlock prevention.

2. **Dynamic Decision-Making**:

   Resource allocation decisions are made based on the current state, ensuring optimal utilization.

Limitations of Deadlock Avoidance

1. **Advance Knowledge**:
   Requires processes to declare maximum resource needs upfront, which is not always feasible.
2. **High Overhead**:
   Constantly checking for a safe state can be computationally expensive.
3. **Limited Scalability**:
   Works well for systems with a small number of resources and processes but becomes complex for large systems.

Deadlock avoidance, particularly using the Banker's Algorithm, ensures the system does not enter an unsafe state. While effective, it requires precise information about resource needs and may incur significant runtime overhead. Balancing its practicality with system requirements is key to its successful implementation.

10.7 DEADLOCK DETECTION

Deadlock detection is a process used by an operating system to determine whether a deadlock has occurred, and if so, which processes are involved. Unlike deadlock prevention or avoidance, where the system takes proactive measures to avoid

deadlocks, deadlock detection allows the system to detect when a deadlock has occurred and take corrective actions.

1. **Resource Allocation Graph (RAG)**:
    - In the detection approach, the system uses a **resource allocation graph** to track the state of resource allocation.
    - This graph includes:
        - **Processes (P)**:
        - Represented as nodes.
        - **Resources (R)**: Also represented as nodes.
        - **Edges**:
            - **Request edge (P → R)**: A process (P) requests a resource (R).
            - **Assignment edge (R → P)**: A resource (R) is assigned to a process (P).
    - A cycle in the resource allocation graph indicates the presence of a deadlock, where each process in the cycle is waiting for a resource held by the next process in the cycle.
2. **Cycle Detection**:
    - The operating system periodically checks for cycles in the resource allocation graph to detect deadlocks. If there is a cycle, it means that the processes in the cycle are deadlocked because they are all waiting for resources held by each other.

Deadlock Detection Algorithms:

- **Single Instance of Each Resource Type**:
    - In systems with only one instance of each resource type, the **Resource Allocation Graph (RAG)** can be used. A cycle in the graph indicates a deadlock.
    - When a process requests a resource, if the resource is already held, the system checks the graph to see if assigning the resource would cause a cycle. If a cycle is detected, deadlock is present.
- **Multiple Instances of Each Resource Type**:
    - In systems with multiple instances of each resource type, **Wait-for Graphs** are used.
    - A wait-for graph is derived from the resource allocation graph by removing the resource nodes and focusing only on the processes.
    - If there is a cycle in the wait-for graph, then a deadlock is present.

Example of Deadlock Detection:

Consider a system with **3 processes (P0, P1, P2)** and **2 resource types (A, B)**.

- **Resource Allocation**:
  - P0 holds 1 instance of resource A and requests 1 instance of resource B.
  - P1 holds 1 instance of resource B and requests 1 instance of resource A.
  - P2 holds no resources but requests 1 instance of resource A and 1 instance of resource B.

Step-by-step detection:

1. **Initial State**:
   - **P0** holds **A**, requests **B**.
   - **P1** holds **B**, requests **A**.
   - **P2** holds nothing, requests **A** and **B**.
2. **Resource Allocation Graph**:
   - Request edges from **P0 → B**, **P1 → A**, and **P2 → A, B**.
   - Assignment edges: **A → P0**, **B → P1**.

     The graph looks like this:
     P0 → B ← P1 → A ← P2
     |_____|

   - **Cycle Detection**:
     1. A cycle exists in the graph: **P0 → B → P1 → A → P0**. This cycle indicates a deadlock because P0 is waiting for P1 to release resource A, P1 is waiting for P0 to release resource B, and P2 is also waiting for both resources.
   - **Deadlock Conclusion**:
     1. Since a cycle exists, we can conclude that the system is in a deadlocked state, with processes **P0**, **P1**, and **P2** involved.

Deadlock Recovery:

Once deadlock is detected, the operating system can take various actions to recover from it, such as:

1. **Terminating Processes**:

   The system may decide to terminate one or more processes involved in the deadlock.
2. **Resource Preemption**:

   Resources may be forcibly taken from a process and reassigned to other processes to break the cycle.

3. **Rollback**:

   A process can be rolled back to a safe state before it entered the deadlock.

Deadlock detection algorithms are crucial for identifying when a system enters a deadlocked state, allowing the system to take corrective actions. By using resource allocation graphs and cycle detection techniques, operating systems can effectively manage and detect deadlocks, especially in environments where deadlock prevention or avoidance isn't feasible.

10.8 RECOVERY FROM DEADLOCK

When a detection algorithm identifies a deadlock, several alternatives are available for recovery. These options can either involve human intervention or allow the system to recover automatically. Two main strategies for breaking a deadlock are **process termination** and **resource preemption**.

1. Process Termination

To eliminate deadlocks by aborting processes, there are two approaches:

- **Abort all deadlocked processes**:

  This method terminates all processes involved in the deadlock cycle, which guarantees that the cycle will be broken. However, this comes at a high cost as any work done by the terminated processes is lost, requiring recomputation.

- **Abort one process at a time**:

  In this method, processes are terminated one by one until the deadlock cycle is eliminated. While it avoids terminating all processes, it introduces overhead as the deadlock detection algorithm must be rerun after each process termination.

Challenges with process termination:

- If a process was updating a file or performing I/O operations, terminating it may leave the system in an inconsistent state, requiring additional effort to recover the system.
- Deciding which process to terminate involves considering several factors, such as:
  - **Priority of the process**
  - **Progress made** (e.g., how much time has been spent and how much more is needed)
  - **Resource utilization** (e.g., how easy it is to preempt resources)
  - **Number of processes to be terminated**
  - **Type of process** (interactive vs batch)

2. Resource Preemption

Deadlocks can also be resolved by preempting resources from one or more processes involved in the deadlock and reallocating them to others.

Key issues in resource preemption:

1. **Selecting a victim**:

   The system needs to decide which resources and processes to preempt. This decision must minimize cost, considering factors like:

   - The amount of resources held by a process
   - The time the process has already executed
   - The difficulty of preempting certain resources
   - The number of processes that need to be preempted

2. **Rollback**:

   Once a resource is preempted, the affected process cannot continue its execution as it is missing required resources. The process must be rolled back to a safe state and restarted. While rolling back to the safest state (i.e., aborting the process) is an option, it may be more efficient to roll it back only as far as necessary to break the deadlock. This approach, however, requires maintaining detailed state information for processes.

3. **Starvation**:

   To prevent starvation (where the same process is continually preempted), the system must ensure that a process cannot be selected as a victim too often. A common solution is to limit the number of times a process can be preempted before it is allowed to complete its task.

   Both process termination and resource preemption provide mechanisms for recovering from deadlocks, each with its own set of trade-offs and complexities. Deciding between these methods depends on factors like the system's resources, process priorities, and the cost of rollback or termination. Effective management of these recovery strategies is crucial to maintaining system stability and fairness.

Advantages of Starvation

1. **Priority Handling:** Ensures that critical or high-priority processes are executed quickly, improving responsiveness for time-sensitive tasks.
2. **Efficient Resource Utilization:** Allocates resources to processes with higher priorities, potentially increasing overall system performance for critical workloads.

3. **Improved System Responsiveness:** High-priority processes, such as those in real-time systems, get faster access to resources.

Disadvantages of Starvation

1. **Process Inefficiency:** Lower-priority processes may never get the resources they need, leading to wasted CPU cycles and reduced throughput.
2. **Unfairness:** The system may become unfair, as certain processes are consistently neglected, violating fairness principles in scheduling.
3. **Deadlock Risk:** In some cases, prolonged starvation can contribute to deadlock if processes waiting indefinitely block others.
4. **Reduced User Satisfaction:** Starvation can negatively impact user experience, especially in interactive systems where low-priority tasks are delayed excessively.

To reduce starvation, strategies such as **aging** (gradually increasing the priority of waiting processes) and fair scheduling algorithms like **Round Robin** or **Fair Queueing** can be implemented. These approaches aim to balance system efficiency with fairness.

10.9 SUMMARY

Deadlocks occur in computer systems when a set of concurrent processes becomes stuck, each waiting for resources held by another, preventing the completion of their tasks. Understanding deadlocks involves recognizing the conditions that cause them: mutual exclusion, hold and wait, no preemption, and circular wait.

To address deadlocks, various strategies can be applied, including prevention, avoidance, detection, and recovery. Deadlock prevention involves eliminating one or more of the conditions that lead to deadlocks, while avoidance ensures the system remains in a safe state through careful resource allocation. Detection and recovery focus on identifying deadlocks after they occur and taking corrective actions to resolve them. Each approach has its advantages and challenges, making their application dependent on the specific requirements of the system.

10.10 KEY TERMS

Deadlock, Request, Release, Mutual Exclusion, Circular Wait, prevention, avoidance, detection, recovery

10.11 SELF-ASSESSMENT QUESTIONS

1. What is a deadlock, and how does it affect the execution of concurrent processes?
2. What are the four necessary conditions for a deadlock to occur?
3. How can deadlocks be prevented in a computer system?

4. What is the difference between deadlock prevention and deadlock avoidance strategies?
5. What techniques are used to detect and recover from deadlocks once they occur?

10.12 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. "Operating System Principles", Seventh Edition, Wiley.
2. William Stallings, "Operating Systems – Internals and Design Principles", Fifth Edition, Pearson Education (2007)
3. Achyut S Godbole, "Operating Systems", Second Edition, TMH (2007).
4. Flynn/McHoes, "Operating Systems", Cengage Learning (2008).
5. Deitel&Deitel, "Operating Systems",Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Mrs. Appikatla Pushpa Latha

CHAPTER 11

MEMORY MANAGEMENT STRATEGIES

Aim and Objectives:

- To provide a detailed description of different ways to organize memory hardware.
- To discuss various memory-management techniques, such as paging and segmentation.
- To explain the Intel Pentium architecture, which supports both pure segmentation and segmentation with paging

Structure:

11.1 INTRODUCTION

Memory management is a fundamental aspect of operating systems that involves efficiently allocating, utilizing, and reclaiming memory resources to meet the needs of various processes. The primary goal is to optimize the use of limited memory resources while ensuring system stability and performance.

In modern computing, memory management plays a crucial role in bridging the gap between fast, volatile main memory (RAM) and slower, non-volatile storage.

Effective memory management ensures that multiple processes can execute concurrently, providing smooth multitasking and efficient resource utilization.

11.1.1 Background

Memory is essential to modern computer systems, functioning as a large array of addresses accessed by the CPU during instruction execution. The CPU fetches instructions and data from memory based on the program counter, while the memory unit handles only address streams without knowing their purpose. Key topics in memory management include binding symbolic addresses to physical ones, differentiating logical and physical addresses, and dynamically loading code and libraries. Instruction cycles typically involve fetching, decoding, and executing instructions, often requiring memory access for operands and results.

11.2. BASIC HARDWARE

The CPU can directly access only registers and main memory, with no machine instructions supporting direct disk access. Data and instructions must be in registers or main memory to be processed. Registers offer faster access, typically within one CPU cycle, while main memory access may require multiple cycles, causing processor stalls. To mitigate this, fast intermediate storage called cache is used. Memory protection ensures correct operation, isolating processes from each other and the operating system.

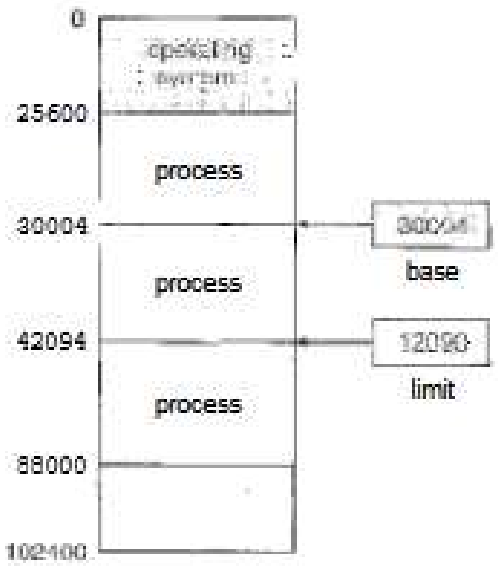


Figure 11.1 A base and a limit register define a logical address space.

This is achieved using base and limit registers, which define the legal memory range for a process. Any unauthorized memory access triggers a system trap, treating it as an error. Only the operating system, operating in kernel mode, can modify these registers, ensuring secure and controlled memory access

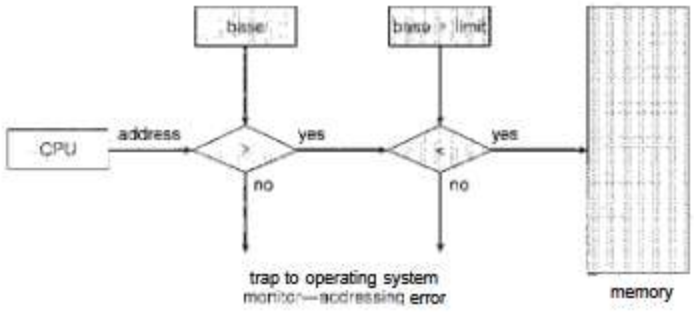


Figure 11.2 Hardware address protection with base and limit registers

11.2.1 Address Binding

Address binding is the process of mapping symbolic program addresses to actual physical memory addresses, enabling seamless program execution. This concept is crucial in operating systems, as it ensures that instructions and data are correctly located and accessed in memory. The binding process can occur at various stages during program execution, offering different levels of flexibility and efficiency.

Types of Addresses in a Program

To understand address binding, it is essential to distinguish between different types of addresses used in a program:

1. **Symbolic Addresses**:
   - Found in source code, such as variable names and function labels.
   - These addresses are abstract and do not correspond directly to physical memory.
2. **Logical (Virtual) Addresses**:
   - Generated by the CPU during program execution.
   - These addresses form the **logical address space** visible to the process.
   - Logical addresses are independent of physical memory and are mapped to actual locations using address binding.
3. **Physical Addresses**:
   - Actual locations in main memory.
   - These addresses are visible to the memory unit and refer to physical storage in RAM.

Address Binding Process

The process of binding symbolic or logical addresses to physical addresses can occur at different stages of a program's lifecycle:

1. Compile-Time Binding:

- Occurs when the program is compiled.
- If the memory location of the program is known at compile time, the compiler translates symbolic addresses directly into absolute physical addresses.
- **Limitation**: The program must always run in the same memory location.
- **Example**: Embedded systems or simple batch systems where programs are loaded into predefined memory locations.

2. Load-Time Binding:

- Occurs when the program is loaded into memory.
- The loader translates symbolic addresses into relocatable addresses, which are adjusted to the actual memory location at load time.
- Provides flexibility by allowing programs to load into different memory areas.
- **Example**: Systems with relocatable code, where programs are not fixed to specific memory addresses.

3. Execution-Time Binding:

- Performed dynamically during program execution.
- Logical addresses are translated to physical addresses by the **Memory Management Unit (MMU)**.
- Supports features like dynamic relocation and virtual memory, enabling programs to move between memory locations without altering their execution.
- **Example**: Multitasking systems and virtual memory systems where processes share memory.

Logical Address Space vs. Physical Address Space

- **Logical Address Space**:
  - The set of all logical addresses a process can generate.
  - Represents the program's view of memory, defined by the program's code and data structure.
- **Physical Address Space**:
  - The set of all physical addresses available in the system.
  - Represents the actual locations in main memory where data is stored.

In systems with execution-time binding, logical and physical address spaces differ significantly, enabling features like process isolation and memory abstraction.

Importance of Address Binding

1. **Dynamic Relocation**: Allows programs to be moved in memory during execution without affecting their functionality.

2. **Flexibility**: Supports processes in dynamic environments, such as multitasking and virtual memory systems.
3. **Efficient Resource Utilization**: Enables operating systems to allocate and manage memory resources more effectively.
4. **Abstraction**: Separates program memory from physical memory, simplifying program design and development.

Address Binding in Modern Systems

Modern operating systems rely heavily on execution-time binding due to its flexibility and ability to support advanced memory management techniques like **paging**, **segmentation**, and **virtual memory**. These techniques allow processes to use more memory than physically available, share memory efficiently, and provide isolation between processes.

Address binding is a core concept in memory management, ensuring that symbolic and logical addresses are accurately mapped to physical memory during program execution. By facilitating features like dynamic relocation and memory abstraction, address binding contributes to the efficiency, flexibility, and scalability of modern operating systems.

11.2.2 Logical Versus Physical Address Space

In operating systems, the **logical address space** and **physical address space** are fundamental concepts in memory management. They help distinguish between how a program perceives memory and how the memory is physically implemented. Understanding the distinction between these address spaces is essential for efficient memory management, program execution, and system design.

Logical Address Space

The **logical address space**, also known as the **virtual address space**, refers to the set of addresses that a process can use to reference memory. These addresses are generated by the CPU during program execution. Logical addresses are an abstraction provided by the operating system to simplify memory management and enhance flexibility.

Characteristics of Logical Address Space:

1. **Generated by the CPU**:
    - These addresses are part of the program's perspective and are used during instruction execution.

2. **Defined by the Program**:
    - Logical addresses depend on the program's structure, including its variables, code, and data segments.
3. **Independent of Physical Memory**:
    - Logical addresses do not directly correspond to physical locations in memory.
4. **Isolated for Each Process**:
    - Each process operates within its own logical address space, providing isolation and security.

Example of Logical Address Space:

A program might access memory locations such as 0x0010, 0x0020, etc., but these are logical addresses that need to be mapped to actual physical memory by the operating system.

Physical Address Space

The **physical address space** refers to the set of all addresses that are available in the physical memory (RAM) of the computer. Physical addresses represent the actual locations in memory where data is stored.

Characteristics of Physical Address Space:

1. **Visible to the Memory Unit**:
    - Physical addresses directly reference memory cells in the RAM.
2. **Dependent on Hardware**:
    - The size of the physical address space is determined by the amount of physical memory installed in the system.
3. **Shared Resource**:
    - Physical memory is shared among all processes and managed by the operating system.

Example of Physical Address Space:

If a computer has 4 GB of RAM, its physical address space will range from 0x00000000 to 0xFFFFFFFF in a 32-bit addressing system.

Relationship Between Logical and Physical Address Space

The operating system and the **Memory Management Unit (MMU)** are responsible for translating logical addresses into physical addresses. This translation occurs dynamically and depends on the memory management strategy employed.

Scenarios in Logical and Physical Address Space:

1. **Identical Address Spaces**:
   - In simple systems without virtual memory, logical and physical addresses are identical.
   - Example: Single-tasking systems or those using compile-time address binding.
2. **Different Address Spaces**:
   - In systems using multitasking, virtual memory, or execution-time address binding, logical and physical addresses differ.
   - The MMU translates logical addresses to physical addresses at runtime.

Benefits of Separating Logical and Physical Address Space

1. **Process Isolation**:
   - Logical addresses ensure that each process operates in its own memory space, preventing interference from other processes.
2. **Memory Abstraction**:
   - Logical addresses abstract the complexity of physical memory, allowing programs to be written without knowledge of hardware details.
3. **Dynamic Memory Management**:
   - Enables features like virtual memory, paging, and segmentation, which enhance memory utilization and flexibility.
4. **Scalability**:
   - Processes can use more memory than physically available, as logical addresses can extend beyond physical memory limits (e.g., with virtual memory).

Practical Example of Logical vs. Physical Address Translation

Scenario:

A program accesses a variable at a logical address 0x1234.

- The MMU translates 0x1234 to the physical address 0xA234 in RAM.
- The operating system determines this mapping using page tables or segment tables, depending on the memory management strategy.

The distinction between logical and physical address spaces is crucial in modern operating systems. Logical addresses provide an abstraction that simplifies program development and enables advanced features like virtual memory and process isolation. The translation of logical addresses to physical

addresses ensures efficient and secure utilization of memory resources, making it a cornerstone of memory management strategies.

11.2.3 Dynamic Loading

Dynamic loading is a memory management technique where a program loads its routines or modules into memory only when they are needed during execution. This approach optimizes memory usage by avoiding the unnecessary loading of unused program components, thereby improving the efficiency of system resource utilization.

In traditional loading methods, all parts of a program are loaded into memory before execution begins. In contrast, **dynamic loading** defers the loading of certain parts, such as functions or libraries, until they are explicitly called. This technique is particularly useful for large programs where not all functions are required in every execution instance.

Mechanism of Dynamic Loading

Dynamic loading involves several key steps:

1. **Program Segmentation**:
   - The program is divided into smaller, independent modules or segments.
   - Only the main module (entry point) is loaded initially.
2. **On-Demand Loading**:
   - When a specific routine or module is called, the operating system loads it into memory from secondary storage (e.g., disk).
   - The operating system updates the program's address space to include the newly loaded module.
3. **Symbol Resolution**:
   - The addresses of dynamically loaded routines are resolved at runtime.
   - This process may involve relocating code to fit the program's memory layout.
4. **Execution and Retention**:
   - Once loaded, the routine executes normally.
   - Depending on the system's memory management policy, the loaded module may be retained in memory for future use or unloaded after execution.

Advantages of Dynamic Loading

1. **Efficient Memory Utilization**:
   - Only the necessary parts of a program are loaded into memory, reducing memory wastage.

2. **Faster Startup Time**:
    - Programs start executing without waiting for all modules to load, improving responsiveness.
3. **Modular Programming Support**:
    - Encourages the development of modular and reusable code, as routines can be loaded independently.
4. **Support for Large Programs**:
    - Allows the execution of large programs that may not fit entirely in memory.
5. **Enhanced Flexibility**:
    - Enables the addition or update of program modules without recompiling the entire program.

Disadvantages of Dynamic Loading

1. **Runtime Overhead**:
    - Loading modules during execution introduces additional overhead, which may slightly degrade performance.
2. **Complex Implementation**:
    - Requires sophisticated operating system support for managing address space and symbol resolution.
3. **Error Handling**:
    - Errors during module loading or symbol resolution can disrupt program execution.

Use Cases of Dynamic Loading

1. **Large Software Applications**:
    - Applications with numerous features (e.g., office suites) load only the required components, like a spell checker or advanced formulas, when needed.
2. **Operating Systems**:
    - Modern operating systems dynamically load kernel modules or device drivers to handle specific hardware or tasks.
3. **Shared Libraries**:
    - Programs often load shared libraries dynamically, allowing multiple processes to use the same library code simultaneously.

Example of Dynamic Loading

Scenario:

A text editor program consists of a main editing module and additional features like a grammar checker.

- **At Startup**: Only the main module is loaded.
- **When Grammar Check is Invoked**: The operating system loads the grammar checker module dynamically.
- **After Use**: The module may stay in memory or be unloaded, depending on the memory management policy.

Dynamic loading is a powerful memory management strategy that enhances the efficiency and flexibility of program execution. By loading only the required parts of a program on demand, it minimizes memory usage and supports the development of modular and extensible software. While it introduces some runtime overhead, its benefits in optimizing memory and enabling large-scale applications make it a critical feature in modern operating systems.

11.2.4 Dynamic Linking and Shared Libraries

Dynamic linking is the process of linking program modules (such as functions or libraries) to a program at runtime, rather than during the compilation process. When a program is compiled, it contains references to external libraries, but the actual linking of these references to the library code occurs only when the program is executed.

Key Characteristics of Dynamic Linking:

1. **Runtime Linking**:
   - The actual address resolution happens when the program is executed, rather than during compilation.
2. **Efficiency**:
   - Only the necessary libraries are linked at runtime, allowing the system to optimize memory use by loading libraries as needed.
3. **Modular Design**:
   - Programs can use external libraries without incorporating all their code into the main executable, making them more modular.
4. **Separate Compilation**:
   - The program can be compiled independently of its dependencies, allowing for easier maintenance and versioning of libraries.

Benefits of Dynamic Linking:

- **Memory Efficiency**: Shared libraries are loaded into memory only when needed, and multiple programs can share the same instance of a library, reducing memory usage.
- **Reduced Program Size**: The program itself doesn't need to include all the code of the libraries it uses, resulting in smaller executable files.
- **Easy Updates**: Libraries can be updated independently of the programs that use them, which makes patching and maintaining systems easier without recompiling the entire application.
- **Code Reusability**: Common functions or routines are stored in shared libraries, and different applications can link to them, promoting code reuse across multiple programs.

Shared Libraries

Shared libraries are dynamic libraries that are stored separately from the program's executable code but are accessible to the program during execution. Multiple programs can share a single copy of a shared library in memory, which helps optimize memory usage and facilitates efficient code sharing.

Types of Shared Libraries:

1. **Dynamic Link Libraries (DLLs)**:
    - Primarily used in Windows operating systems, DLLs are shared libraries that contain code and data that can be used by multiple applications simultaneously.
2. **Shared Object Files (.so)**:
    - Used in Unix-like operating systems (Linux, macOS), .so files are the equivalent of DLLs in Windows and can be loaded dynamically by programs during execution.

How Shared Libraries Work:

- When a program starts, it references the functions or routines in shared libraries using symbolic names.
- The operating system loads the shared libraries into memory only when those specific functions are called.
- If the library is already loaded into memory (because another program is using it), the operating system reuses the existing memory copy rather than loading it again.

This mechanism significantly reduces the memory footprint, especially when multiple programs use the same libraries.

Dynamic Linking and Shared Libraries in Action

Let's consider an example where a program uses dynamic linking and shared libraries:

1. **Program Compilation**:
   - During compilation, the program does not include the code for the libraries it uses. Instead, it contains references to external library functions, such as math.sin() or graphics.drawLine().
2. **Runtime Linking**:
   - When the program is executed, the operating system dynamically links the program to the shared libraries. The system resolves the addresses for the functions in the shared libraries, so the program can call them.
3. **Shared Library Use**:
   - If multiple programs require the same library, the operating system loads the library once into memory and allows each program to use that instance, reducing memory usage.

Example:

Consider a scenario where three programs (A, B, and C) all use a shared library libgraphics.so, which contains a function drawCircle():

- When the first program (A) runs, the operating system loads libgraphics.so into memory.
- Program A calls drawCircle(), and the function is executed from the shared library in memory.
- Program B and Program C are started afterward, and the operating system does not reload libgraphics.so. Instead, both programs reuse the already loaded copy of the library, saving memory and time.

Advantages of Dynamic Linking and Shared Libraries

1. **Reduced Memory Usage**:
   - Shared libraries allow multiple programs to use the same library code in memory, reducing the overall memory footprint of the system.
2. **Modularity**:
   - By separating functionality into shared libraries, programs can be designed in a modular way. This allows easier updates and debugging as individual modules can be modified without affecting the entire system.

3. **Code Reusability**:

- Shared libraries allow common functionality to be reused across different programs without duplicating code in each program's executable. This saves disk space and reduces redundancy.

4. **Faster Updates and Bug Fixes**:
   - Updating a shared library (such as fixing a bug in libgraphics.so) automatically updates all programs using that library, as long as the library interface remains consistent.
5. **Smaller Executable Size**:
   - Programs that use dynamic linking and shared libraries have smaller executable sizes, as they don't need to contain all the code of the libraries they use.

Disadvantages of Dynamic Linking and Shared Libraries

1. **Runtime Overhead**:
   - Linking libraries at runtime can introduce slight overhead, as the operating system must locate and load the required shared libraries.
2. **Compatibility Issues**:
   - If the interface or behavior of a shared library changes (e.g., due to an update), programs that depend on that library may break if the new version is not backward-compatible.
3. **Version Management**:
   - Managing multiple versions of the same shared library can become complex. Some programs may depend on older versions of a library, while others may require the latest version.

Dynamic linking and shared libraries are essential tools in modern operating systems, offering significant advantages in terms of memory efficiency, modularity, and code reusability. These techniques allow programs to be smaller, load libraries on demand, and share common code across multiple processes, reducing redundancy and improving system performance. While there are some challenges, such as potential compatibility issues and runtime overhead, dynamic linking and shared libraries remain a fundamental aspect of efficient memory management in modern systems.

11.3 SWAPPING

Swapping is a memory management technique that involves moving processes between main memory and secondary storage (usually disk) in order to optimize the use of system memory and ensure that the operating system can run multiple processes concurrently, even if there is not enough physical memory to hold them all at once. Swapping helps maximize the utilization of available RAM by allowing processes to temporarily be moved out of memory to free up space for other processes.

In a system using swapping, processes are swapped in and out of the main memory to the secondary storage (often referred to as the **swap space** or **page file**). When the system detects that memory is becoming scarce, it moves a process or part of a process (often a part of a process's memory called a **page**) to the swap space, freeing up memory for other processes. Later, when the swapped-out process needs to execute again, it is swapped back into the memory.

Key Operations in Swapping

1. **Process Swap In**:
    - When a process is ready to run but there isn't enough free memory, part or all of a process can be swapped from the secondary storage into main memory.
2. **Process Swap Out**:
    - When a running process needs to be moved out of memory to free space, it is swapped to the secondary storage.
3. **Swapping Mechanism**:
    - The operating system must keep track of processes that are in memory and those that are on disk. It uses **swap tables** to maintain information about which process is in which part of memory or disk.

Types of Swapping

1. **Full Swapping**:
    - In **full swapping**, the entire process is moved between memory and secondary storage. This is typically done when there is not enough free space in memory to accommodate a new process.
2. **Partial Swapping**:
    - In **partial swapping**, only part of the process (such as a specific memory segment or a page) is swapped to secondary storage, while other parts of the process remain in memory. This is more efficient than full swapping, as it allows the most frequently used parts of a process to stay in memory.

Swapping Process

The general process of swapping can be described in the following steps:

1. **Identify the Process to Swap**:
    - The operating system identifies a process that is not currently being used or is in a "blocked" state, making it a candidate for swapping out.
2. **Move Process to Secondary Storage**:

   - The selected process is then transferred from memory to the swap space. During this time, the state of the process (register values, memory pages, etc.) is saved so it can be resumed later.
3. **Allocate Memory for New Processes**:
   - The freed-up memory is then allocated to the new process that needs to run. The operating system adjusts the memory allocation accordingly.
4. **Swap the Process Back Into Memory**:
   - When the swapped-out process is needed again, it is swapped back into memory. The operating system loads the process from the swap space and restores its state, allowing the process to continue from where it left off.

Advantages of Swapping

1. **Maximizes Memory Utilization**:
   - Swapping allows the system to continue running even when there is not enough physical memory for all processes. By swapping out processes that are not immediately needed, the system ensures that active processes can be allocated memory.
2. **Supports Multiprogramming**:
   - Swapping allows multiple processes to run concurrently by efficiently utilizing available physical memory, making it easier to implement multiprogramming systems where more than one process can execute at a time.
3. **Enables Large Programs**:
   - Even if a program is too large to fit entirely in memory, swapping allows the system to run such programs by keeping only a portion in memory at any time.
4. **Improves System Responsiveness**:
   - When memory is full, swapping allows the operating system to free up space for new, more important processes, ensuring that the system remains responsive and functional.

Disadvantages of Swapping

1. **Performance Overhead**:
   - The process of swapping data between memory and disk introduces significant overhead, as disk access is much slower than accessing data in memory. Frequent swapping, also known as **thrashing**, can lead to a system slowdown.
2. **Disk I/O Bottleneck**:
   - If the system is swapping processes in and out of memory too frequently, disk I/O can become a bottleneck, causing delays in process execution and reduced overall performance.

3. **Limited by Disk Space**:
    - Swapping relies on disk space to store swapped-out processes. If there is not enough space in the swap area, the system may be unable to swap out processes, which could lead to memory allocation issues.
4. **Fragmentation**:
    - Swapping can cause fragmentation of both the physical memory and the swap space, as processes are moved in and out in arbitrary locations, leading to inefficient memory utilization.

Example of Swapping

Consider a computer with 4GB of RAM and 6GB of active processes running at the same time. The operating system can use swapping to manage memory by performing the following steps:

1. **Initial Condition**:
    - The system is running multiple processes, each requiring memory, but there is only 4GB of RAM available. Some processes must be swapped out to free up space for new processes.
2. **Process Swapping**:
    - The operating system identifies a process that is not currently in use and swaps it out to secondary storage (e.g., hard disk).
3. **Memory Allocation**:
    - The system then loads a new process into the freed-up memory space.
4. **Swapping Back**:
    - When the swapped-out process is needed again, the operating system swaps it back into memory and removes another process that is not in use.

Swapping is an important technique in memory management, enabling the operating system to manage memory efficiently by moving processes in and out of main memory as needed. While it allows the system to run more processes than the available physical memory can hold, it can lead to performance overhead and complications like thrashing if not properly managed. Despite its disadvantages, swapping is crucial for systems that require multitasking and support for large applications, particularly in environments with limited physical memory.

11.4 CONTIGUOUS MEMORY ALLOCATION

Contiguous memory allocation is a method where the main memory is divided into two partitions: one for the operating system and one for user processes. Typically, the operating system is placed in low memory, often due to the location of the interrupt vector. In this approach, several user processes are allocated memory in contiguous blocks. Each process is loaded into a single contiguous section of memory, which simplifies memory management

but can lead to issues like fragmentation. The challenge lies in efficiently allocating memory to processes waiting to be loaded while minimizing wasted space.

11.5 MEMORY MAPPING AND PROTECTION

Memory mapping and protection are achieved using relocation and limit registers. The relocation register holds the smallest physical address, while the limit register defines the range of logical addresses. When a process is selected for execution, the dispatcher loads these registers, ensuring that any address generated by the CPU is within bounds, thus protecting both the operating system and other processes. This scheme allows the operating system's size to change dynamically, accommodating transient operating-system code when necessary.

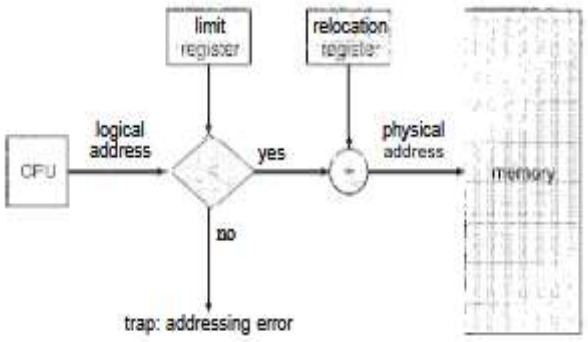


Figure 11.3 Hardware support for relocation and limit registers

11.6 MEMORY ALLOCATION

Memory allocation involves dividing memory into fixed-sized partitions where each partition can hold one process. The degree of multiprogramming is limited by the number of partitions. In the fixed-partition scheme, the operating system maintains a table to track which parts of memory are available. When a process arrives, it is allocated memory from available partitions, and when it terminates, the partition is freed. The operating system searches for a hole large enough to fit a process and allocates memory accordingly. If a hole is too large, it is split. Common strategies to allocate memory include first-fit, best-fit, and worst-fit, with first-fit generally being faster.

11.7 FRAGMENTATION

Memory allocation strategies like first-fit and best-fit suffer from external fragmentation, where free memory is fragmented into small, non-contiguous blocks, making it difficult to allocate memory efficiently. Despite having enough total memory, the fragmented spaces may not be usable for new processes. This fragmentation can be reduced by using techniques like compaction, which moves processes to create larger free blocks, though this can be costly. Alternatively, non-contiguous allocation methods, such as paging and segmentation, can help address external fragmentation. Internal fragmentation, where

small unused spaces are left within partitions, can also occur, and may be minimized by using fixed-size memory blocks.

11.8 SUMMARY

Memory management algorithms in multiprogrammed operating systems range from simple single-user approaches to more complex systems like paged segmentation. The choice of algorithm is largely determined by the hardware capabilities, as all memory addresses generated by the CPU must be checked for legality and mapped to physical addresses.

11.9 KEY TERMS

Address Binding, Logical, Physical , Address Space, Dynamic Loading, Dynamic Linking , Shared Libraries, Swapping, Contiguous Memory Allocation

11.10 SELF-ASSESSMENT QUESTIONS

1. What are the key memory-management algorithms used in multiprogrammed operating systems?
2. How does hardware support influence the choice of memory-management algorithms?
3. What is the impact of performance on memory-management strategies, particularly in paging and segmentation?
4. How does fragmentation affect memory efficiency, and what are the differences between internal and external fragmentation?
5. What role does relocation and compaction play in managing external fragmentation in memory systems?
6. How do swapping, sharing, and protection mechanisms contribute to increasing multiprogramming levels and ensuring system stability?

11.11 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”, Seventh Edition, Wiley.

2.William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3.Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4.Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5.Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Mrs. Appikatla Pushpa Latha

CHAPTER 12

PAGING

Aim and Objectives:

- Understand the basic method and hardware support for paging and its role in memory management.
- Explore memory protection mechanisms and the concept of shared pages in a paging system.
- Study the structure of the page table, including hierarchical paging, hashed page tables, and inverted page tables.
- Learn the concept of segmentation and its basic methods, as well as the hardware support needed.
- Examine the implementation of paging and segmentation on Intel Pentium systems and its application in Linux systems.

Structure:

12.1 INTRODUCTION

Paging and segmentation are fundamental memory management techniques used by operating systems to efficiently manage memory allocation. **Paging** breaks memory into fixed-size blocks called pages and frames, allowing non-contiguous allocation of physical memory. This method is supported by hardware to handle address translation

and avoid fragmentation. **Protection** mechanisms, such as protection bits and valid-invalid bits, ensure memory security. **Shared pages** allow multiple processes to access common memory regions. The **Page Table** structure varies, with methods like hierarchical, hashed, and inverted paging improving efficiency. **Segmentation**, on the other hand, divides memory into variable-sized segments based on logical divisions like code or data, offering flexibility. Hardware support for both techniques is essential for optimizing performance and managing complex address translations. Notably, **Intel Pentium** processors use both segmentation and paging to manage memory, with Linux systems leveraging these features for memory management.

12.2 PAGING

Paging is a memory management technique that allows a process's physical address space to be non-contiguous. This method resolves the significant issue of fitting memory blocks of different sizes onto the backing store, a problem that earlier memory management schemes struggled with. The challenge arose when parts of code or data in main memory needed to be swapped out, requiring space on the backing store. Like main memory, the backing store suffers from fragmentation, but because access is much slower, compaction isn't feasible.

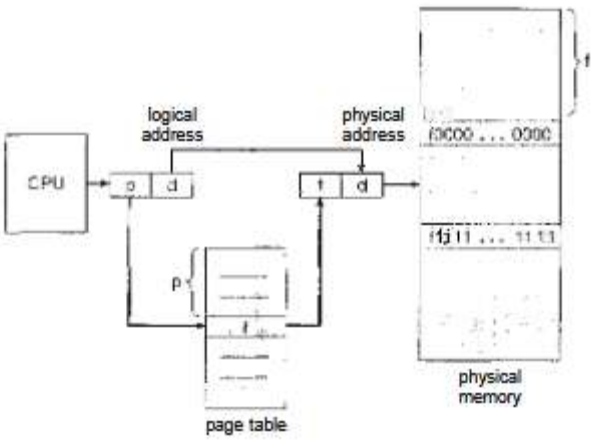


Figure 12.1 Paging hardware.

Due to its advantages over older methods, paging is widely used in modern operating systems. Traditionally, paging support was provided by hardware. However, in recent designs, particularly with 64-bit microprocessors, paging has been implemented through a close integration of hardware and the operating system.

12.2.1 Basic Method

The fundamental method for implementing paging involves dividing physical memory into fixed-sized blocks called frames, while logical memory is divided into blocks of the same size, known as pages. When a process is ready for execution, its

pages are loaded into available memory frames from the backing store, which is also divided into fixed-sized blocks matching the frame size.

Paging hardware works as follows: Each address generated by the CPU is split into two parts: a page number (p) and a page offset (d). The page number is used as an index into the page table, which stores the base address for each page in physical memory. This base address is combined with the page offset to form the final physical memory address.

The page size, defined by the hardware, typically ranges between 512 bytes and 16 MB, with the size being a power of 2. This allows for easy translation of logical addresses into page numbers and offsets. For example, with a logical address space of 2^n and a page size of 2^m, the high-order (n - m) bits of a logical address represent the page number, and the remaining m bits represent the page offset.

To illustrate, consider a system with a 4-byte page size and 32 bytes of physical memory, divided into 8 pages. Logical addresses are mapped to physical addresses based on the page number and frame number in the page table. If a process requires memory that does not align perfectly with page boundaries, internal fragmentation may occur. For example, if a process requires 72,766 bytes and the page size is 2,048 bytes, it would occupy 36 frames, with a small portion of the last frame unused.

While paging eliminates external fragmentation, it can introduce internal fragmentation because memory is allocated in units of entire frames. The overhead involved in managing page-table entries increases with smaller page sizes, but larger page sizes reduce the overhead and improve disk I/O efficiency. Over time, page sizes have generally increased due to growing process sizes and memory capacities. Modern systems typically use page sizes between 4 KB and 8 KB, though some support even larger pages. Some systems, like Solaris, allow for multiple page sizes depending on the data being stored.

Each page-table entry is typically 4 bytes, which can address up to 2^32 physical page frames. For example, with a 4 KB frame size, this setup can address 16 TB of physical memory. When a process is loaded into memory, its pages are mapped to available frames. The operating system checks that enough frames are available and assigns them accordingly, updating the page table as it does so.

Paging creates a clear separation between the user's logical memory and the actual physical memory. The user program views memory as a contiguous block, even though it is scattered across physical memory alongside other programs. This abstraction is maintained by address-translation hardware, which maps logical addresses to physical addresses. The operating system ensures that a process can only access its allocated pages and prevents it from accessing other processes' memory.

The operating system must manage the physical memory, tracking which frames are allocated or free using a structure called the frame table. It also handles the translation of logical addresses to physical ones, especially during system calls. When a process is dispatched to the CPU, the operating system reloads the appropriate page table entries to define the physical memory map. As a result, paging increases context-switch time, as the page table must be loaded or updated with each context switch.

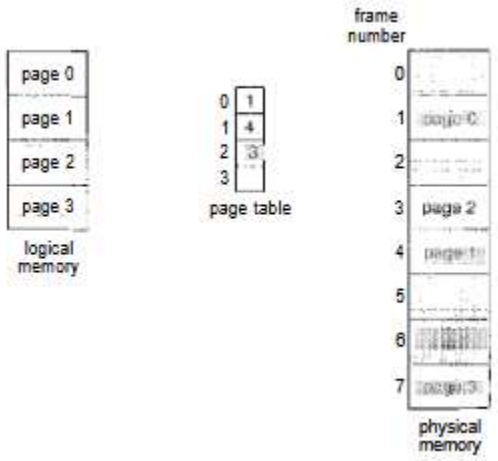


Figure 12.2 Paging model of logical and physical memory.

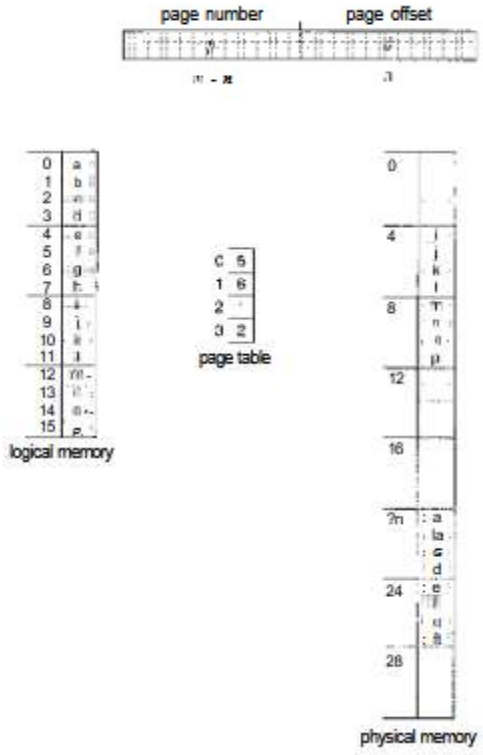


Figure 12.3 Paging example for a 32-byte memory with 4-byte pages.

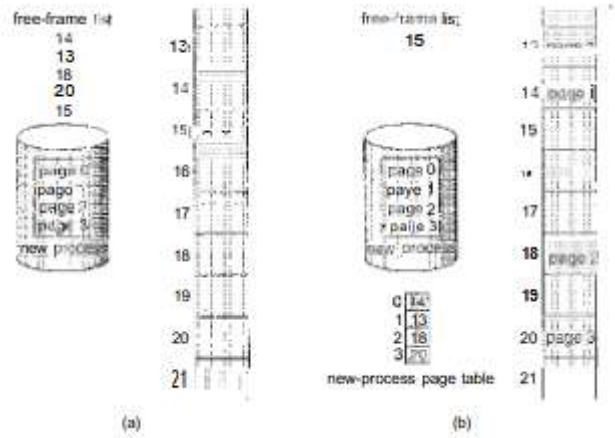


Figure 12.4 Free frames (a) before allocation and (b) after allocation.

12.2.2 Hardware Support

Each operating system has its own method of storing page tables, with most systems allocating a separate page table for each process. A pointer to this page table is stored in the **Process Control Block (PCB)** alongside other register values, such as the instruction counter. When a dispatcher starts a process, it reloads the user registers and sets the correct hardware page-table values based on the stored user page table.

Hardware Implementation of the Page Table

Page tables can be implemented in different ways. The simplest approach involves using **dedicated registers** for the page table. These registers are built with high-speed logic to ensure that memory address translation is efficient. Since all memory accesses must go through the page table, performance is critical. When a context switch occurs, the CPU dispatcher reloads the page-table registers along with other registers. Instructions to load or modify the page-table registers are privileged to ensure that only the operating system can alter the memory mapping.

For example, the **DEC PDP-11** uses a 16-bit address space with a page size of 8 KB. The page table consists of eight entries stored in fast registers, which is suitable for small page tables (e.g., 256 entries). However, modern computers support much larger page tables (e.g., 1 million entries), making the use of registers impractical. In these cases, the page table is stored in **main memory**, with the **Page Table Base Register (PTBR)** pointing to it. Changing page tables requires modifying only this register, thus reducing context-switch time.

However, the drawback of this approach is the additional memory access required to retrieve the frame number. To access a memory location, the system must first access the page table to find the corresponding frame number and then access the actual memory location. This requires two memory accesses, which doubles the time for memory access. This extra delay is typically unacceptable.

Using the Translation Lookaside Buffer (TLB)

To mitigate the performance impact of multiple memory accesses, systems use a **Translation Lookaside Buffer (TLB)**. The TLB is a small, fast memory cache that stores a subset of page-table entries. Each entry in the TLB consists of two parts:

- **Key (tag)**: the virtual page number.
- **Value**: the corresponding physical frame number.

When the CPU generates a logical address, it first checks the TLB for the page number. If the page number is present, the TLB provides the frame number immediately, allowing the system to access memory quickly—typically with less than 10% additional overhead. This is a **TLB hit**.

If the page number is not found in the TLB (a **TLB miss**), the system must access the page table in memory to get the frame number. After this, the frame number is stored in the TLB for future use. If the TLB is full, the operating system must decide which entry to replace, using replacement policies such as **Least Recently Used (LRU)** or random selection.

In some systems, the TLB entries include **Address Space Identifiers (ASIDs)**, which uniquely identify processes. ASIDs are used to ensure address-space protection by verifying that the TLB entry matches the current process's address space. If the ASIDs do not match, it results in a TLB miss. With ASID support, the TLB can hold entries for multiple processes at once, improving performance during context switches.

If the TLB does not support ASIDs, it must be flushed during each context switch to prevent the use of invalid translation entries from a previous process.

Hit Ratio and Memory Access Time

The effectiveness of the TLB is measured by its **hit ratio**, which represents the percentage of times a requested page number is found in the TLB. For example, with an **80% hit ratio**, the following memory access times can be calculated:

- **TLB hit (80%)**:
  Access time = TLB lookup (20 ns) + memory access (100 ns) = **120 ns**.
- **TLB miss (20%)**:
  Access time = TLB lookup (20 ns) + page-table lookup (100 ns) + memory access (100 ns) = **220 ns**.

The **effective access time (EAT)** is the weighted average of these two scenarios:
EAT = 0.80 x 120 + 0.20 x 220 = 140 ns.
This indicates a **40% slowdown** in memory access time compared to a direct memory access.

For a **98% hit ratio**, the calculation becomes:
EAT = 0.98 x 120 + 0.02 x 220 = 122 ns.
This results in only a **22% slowdown** in access time, demonstrating the importance of a high TLB hit ratio for improving memory access efficiency.

Memory Protection in a Paged System

Memory protection in a paged environment is implemented using protection bits associated with each frame. These protection bits are usually stored in the page table. A bit can designate a page as **read-write** or **read-only**, and every memory reference checks these bits to ensure that no illegal access occurs. If a process attempts to write to a read-only page, it causes a **hardware trap**, alerting the operating system.

For finer protection, additional bits can be added to allow **execute-only**, **read-write**, or **read-only** access, or any combination of these. Invalid access attempts trigger a trap to the operating system.

In addition to the protection bits, each entry in the page table often includes a **valid-invalid bit**. If the bit is set to **valid**, the page is in the process's address space. If set to **invalid**, the page is not part of the process's logical address space, and an attempt to access it causes a trap.

For instance, in a system with a 14-bit address space (addresses 0 to 16383) and a 2 KB page size, a program may only use addresses 0 to 10468. Pages 0 to 5 would be marked as valid, while any access beyond that (to pages 6 and 7) would trigger a page fault due to the **invalid** status of those pages.

This system can cause **internal fragmentation** due to the page size. For example, page 5 may extend beyond the program's valid address range, leading to wasted memory in the page. In such cases, a **Page Table Length Register (PTLR)** can be used to indicate the size of the page table and ensure that addresses outside the valid range trigger an error trap to the operating system.

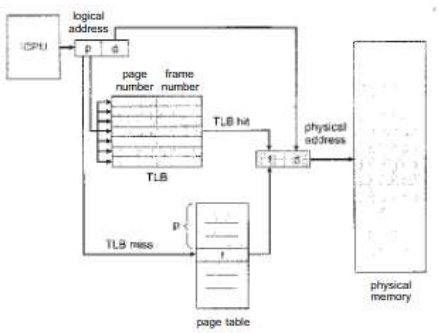


Figure 12.5 Paging hardware with TLB

12.2.3 Protection

The **protection concept** in paging refers to the ability of the operating system to prevent unauthorized access or modification of memory regions by processes. This is important for ensuring that one process does not interfere with the memory of another, leading to potential corruption or security vulnerabilities. In a system that uses paging for memory management, protection mechanisms can be implemented at the granularity of a page, which provides flexibility and fine-grained control over memory access.

Protection Mechanisms in Paging

1. **Read, Write, and Execute Permissions**:
    - Each page in memory can have protection flags associated with it, specifying whether the page is readable, writable, or executable. These flags are stored in the page table entries (PTEs) and determine how the CPU handles memory accesses to each page.
    - **Read-only Pages**: A page marked as read-only can be accessed for reading, but any write attempt to that page will cause a fault (typically a segmentation fault or page fault).
    - **Write-only Pages**: Pages that are marked as write-only can be written to, but cannot be read from.
    - **Executable Pages**: Pages marked as executable can contain code that is executed by the CPU. Non-executable pages cannot contain executable code and prevent processes from executing data.
2. **Page Table Entries (PTEs)**:
    - The page table maintains information about the pages in a process's virtual address space. Each entry in the page table not only contains the physical address of the page but also contains protection information like read/write/execute permissions. For example, a page table entry might contain flags such as:
        - R for readable
        - W for writable
        - X for executable
    - If a process tries to access a page in an illegal manner, such as attempting to write to a read-only page, the CPU triggers a **page fault**.
3. **Access Control**:
    - **User vs. Supervisor (Kernel) Mode**: In many operating systems, memory protection is used to separate user space from kernel space. Pages containing kernel code and data are protected to prevent user-mode processes from accessing or modifying critical operating system data. The CPU enforces this by checking the current privilege level of the process attempting to access a page.

   - For example, when a process in user mode tries to access a page marked as kernel-only, a protection fault occurs.
4. **Page Fault Handling**:
   - A **page fault** occurs when a process tries to access a page in an illegal way, such as trying to write to a read-only page, or attempting to access a page that does not exist or is not currently mapped.
   - The operating system handles the page fault by determining whether the access attempt is a legitimate error or due to a specific requirement, such as loading a page into memory from disk.

Example of Protection in Paging

Let's consider a scenario in which an operating system uses paging with different protection levels for the pages.

Example: Virtual Memory for a Text Editor

Suppose we have a text editor process running with the following memory layout:

- **Code Segment**: Contains the program code.
- **Data Segment**: Contains global variables and heap space.
- **Stack Segment**: Contains the stack for function calls and local variables.

Here is how the protection can be applied:

1. **Code Segment (Read-Only and Executable)**:
   - The code pages are marked as **read-only** and **executable**. The operating system ensures that the program can execute instructions from these pages but cannot modify them. If the program tries to write to its code segment (for example, a bug tries to modify an instruction), the operating system raises a page fault.
   - **Page Table Entry**: R (readable), X (executable), W (not writable).
2. **Data Segment (Read-Write)**:
   - The data pages are marked as **read-write**, allowing the program to modify variables stored in this segment. These pages can be read from and written to. If a process tries to execute data (for instance, if it tries to run a data section as code), the operating system will generate a protection fault.
   - **Page Table Entry**: R (readable), W (writable), X (not executable).
3. **Stack Segment (Read-Write)**:
   - The stack pages are also marked as **read-write** to allow pushing and popping data (such as function return addresses and local variables) to and from the stack. The operating system typically prevents the stack from growing into the data or code segments.

   - **Page Table Entry**: R (readable), W (writable), X (not executable).
4. **Kernel Space (Protected Pages)**:
   - The operating system ensures that the kernel space is protected by marking the kernel pages as **read-write** and **non-executable** from user space. User-mode processes cannot access the kernel's code or data directly, which helps prevent unauthorized access or modifications to the operating system's critical resources.
   - **Page Table Entry**: R (readable), W (writable), X (not executable) for kernel pages, but these pages cannot be accessed from user mode.

Scenario: Protection Fault in Paging

Let's say the text editor has a bug, and it tries to write to the code segment. Here's what happens:

1. **Illegal Access**: The program attempts to modify an instruction in the code segment, which is a read-only page.
2. **Page Table Check**: The CPU checks the page table entry for the page containing the code. It finds that the page is marked as read-only (W not set).
3. **Page Fault**: Since the process is trying to write to a read-only page, a protection fault occurs, causing a page fault exception.
4. **Handling the Fault**: The operating system handles the fault by terminating the program, displaying an error, or potentially invoking a debugger if the error is expected.

Advantages of Protection in Paging

1. **Isolation**: Each process is isolated from other processes, preventing one process from reading or modifying the memory of another process.
2. **Security**: Memory protection prevents malicious processes or bugs in one process from accessing or corrupting sensitive memory regions, like the operating system's kernel memory.
3. **Error Prevention**: Protection mechanisms help prevent errors like writing to read-only memory or executing data as code, which could result in crashes or unpredictable behavior.

Protection in paging is a vital concept that ensures the integrity, security, and isolation of processes in a system. By leveraging read, write, and execute permissions, operating systems can enforce strict control over memory access, preventing processes from corrupting each other's data and avoiding security breaches. The protection mechanisms are managed through the page table, with each entry containing flags that determine the type of access allowed to each page.

12.2.4 Shared Pages

Shared pages refer to memory pages that can be accessed by multiple processes at the same time. This is an essential concept in modern operating systems to optimize memory usage, as it allows processes to share common data or code without duplicating it in each process's private memory space. Shared pages are used in a variety of contexts, such as shared libraries, inter-process communication (IPC), and memory-mapped files.

Why Use Shared Pages

1. **Memory Efficiency**: When multiple processes require access to the same data or code (e.g., a shared library), instead of loading the same code or data into each process's memory space, the operating system can load it once and allow all processes to access it via a shared page. This reduces the total memory usage.
2. **Performance**: Accessing shared data in memory is faster than inter-process communication mechanisms like pipes or message queues because it avoids the overhead of context switching and copying data between processes.
3. **Consistency**: In some cases, multiple processes need to access the same data and reflect any changes made to that data immediately across all processes. Shared memory provides a mechanism for this direct interaction.

How Shared Pages Work

In an operating system that supports shared pages, the system assigns a region of memory to be shared across different processes. The virtual addresses for the shared pages are mapped into the address spaces of the processes that need to access the shared data.

Each process gets its own private memory for its private data, but the shared pages are mapped into each process's virtual memory, with all processes accessing the same physical memory page. If one process modifies data in a shared page, the change is immediately visible to all other processes that are accessing that page.

Example of Shared Pages

Consider a scenario where multiple processes are running on an operating system that uses shared memory for a common library. For example, the libc (C Standard Library) is often shared by many running programs, so each process doesn't need its own copy of the library in memory.

Process 1 (e.g., a text editor) and Process 2 (e.g., a web browser) need access to libc.

Both processes require the same library to execute functions like printf, malloc, etc. Here's how shared pages help:

1. **Step 1: Library Mapping**: The operating system loads the libc library into physical memory. When a program (like Process 1 or Process 2) starts, it does not load its own copy of libc. Instead, the OS maps the same physical memory page containing the libc code into the virtual address space of each process.
2. **Step 2: Shared Memory Access**: The virtual addresses in each process's address space point to the same physical page where the libc code is stored. Both processes can now execute the functions in libc without each having to load their own copy.
3. **Step 3: No Duplication**: Since the same physical page is shared, memory is used efficiently. Even though multiple processes use the libc, it exists only once in physical memory.

Shared Memory Example: IPC between Two Processes

Another common use of shared pages is for inter-process communication (IPC). In many systems, shared memory is used to allow processes to exchange information directly, without the need for slower IPC mechanisms.

For example:

1. **Step 1: Shared Memory Creation**: Process A and Process B create a shared memory segment (a shared page) using system calls like shmget (in UNIX-like systems).
2. **Step 2: Memory Mapping**: Both processes map this shared memory segment into their address spaces using calls like shmat.
3. **Step 3: Data Exchange**: Process A writes data to the shared memory, and Process B reads from it. Both processes access the same physical page, so no data copying is needed.

Mechanism for Managing Shared Pages

1. **Page Table Entries**: In the page tables of each process, the entries corresponding to the shared pages will point to the same physical memory location. The operating system maintains the mapping to ensure that multiple processes share the same physical memory page.
2. **Copy-on-Write (COW)**: In some cases, shared pages are marked as copy-on-write. This means that if a process tries to modify a shared page, the operating system will create a copy of that page for the process, ensuring that modifications do not affect other processes. COW is commonly used in scenarios like fork operations.

Example of Copy-on-Write with Shared Pages

1. **Step 1: Fork**: When a process forks, the child process inherits a copy of the parent's page table. The shared pages are marked as copy-on-write.
2. **Step 2: Access**: The child process and the parent process both use the shared page. If either process writes to the shared page, the operating system makes a copy of the page for the writing process, allowing the child and parent to continue working without interfering with each other.
3. **Step 3: Modification**: If the parent process modifies its version of the shared page, it gets its own copy, and the child process keeps using the original, unmodified page. This way, shared data is protected from unintended modifications by separate processes.

Advantages of Shared Pages

1. **Memory Efficiency**: By allowing multiple processes to share the same physical memory, shared pages reduce the overall memory footprint.
2. **Faster IPC**: Shared memory provides a fast means of communication between processes, as it avoids the overhead of copying data between user spaces.
3. **Simplicity**: Implementing shared memory for communication can simplify the code compared to message-passing or other forms of IPC.

Disadvantages of Shared Pages

1. **Synchronization Issues**: If multiple processes modify shared memory simultaneously, it can lead to inconsistent data. Proper synchronization (e.g., semaphores or mutexes) is necessary to ensure that only one process modifies the shared memory at a time.
2. **Security Risks**: If not properly managed, shared memory can introduce security risks. Malicious processes might alter shared pages in ways that compromise the integrity of other processes.
3. **Limited Flexibility**: The shared pages need to be explicitly mapped into each process's address space, which can be cumbersome if the shared data needs to be accessed by many processes.

Shared pages are a powerful tool for efficiently managing memory in modern operating systems. By allowing multiple processes to access the same physical memory page, shared memory can save memory, increase performance, and simplify inter-process communication. However, managing shared pages requires careful handling of synchronization, access control, and security to avoid issues like race conditions and data corruption.

12.3 Structure of the Page Table

In this section, we explore some of the most common techniques used for structuring the page table. These methods aim to efficiently map virtual addresses to physical addresses while minimizing memory usage and search times. We will discuss various strategies, including hierarchical page tables, inverted page tables, and hashed page tables, as well as the trade-offs associated with each approach. These techniques are crucial in managing memory in modern systems, especially when dealing with large address spaces or complex memory management requirements. Each method provides a different way to organize and access the page table entries, impacting system performance and resource utilization.

12.3.1 Hierarchical Paging

Hierarchical paging is a technique used in modern operating systems to handle large virtual address spaces efficiently. Instead of using a single-level page table to map virtual addresses to physical addresses, hierarchical paging breaks the page table into multiple levels. This structure allows the operating system to use memory more efficiently, especially in systems with large address spaces.

Overview of Hierarchical Paging

In hierarchical paging, the virtual address is divided into several parts, each corresponding to a different level of page tables. These levels are typically organized as follows:

1. **Page Directory**: This is the first level of the page table, containing entries that point to second-level page tables.
2. **Page Table**: This is the second level of the page table, which contains entries pointing to the actual physical pages.
3. **Page Frame**: The final level contains the actual data, which resides in physical memory.

How Hierarchical Paging Works

To understand hierarchical paging, let's consider an example using a two-level paging system. A virtual address in such a system is divided into three parts:

- **Page Directory Index**: The highest-order bits of the virtual address point to an entry in the page directory.
- **Page Table Index**: The next set of bits point to an entry in a page table.
- **Page Offset**: The remaining bits give the offset within the page.

Example: Two-Level Paging

Consider a system with a 32-bit virtual address space and a page size of 4 KB (i.e., 2^12 bytes). Here's how a two-level paging system works:

- **Virtual Address Breakdown**:
  - The 32-bit address is split into three parts:
    - The first 10 bits (bits 31-22) represent the **Page Directory Index** (PDI).
    - The next 10 bits (bits 21-12) represent the **Page Table Index** (PTI).
    - The last 12 bits (bits 11-0) represent the **Page Offset**.

Given these 32 bits, the system can address a 4 GB virtual memory space (2^32 bytes), with each page being 4 KB in size.

Steps of Translation in Two-Level Paging

Let's break down the address translation process step by step:

1. **Virtual Address**: A process generates a 32-bit virtual address. For example, consider the address 0x12345678 (in hexadecimal).
2. **Page Directory Lookup**: The first 10 bits (0x12345 → bits 31-22) of the address point to an entry in the **Page Directory**. This entry holds the base address of a second-level **Page Table**.
3. **Page Table Lookup**: The next 10 bits (0x678 → bits 21-12) point to an entry in the **Page Table**. This entry holds the physical address of the **Page Frame**.
4. **Page Offset**: The final 12 bits (0x78 → bits 11-0) specify the exact location of the byte within the page.

The process would look something like this:

- The address 0x12345678 is split into:
  - Page Directory Index (PDI) = 0x12345 (bits 31-22)
  - Page Table Index (PTI) = 0x678 (bits 21-12)
  - Page Offset = 0x78 (bits 11-0)

Using the **PDI**, the operating system locates the page directory entry, which gives the base address of the **Page Table**. Using the **PTI**, it locates the page table entry, which gives the physical page frame. Finally, the **Page Offset** gives the exact byte within that page.

Diagram of Hierarchical Paging

Here's how this would look graphically:

Virtual Address (32 bits) --> Page Directory Index (10 bits) --> Page Table Index (10 bits) --> Page Offset (12 bits)

```
[ Page Directory Entry (points to Page Table) ]
 |
 V
[ Page Table Entry (points to physical page) ]
 |
 V
Physical Page Frame + Page Offset = Physical Address
```

Advantages of Hierarchical Paging

1. **Memory Efficiency**: By breaking the page table into multiple levels, the system only allocates memory for the page tables and page directories that are actually needed. This reduces memory overhead, especially in sparse address spaces where not all virtual addresses are used.
2. **Scalability**: Hierarchical paging makes it possible to handle large address spaces. For example, in a 64-bit system, hierarchical paging allows the operating system to manage extremely large address spaces without having to allocate a massive single-level page table.
3. **Reduced Fragmentation**: Hierarchical paging can reduce internal fragmentation within the page table by allocating space only when needed.

Disadvantages of Hierarchical Paging

1. **Address Translation Overhead**: With multiple levels of page tables, address translation can take multiple memory accesses. For each virtual address, the operating system must access the page directory, then the page table, and finally the actual data. This can lead to higher latency in address translation.
2. **Complexity**: Managing multiple levels of page tables adds complexity to the memory management system, and requires efficient algorithms for searching and updating page tables.

Hierarchical paging is a powerful technique that allows modern operating systems to manage large virtual address spaces efficiently. By breaking down a single, large page table into multiple levels, hierarchical paging reduces memory overhead and increases scalability. However, it introduces additional complexity and overhead in address translation, which must be carefully managed by the operating system. This technique

is particularly useful in systems that need to handle large or sparse virtual address spaces, such as 64-bit systems.

12.3.2 Hashed Page Tables

A common method for handling address spaces larger than 32 bits is the use of a hashed page table, where the virtual page number is hashed to determine its location in the table. Each entry in the hash table contains a linked list to manage collisions. The list's elements include three fields: (1) the virtual page number, (2) the mapped page frame, and (3) a pointer to the next element in the list.

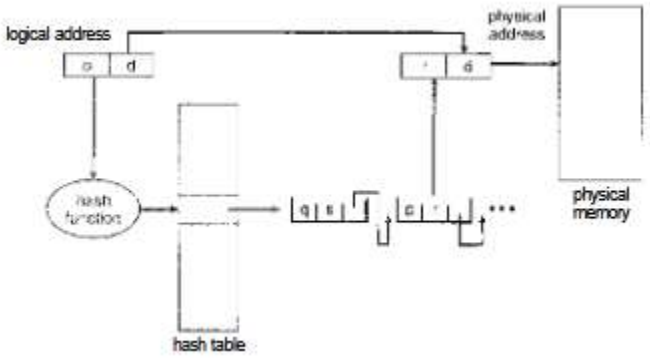


Figure 12.7 Hashed page table.

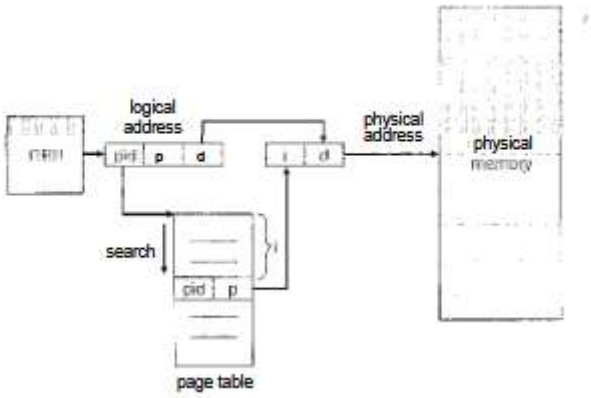


Figure 12.8 Inverted page table

The process works as follows: The virtual page number from the virtual address is hashed into the hash table. The number is then compared with the first element's virtual page number in the linked list. If a match is found, the corresponding page frame (stored in field 2) is used to generate the physical address. If there is no match, the search continues through the linked list to find an entry with a matching virtual page number. This is illustrated in Figure 8.16.

A variation of this approach, particularly suited for 64-bit address spaces, is the clustered page table. Instead of mapping a single page, each entry in a clustered page table refers to multiple pages (e.g., 16). This allows each entry to store the mappings for several physical page frames, making clustered page tables efficient for sparse address spaces, where memory references are scattered and non-contiguous.

12.3.3 Inverted Page Tables

Each process typically has its own page table, which contains one entry for each page the process is using or for each virtual address, regardless of its validity. This structure allows the operating system to translate virtual addresses into physical memory addresses efficiently. Since the page table is sorted by virtual address, the OS can quickly locate the corresponding entry for a given virtual address.

However, the major drawback is that these tables can be quite large, with millions of entries, consuming substantial physical memory.

To address this issue, an inverted page table can be used. An inverted page table has one entry for each physical page (or memory frame) rather than for each virtual page. Each entry includes the virtual address of the page stored in that physical memory location, along with information about the process that owns the page. This system uses only one page table for the entire system, reducing memory usage. Inverted page tables are often used in systems like the 64-bit UltraSPARC and PowerPC, and they typically include an address-space identifier to distinguish between different processes mapping to the same physical memory.

In the IBM RT system, for example, each virtual address is represented as a triple <process-id, page-number, offset>. The inverted page table entry consists of a pair <process-id, page-number>, and when a memory reference occurs, the virtual address is searched for a match in the table. If a match is found, the corresponding physical address is generated; otherwise, an illegal access error is triggered.

While inverted page tables reduce memory usage, they can slow down address translation since the table must be searched for every memory reference. To speed up the lookup, a hash table is often used to minimize the number of entries that need to be searched. However, this requires additional memory accesses, as each lookup involves a hash table search followed by a page table search. To improve performance further, the Translation Lookaside Buffer (TLB) is checked first before accessing the hash table.

One challenge with inverted page tables is supporting shared memory, as they only map one virtual address to each physical page. This makes it difficult for multiple processes to share the same physical page. To work around this, the page table can be designed to only store one mapping per shared physical page, with page faults occurring for any unmapped virtual addresses.

12.4 SEGMENTATION

A key aspect of memory management that became necessary with paging is the separation between the user's view of memory and the actual physical memory. As previously discussed, the user's perspective of memory differs from the physical memory itself. This view is mapped onto physical memory, allowing for a clear distinction between **logical memory** (as seen by the user) and **physical memory** (the actual hardware storage).

12.4.1 Basic Method

Most users do not view memory as a linear array of bytes containing instructions and data. Instead, they perceive it as a **collection of variable-sized segments** without any particular ordering. For instance, when writing a program, users think of it as a set of methods, procedures, or functions along with data structures like arrays, stacks, and variables. They refer to these elements by name (e.g., "the stack" or "the math library") without concern for their actual memory addresses or relative positions.

Segmentation is a memory-management scheme that aligns with this perspective. It divides a program's logical address space into **segments**, each identified by a **name** and having a **variable length**. Memory elements within a segment are referenced by their **offset** from the segment's starting point. For example, users might refer to specific instructions, stack frames, or function calls by their relative positions within a segment.

Logical addresses in segmentation are represented as a two-tuple:
<segment-number, offset>.

For simplicity, segments are numbered instead of named, and compilers automatically create these segments based on the program's structure. For example, a C compiler might generate segments for:

Code
Global variables
The heap (for dynamic memory allocation)
Stacks (for each thread)
Standard C library

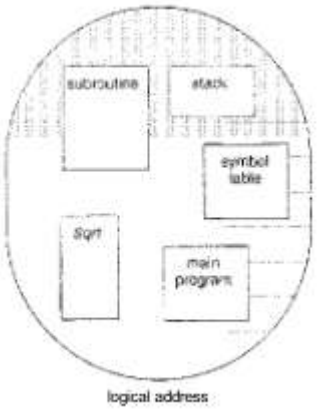


Figure 12.9 User's view of a program

Libraries linked during compile time may also be assigned separate segments. The loader assigns **segment numbers** to all segments, enabling efficient memory management and alignment with the user's conceptual view of memory.

12.4.2 Hardware

In computer systems, although a program can use two-dimensional addresses to reference objects, the actual physical memory is a one-dimensional sequence of bytes. To bridge this difference, a **segment table** is used to map two-dimensional user-defined addresses into one-dimensional physical addresses. Each entry in the segment table includes:

- **Segment base**: The starting physical address of the segment in memory.
- **Segment limit**: The length of the segment.

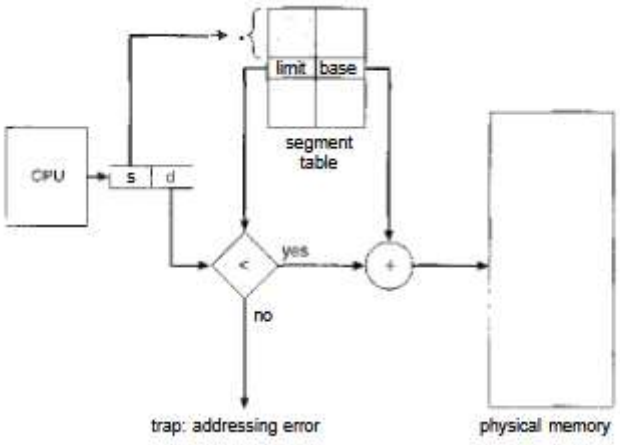


Figure 12.10 Segmentation hardware

A **logical address** consists of two parts: a **segment number (s)** and an **offset (d)** within the segment. The segment number serves as an index into the segment

table, and the offset must fall between 0 and the segment limit. If the offset exceeds the segment limit, a trap to the operating system occurs. For valid offsets, the segment base is added to the offset to compute the physical memory address.

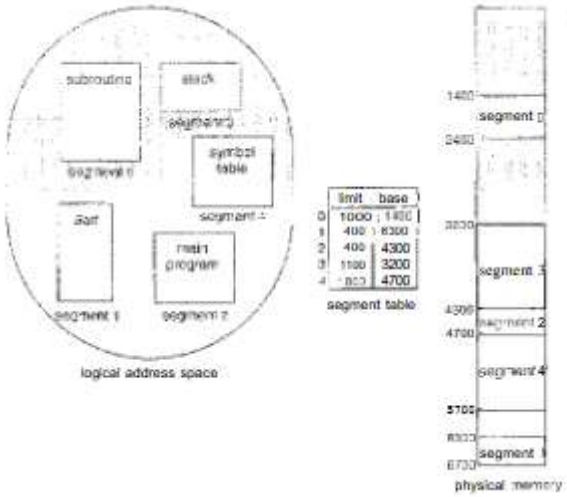


Figure 12.11 Example of segmentation.

For example, consider five segments stored in physical memory. If segment 2 starts at **4300** and has a length of **400 bytes**, referencing byte 53 of segment 2 maps to **4300 + 53 = 4353**. A reference to byte 852 of segment 3 (base **3200**) maps to **3200 + 852 = 4052**. However, a reference to byte 1222 of segment 0 (limit **1000 bytes**) triggers a trap, as it exceeds the segment's size.

12.5 EXAMPLE: THE INTEL PENTIUM

Paging and segmentation each have their own advantages and disadvantages, and some architectures, like the Intel Pentium, support both. The Pentium architecture accommodates **pure segmentation** as well as **segmentation with paging**, providing flexible memory management capabilities. Rather than detailing the entire memory-management structure, this overview focuses on its key concepts, concluding with Linux address translation on Pentium systems.

In the Pentium, the CPU generates **logical addresses**, which are first processed by the **segmentation unit**. This unit converts the logical address into a **linear address**. The linear address is then passed to the **paging unit**, which translates it into a **physical address** in main memory. Together, the segmentation and paging units operate as a **Memory Management Unit (MMU)**, efficiently managing address translation.

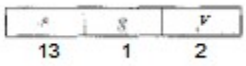
12.5.1 Pentium Segmentation

The Pentium architecture supports segments up to **4 GB** and allows a process to use up to **16 KB segments**, divided into two partitions: private segments (managed by the **Local Descriptor Table, LDT**) and shared segments (managed by the **Global**

Descriptor Table, GDT). Segment descriptors in LDT and GDT are **8 bytes** long, containing base location and limit details.

Figure 12.12 Logical to physical address translation in the Pentium

Logical addresses are represented as a **(selector, offset)** pair. The **16-bit selector** specifies the segment, whether it is in the LDT or GDT, and its protection level. The **32-bit offset** locates the byte or word within the segment.



The Pentium has **six segment registers** and associated **microprogram registers** that cache descriptors, avoiding frequent memory reads. To generate a linear address, the segment register fetches the segment's base and limit. If valid (checked against the limit), the offset is added to the base, forming a **32-bit linear address**. Invalid addresses trigger a memory fault handled by the OS. This linear address is further translated into a physical address via paging.

12.5.2 Pentium Paging

Pentium processors use a two-level paging model to translate linear addresses into physical addresses for memory management. This model consists of a **page directory** and **page tables**, which divide the address space into manageable units.

1. The linear address is divided into three parts: a directory index, a table index, and a page offset.
2. The **page directory** contains entries pointing to page tables.
3. Each **page table** holds entries pointing to physical memory pages.
4. Page sizes are typically 4 KB, but large pages (4 MB) are also supported.
5. The Control Register (CR3) holds the physical address of the current page directory.
6. During context switches, CR3 is updated to point to the new task's directory.
7. Paging provides support for memory protection and isolation between processes.
8. Access permissions and status bits in page table entries control read/write access.
9. The Translation Lookaside Buffer (TLB) caches recent address translations for efficiency.
10. Pentium's paging supports both 32-bit and 64-bit systems when combined with extensions.

12.5.3 Linux on Pentium Systems

The Linux operating system is designed to operate on diverse hardware platforms, including Intel's Pentium. To accommodate limited segmentation support on some processors, Linux minimizes segmentation usage. It employs six segments: kernel code, kernel data, user code, user data, a task-state segment (TSS), and a default LDT segment. User code and data segments are shared across all processes, leveraging a common logical address space.

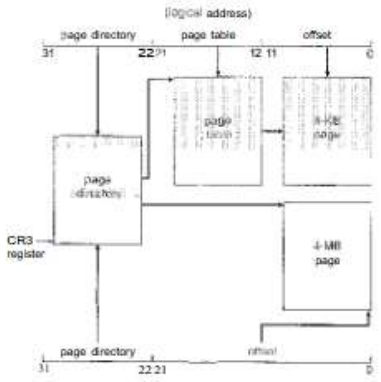


Figure 12.13 Paging in the Pentium architecture

Each process has its own TSS, storing hardware context during task switches. Segment descriptors reside in the global descriptor table (GDT). Though Linux supports the Pentium's two-level paging, it implements a three-level paging model to ensure compatibility with 32-bit and 64-bit platforms. The middle directory in this model is omitted on Pentium systems.

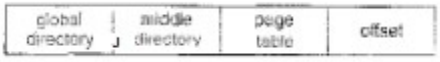


Figure 12.14 highlights the three-level paging model in Linux

Protection levels are simplified to kernel mode and user mode, despite the Pentium's four-level capability. Each task in Linux has distinct page tables, with the CR3 register pointing to the global directory. During context switches, the CR3 value is saved and restored via TSS for efficient memory management.

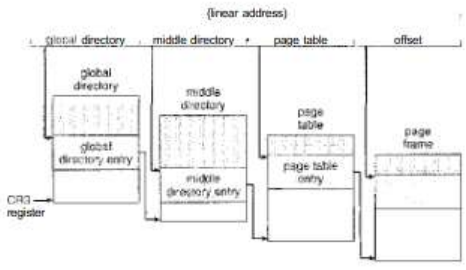


Figure 12.15 Three-level paging in Linux

12.6 SUMMARY

The main memory management strategies, including contiguous allocation, paging, and segmentation, each have their own advantages and challenges. Performance can be impacted by the complexity of the memory-management algorithm, and efficient mapping of logical addresses to physical ones is crucial. Fragmentation, either internal or external, is a significant concern, and techniques like compaction are used to alleviate external fragmentation. Swapping, sharing, and protection mechanisms help optimize memory usage, support multiprogramming, and provide security and error-checking during execution.

12.7 KEY TERMS

Paging, Hit Ratio and Memory Access Time, Memory, Protection ,Paged System, Protection, Code Segment**,** Stack Segment Data Segment**,** Segmentation, Pentium Paging

12.8 SELF-ASSESSMENT QUESTIONS

1. What is the purpose of paging in memory management?
2. How does the basic method of paging work?
3. What is the role of hardware support in implementing paging?
4. What are shared pages, and how are they useful?
5. What is the difference between hierarchical paging and inverted page tables?

12.9 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”,

Seventh Edition, Wiley.

2.William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3.Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4.Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5.Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Mrs. Appikatla Pushpa Latha

CHAPTER 13

VIRTUAL MEMORY MANAGEMENT

Aim and Objectives:

- To describe the advantages of virtual memory systems in enhancing memory management and system performance.
- To explain the concepts of demand paging, focusing on efficient memory usage by loading pages only when needed.
- To analyze page-replacement algorithms and allocation of page frames for optimal memory utilization.
- To discuss the working-set model's principles and its role in managing active memory pages effectively.

Structure:

13.1 INTRODUCTION

The concept of virtual memory arises from the need to execute programs in physical memory, which traditionally limits a program's size to the available physical memory. However, not all parts of a program are needed simultaneously, and many programs include unused code or features. For instance, error handling code or large data structures may go unused during execution. Virtual memory allows a program to execute even if it is not fully loaded into physical memory, enabling programs to be larger than the available memory and allowing more programs to run simultaneously. This enhances system performance by reducing I/O operations, improving CPU utilization, and increasing throughput without increasing response time.

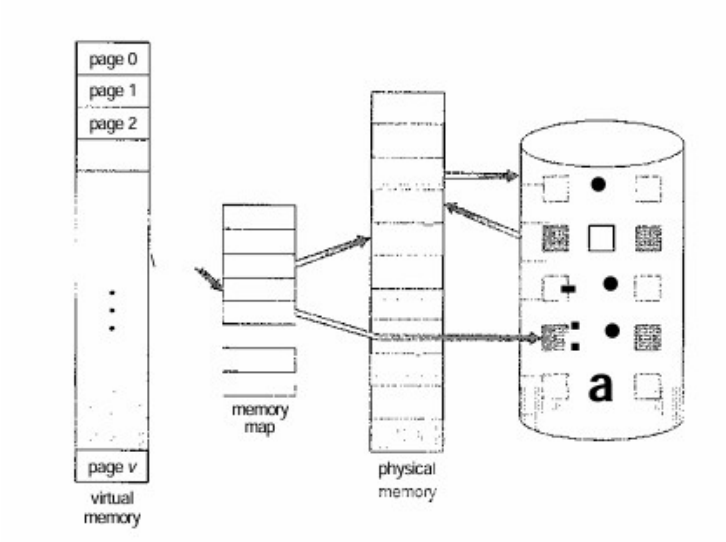


Figure 13.1 Diagram showing virtual memory that is larger than physical memory.

Virtual memory separates logical memory, as seen by the user, from physical memory. This allows programmers to work with an effectively larger virtual address space, while physical memory remains limited. The operating system manages the mapping of logical pages to physical frames. The system also uses techniques like sparse address spaces, where gaps between the stack and heap can be dynamically filled, which optimizes memory usage. Additionally, virtual memory facilitates sharing of system libraries and memory between processes. This can be particularly beneficial for communication and for reducing redundancy in memory usage.

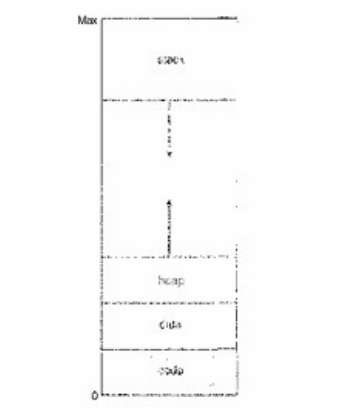


Figure 13.2 Virtual address space

13.1 Background

Virtual memory's flexibility in memory allocation and process communication, through techniques like page sharing and memory mapping, provides significant efficiency gains. By implementing virtual memory with demand paging, the system can load only the required parts of a program into memory, reducing both the memory footprint and the time spent loading programs. The following sections will explore the implementation of virtual memory through demand paging and its associated benefits.

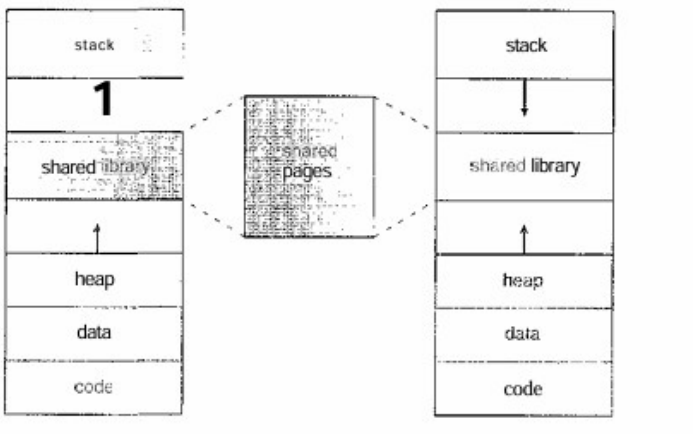


Figure 13.3 Shared library using virtual memory.

13.2 DEMAND PAGING

When loading an executable program from disk into memory, one approach is to load the entire program into physical memory at execution. However, this method is inefficient as the entire program may not be required at once. For example, a program displaying a menu of options loads all executable code, even for unused options. A more efficient strategy is **demand paging**, where only the necessary pages are loaded into memory during program execution. Unaccessed pages remain on disk, saving memory space and improving performance.

Demand paging operates similarly to a paging system with swapping, where processes reside in secondary storage and are brought into memory when needed. Unlike traditional swapping that loads entire processes, demand paging employs a **lazy swapper** (more accurately termed a **pager**) that loads individual pages into memory only as required. This distinction reflects the focus of demand paging on managing pages rather than whole processes, optimizing memory utilization by loading and keeping only the essential parts of a program in memory. This technique is a cornerstone of virtual memory systems, balancing memory usage and computational efficiency.

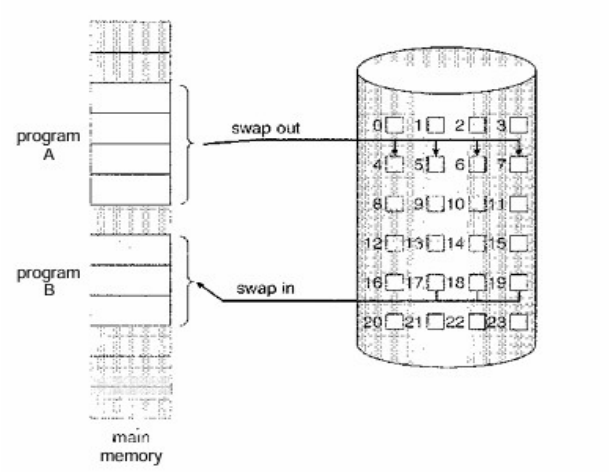


Figure 9.4 Transfer of a paged memory to contiguous disk space

13.2.1 Basic Concepts

When a process is swapped into memory, the **pager** predicts which pages will be needed and loads only those, avoiding unnecessary memory usage and reducing swap time. To manage this efficiently, hardware support like a **valid-invalid bit scheme** distinguishes between pages in memory and those on disk. A page marked "invalid" might either not belong to the process or be valid but not yet in memory.

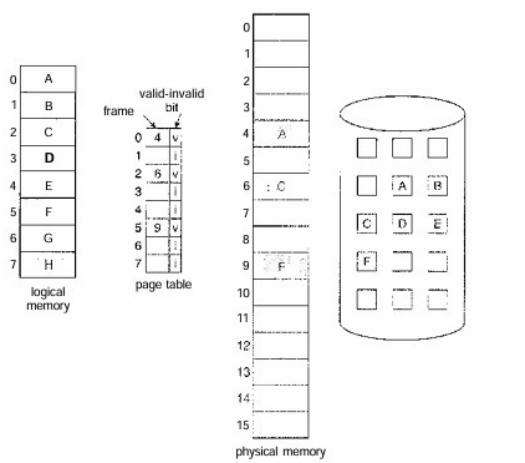


Figure 9.5 Page table when some pages are not in main memory

If a process accesses an "invalid" page, a **page-fault trap** occurs. The operating system checks if the access is valid. If valid, it allocates a free frame, reads the page from disk, updates the page table, and resumes the interrupted instruction. This process, known as **pure demand paging**, starts execution with no pages in memory, bringing in pages only as required.

Hardware for Demand Paging:

1. **Page Table:** Marks entries as valid or invalid using special bits.
2. **Secondary Memory:** Stores non-resident pages on high-speed disks in swap space.

Instruction Handling and Restarts:

After a page fault, the process must restart in the same state, with the desired page now in memory. This is manageable for most instructions, but complex cases, like overlapping memory operations, may require additional measures such as precomputing faults or using temporary registers to preserve data integrity.

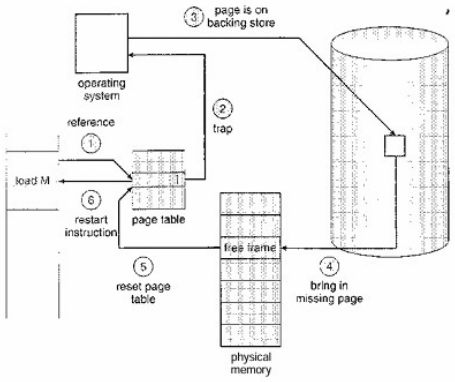


Figure 13.6 Steps in handling a page fault

Challenges with Paging:

Adding paging to existing architectures poses difficulties, particularly in ensuring that it remains transparent to the user. While paging works seamlessly in systems where page faults are fatal errors, handling demand paging—where faults require loading additional pages and restarting processes—adds complexity.

13.2.2 Performance of Demand Paging

The performance of demand paging depends on the **page-fault rate**, which ranges from 0 (no faults) to 1 (every access causes a fault). A low page-fault rate is essential for maintaining system performance since page faults involve significant overhead due to disk I/O and memory access.

Effective Access Time (EAT)

The effective access time is calculated by combining the time for a memory access and the overhead of handling page faults:

EAT=(1−p)×memory access time×(page fault service time)

Where:

- p: Page-fault rate (probability of a page fault).
- **Memory access time**: The time to access memory (e.g., 200 ns).
- **Page fault service time**: Includes time to retrieve the page from disk and update page tables (e.g., 10 ms for disk access).

Example Calculation

Suppose:

- Memory access time = **200 ns**
- Page fault service time = **10 ms** = **10,000,000 ns**
- Page-fault rate = **0.001** (1 fault per 1,000 accesses)

EAT=(1−0.001)×200+0.001×10,000,000

EAT=0.999×200+0.001×10,000,000=200+10,000=10,200ns

If the page-fault rate increases (e.g., p=0.01), the EAT becomes:

EAT=0.99×200+0.01×10,000,000=200+100,000=100,200ns

This demonstrates that even a slight increase in the page-fault rate drastically impacts performance.

Optimizing Demand Paging Performance

- **Reducing Page Faults:** Techniques like **working set models** and **locality of reference** help minimize faults by keeping frequently accessed pages in memory.
- **Efficient Replacement Policies:** Algorithms like **Least Recently Used (LRU)** or **Clock** aim to replace the least useful pages.
- **Prefetching:** Predicts and loads required pages in advance.
- **Adequate Physical Memory:** Ensures sufficient memory to accommodate active pages.

By optimizing these factors, demand paging can balance the benefits of virtual memory with manageable overhead.

13.3 COPY-ON-WRITE

Efficient Process Creation: Copy-on-Write and vfork()

In Section 9.2, we saw how demand paging helps a process start quickly by loading only the page containing the first instruction. However, the process creation using the fork() system call can be optimized even further by using a technique known as **Copy-on-Write (COW)**. This approach minimizes the overhead of creating new pages for a child process and can significantly reduce memory usage when the parent process is duplicated.

Copy-on-Write (COW) with fork()

The traditional behavior of fork() involves creating an exact duplicate of the parent process's address space for the child process. This process typically includes copying all the pages of the parent, even though many of these pages might never be modified by the child. Since many child processes call exec() shortly after they are created (which loads a new program into memory), copying the parent's address space becomes inefficient. Instead, **Copy-on-Write** (COW) allows the parent and child processes to share the same pages, which significantly improves performance and memory efficiency.

How Copy-on-Write Works:

- When a child process is created using fork(), the parent and child processes initially share the same memory pages.
- These shared pages are marked as **copy-on-write**. This means that the operating system ensures that if either process tries to modify a shared page, a copy of the page is created for that process. The modification will then occur on the new copy, while the other process still uses the original page.

- Pages that are read-only, such as those containing executable code, can remain shared between the parent and child since they do not need to be modified.

Example:

Assume the parent process has a page (let's call it Page C) containing a portion of its stack. When fork() is called, the child process does not get its own copy of Page C right away. Instead, both processes initially share Page C as read-only, marked for copy-on-write. If the child process attempts to modify Page C (e.g., by writing to the stack), the operating system creates a copy of Page C specifically for the child. The parent's page remains untouched.

- **Before modification**: Parent and child processes share the same copy-on-write pages.
- **After modification**: The child modifies Page C, and a new copy of Page C is created just for the child.

This approach ensures that only the modified pages are copied, while unmodified pages remain shared. The result is reduced memory usage and faster process creation.

Zero-Fill-on-Demand:

When a copy-on-write page is allocated, it is often allocated using a technique called **zero-fill-on-demand**. This means that the page is filled with zeros before being allocated to the process. This technique ensures that no residual data from other processes is present in the newly allocated memory.

vfork() - A More Efficient Alternative

In addition to fork(), some operating systems offer a variation known as **vfork()**. The primary difference between fork() and vfork() is that vfork() suspends the parent process until the child process calls exec() (or exits). This means that the child process uses the parent's address space directly, without copying any pages, and thus is much faster in process creation.

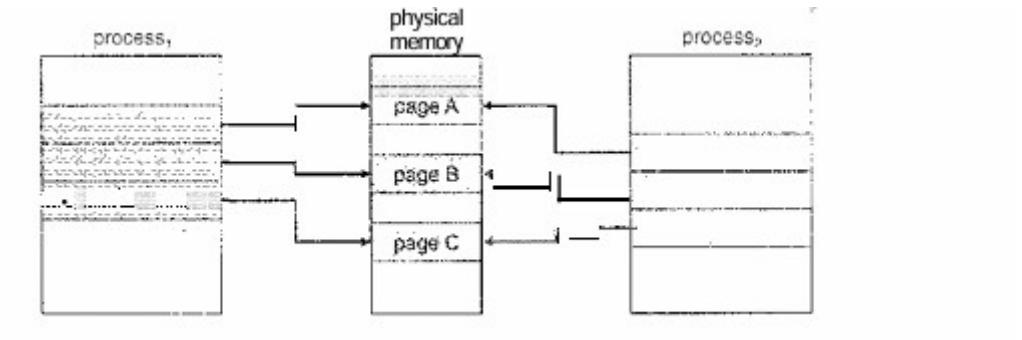


Figure 13.7 Before process 1 modifies page C.

How vfork() Works:

- The child process created by vfork() uses the **same address space** as the parent. If the child modifies a page, those changes will be visible to the parent process when it resumes execution.
- This is highly efficient for cases where the child process will immediately call exec() to load a new program. However, because vfork() does not use copy-on-write, it can be dangerous if the child modifies the parent's address space before calling exec(), as this could lead to unexpected results.

vfork() is commonly used in **Unix-like operating systems** (such as Linux and Solaris) for creating processes that are expected to call exec() immediately afterward, such as when implementing command-line interfaces in shells.

- **Copy-on-Write (COW)** optimizes process creation by allowing the child process to share memory pages with the parent, copying only when a modification occurs.
- **Zero-fill-on-demand** is often used to allocate new pages that are filled with zeroes before they are used.
- **vfork()** provides a more efficient method of process creation by having the child process share the parent's address space, making it suitable for situations where the child immediately calls exec().

Both **COW** and **vfork()** are essential techniques for improving process creation speed and memory efficiency in modern operating systems.

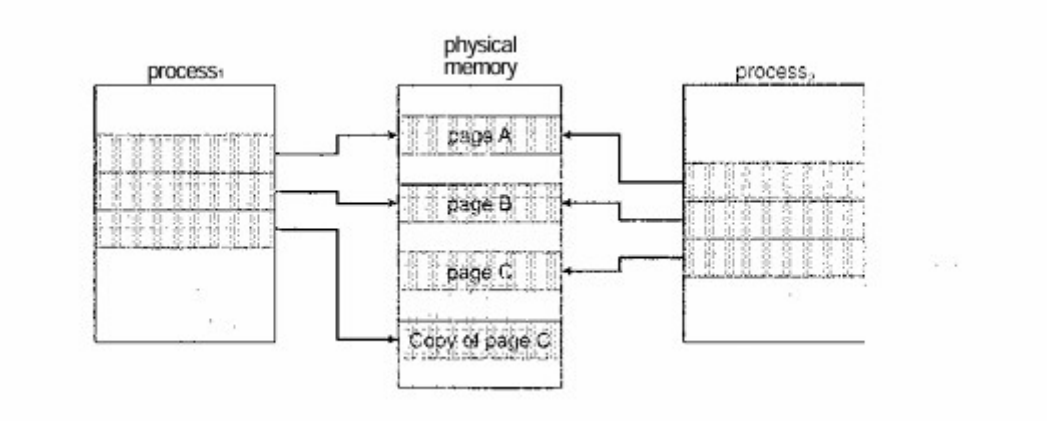


Figure 13.8 After process 1 modifies page C

13.4 Page Replacement

In our previous discussion on page-fault rates, we assumed that each page would only fault once when it is first accessed. However, this assumption isn't entirely accurate. For example, if a process consists of ten pages but only uses five, demand paging helps save I/O time by not loading the unused pages into memory. Additionally, by increasing the degree of multiprogramming (i.e., running more processes), we can make more efficient use of the

available memory. With 40 frames, we could run 8 processes, each requiring 5 pages, instead of just 4 processes, where each process requires 10 pages (and half of them go unused).

However, increasing the number of processes in this way can lead to **over-allocation of memory**. For instance, running 6 processes, each requiring 10 pages but only using 5, increases CPU utilization and throughput, leaving 10 frames unused. But, if any of these processes suddenly needs to use all 10 of its pages, we would need 60 frames, but only 40 are available.

Moreover, system memory isn't only used for storing program pages—**I/O buffers** also consume significant memory. This adds complexity to memory management, as we need to balance the allocation of memory between user processes and I/O buffers. Some systems allocate a fixed portion of memory for I/O, while others let both the user processes and the I/O subsystem compete for the available memory.

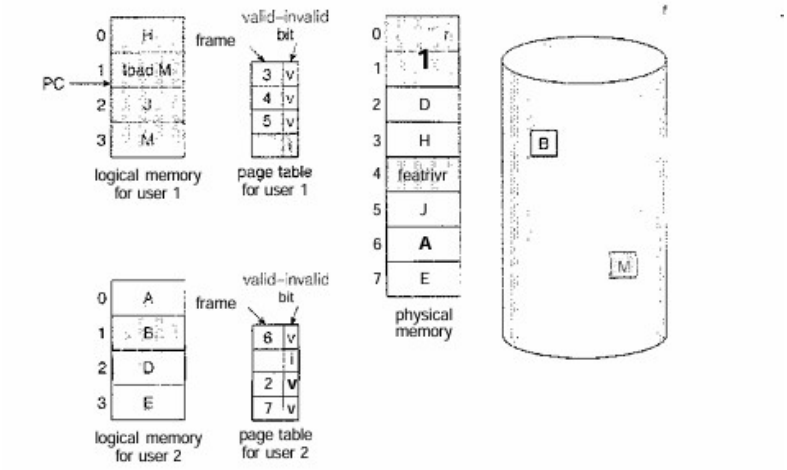


Figure 13.9 Need for page replacement.

When memory is fully allocated, and a page fault occurs, the operating system faces a challenge. It will determine where the needed page resides on the disk, but if there are no free frames available, it must find a way to free up memory to load the page.

At this point, the operating system has several choices:

1. **Terminate the Process**: The OS could terminate the user process that caused the page fault. However, this would defeat the purpose of **demand paging**, which is designed to improve system utilization and throughput without the user being aware of the paging system. This is not the best solution.
2. **Swap Out a Process**: The OS could swap out a process, freeing all its memory frames and reducing the number of processes running concurrently. This can be an effective solution in some cases, which we will discuss in more detail in Section 9.6.

3. **Page Replacement**: The most common solution is **page replacement**. When a page fault occurs and there are no free frames, the OS uses a page replacement algorithm to determine which pages to swap out of memory, making space for the required page. This allows the system to continue running multiple processes efficiently, even when memory is fully utilized.

While demand paging allows for more efficient memory use, increasing the number of processes can lead to over-allocation, which may cause memory shortages. When this happens, the operating system must choose between terminating processes, swapping processes out of memory, or using page replacement algorithms to free up space for the required pages. The most common approach is page replacement, which ensures that memory is allocated effectively and helps maintain system performance.

13.4.1 Basic Page Replacement

Page replacement operates by addressing situations where no free frame is available in memory. If no frame is free, the system must use a page-replacement algorithm to select a "victim" frame that can be evicted. Here's the process:

1. **Locate the desired page** on the disk.
2. **Find a free frame**:
    - If a free frame is available, use it to load the requested page.
    - If no free frames are available, apply a page-replacement algorithm to select a victim frame.
3. **Write the victim frame to disk** (if modified), and update the page and frame tables accordingly.
4. **Load the desired page** into the newly freed frame and update the page and frame tables.
5. **Restart the user process**.

It's important to note that if no frames are free, this process involves two I/O operations—one to swap the victim frame out and another to load the requested page in—effectively doubling the page-fault service time and increasing the effective access time.

Using the Modify Bit (Dirty Bit)

To optimize this process and reduce the overhead, a **modify bit** (or dirty bit) can be used. This bit is associated with each page or frame and is set whenever a page is modified. When a page needs to be replaced, the operating system checks the modify bit:

- If the modify bit is **set**, the page has been changed and must be written back to disk.
- If the modify bit is **not set**, the page has not been modified and does not need to be written back to disk, saving time and reducing I/O operations.

This technique is especially useful for **read-only pages**, such as executable code, which cannot be modified and can safely be discarded without needing to write them back to disk.

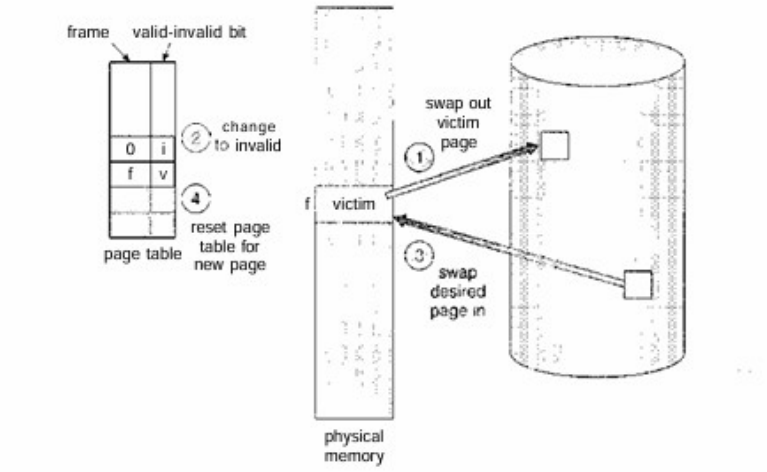


Figure 13.10 Page replacement

Demand Paging and Memory Management

Page replacement is a crucial part of demand paging, which separates logical memory from physical memory. This mechanism allows for much larger virtual memory than the actual physical memory available. For example, a process requiring twenty pages can be executed in just ten frames by using demand paging and a page-replacement algorithm to free frames when necessary. If a modified page is replaced, it is written to disk and can be loaded back into memory when needed.

Frame Allocation and Page Replacement Challenges

To effectively implement demand paging, two primary challenges must be addressed:

1. **Frame-allocation algorithm**: This determines how memory is distributed among multiple processes.
2. **Page-replacement algorithm**: This determines which pages are swapped out when new pages need to be loaded.

Choosing efficient algorithms is crucial, as disk I/O is a costly operation. Even small improvements in these algorithms can significantly boost system performance.

Evaluating Page Replacement Algorithms

The performance of a page-replacement algorithm is evaluated based on its ability to minimize page faults. A reference string, which consists of a series of memory accesses (page references), is used to test algorithms. The number of page faults depends on how well the algorithm predicts which pages are likely to be used next.

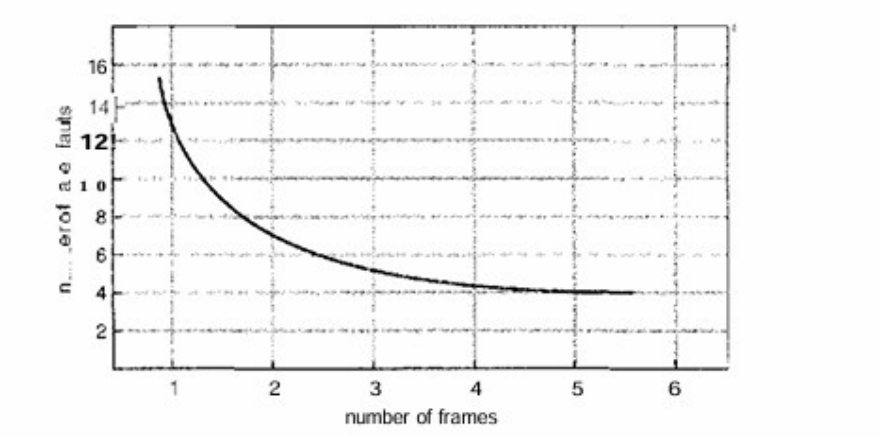


Figure 13.11 Graph of page faults versus number of frames

For instance, given the following reference string:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

And assuming a system with **three frames**, the number of page faults varies depending on the algorithm used.

The performance of page-replacement algorithms is often depicted in a graph, where the number of page faults decreases as the number of frames increases. However, even with more frames, the number of page faults eventually levels off.

In conclusion, efficient page replacement and frame allocation are critical for maximizing system performance, particularly in environments with limited physical memory.

13.4 PAGE REPLACEMENT

13.4.1 Basic Page Replacement

The **FIFO** page replacement algorithm is a simple method that replaces the oldest page in memory when a new page is needed. The pages are managed in a queue, and the page that enters memory first is the first to be replaced when a page fault occurs.

How FIFO Works:

1. **Queue Maintenance**: Pages in memory are kept in a queue, with the oldest page at the front.
2. **Page Fault Handling**: When a page fault occurs:
    - If there's a free frame, load the new page.
    - If there are no free frames, replace the page at the front of the queue (the oldest page).

Example:

Consider 3 page frames and a reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3

- **Reference 7**: Load 7 → Memory: [7]
- **Reference 0**: Load 0 → Memory: [7, 0]
- **Reference 1**: Load 1 → Memory: [7, 0, 1]
- **Reference 2**: Replace 7 → Memory: [2, 0, 1]
- **Reference 0**: No fault → Memory: [2, 0, 1]
- **Reference 3**: Replace 1 → Memory: [2, 0, 3]
- **Reference 4**: Replace 2 → Memory: [4, 0, 3]
- **Reference 2**: Replace 3 → Memory: [4, 0, 2]
- **Reference 3**: Replace 0 → Memory: [4, 3, 2]

Result: 8 page faults.

Advantages:

- Simple and easy to implement.

Disadvantages:

- **Belady's Anomaly**: Adding more frames can sometimes increase the page fault rate.
- Inefficient as it does not consider which pages are frequently used.

13.4.2 FIFO Page Replacement

FIFO (First-In-First-Out) is one of the simplest page replacement algorithms used in virtual memory management. The primary goal of a page replacement algorithm is to decide which page in memory should be replaced when a page fault occurs — that is, when the requested page is not currently in memory. FIFO follows a straightforward approach based on the order in which pages were loaded into memory.

Concept of FIFO Page Replacement

In FIFO, pages are replaced in the order in which they were brought into memory. The first page that was loaded into memory is the first one to be replaced when a new page needs to be brought in. This method is based on the **queue** data structure where pages are added to the rear and removed from the front.

- **Queue Representation**: FIFO maintains a queue of pages in memory. When a page is loaded into memory, it is placed at the rear of the queue. When a page fault occurs and a new page must be loaded into memory, the page at the front of the queue (the oldest page) is replaced.

- **Page Fault Handling**: When a page fault occurs (i.e., the requested page is not in memory), FIFO checks the oldest page in memory (the page at the front of the queue). If the page is not in memory, it is replaced with the new page, and the new page is inserted at the rear of the queue.

Working of FIFO Page Replacement Algorithm

1. **Initialization**: When a process starts, the operating system allocates a certain number of frames (slots) in physical memory for the pages of the process. The queue initially starts empty.
2. **Page Fault**: When a page is needed but not found in memory (a page fault), the page is loaded into memory. If there is still space available in memory (not all frames are full), the new page is simply added to the queue.
3. **Page Replacement**: When memory is full and a new page must be loaded into memory, the page at the front of the queue (the oldest page) is removed to make room for the new page. The new page is then inserted at the rear of the queue.
4. **Repeat**: This process continues each time a page fault occurs. Pages are continuously swapped in and out of memory based on their order of arrival.

Example of FIFO Page Replacement

Consider a system with 3 page frames and a reference string (sequence of page requests) as follows:

Reference String: 7, 0, 1, 2, 0, 3, 0, 4

Step-by-Step Execution:

- Initially, memory is empty. We begin with an empty queue for the 3 page frames.

1. **Page 7**: Page 7 is requested. It's not in memory, so it is loaded into frame 1. Queue: [7].
2. **Page 0**: Page 0 is requested. It’s not in memory, so it is loaded into frame 2. Queue: [7, 0].
3. **Page 1**: Page 1 is requested. It’s not in memory, so it is loaded into frame 3. Queue: [7, 0, 1].
4. **Page 2**: Page 2 is requested. It’s not in memory, and memory is full. The first page (7) is replaced with page 2. Queue: [0, 1, 2].
5. **Page 0**: Page 0 is requested. It’s already in memory, so no replacement occurs. Queue remains: [0, 1, 2].
6. **Page 3**: Page 3 is requested. It’s not in memory, and memory is full. The first page (1) is replaced with page 3. Queue: [0, 2, 3].
7. **Page 0**: Page 0 is requested. It’s already in memory, so no replacement occurs. Queue remains: [0, 2, 3].

8. **Page 4**: Page 4 is requested. It's not in memory, and memory is full. The first page (2) is replaced with page 4. Queue: [0, 3, 4].

Thus, after processing the reference string, the final contents of the frames are [0, 3, 4].

Advantages of FIFO Page Replacement

1. **Simplicity**: FIFO is easy to understand and implement. The algorithm requires only a simple queue to keep track of the pages in memory, making it straightforward for operating systems to manage memory.
2. **Efficiency in Small Systems**: In systems with limited memory and few processes, FIFO can perform reasonably well, as the number of page faults may be manageable.

Disadvantages of FIFO Page Replacement

1. **Belady's Anomaly**: One of the major drawbacks of FIFO is the phenomenon known as **Belady's anomaly**. This is where increasing the number of page frames can sometimes result in more page faults. This counterintuitive behavior occurs because FIFO may replace a page that is actually needed soon, even though there may be room for additional pages.
2. **Inefficient Replacement**: FIFO does not consider the frequency or recency of page usage. It purely replaces the oldest page, which might not be the best choice. Pages that are heavily used might be replaced in favor of pages that have been unused for a long time, leading to inefficient memory utilization.
3. **High Page Fault Rate**: For certain reference strings and under specific conditions, FIFO can lead to a high page fault rate because it does not take into account the actual access pattern of the program, leading to unnecessary replacements.

13.4.3 Optimal Page Replacement

The **Optimal Page Replacement** (OPT) algorithm replaces the page that will not be used for the longest period of time in the future. It minimizes the number of page faults by making the optimal choice of which page to remove, but it requires knowledge of future page references, which makes it impractical for real systems.

How OPT Works:

1. **Future Knowledge**: The algorithm examines all future page references.
2. **Page Fault Handling**: When a page fault occurs:
    - If there's a free frame, load the new page.
    - If there are no free frames, replace the page that will not be needed for the longest period in the future.

Example:

Consider 3 page frames and a reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3

- **Reference 7**: Load 7 → Memory: [7]
- **Reference 0**: Load 0 → Memory: [7, 0]
- **Reference 1**: Load 1 → Memory: [7, 0, 1]
- **Reference 2**: Replace 7 (7 is used furthest in the future) → Memory: [2, 0, 1]
- **Reference 0**: No fault → Memory: [2, 0, 1]
- **Reference 3**: Replace 1 (1 is used furthest in the future) → Memory: [2, 0, 3]
- **Reference 4**: Replace 2 (2 is used furthest in the future) → Memory: [4, 0, 3]
- **Reference 2**: Load 2 → Memory: [4, 0, 2]
- **Reference 3**: No fault → Memory: [4, 0, 2]

Result: 6 page faults.

Advantages:

- Optimal in terms of page faults (minimizes them).

Disadvantages:

- Impractical in real systems as it requires knowledge of future references, which is generally unavailable.

13.4.4 LRU Page Replacement

The **Least Recently Used (LRU)** algorithm replaces the page that has not been used for the longest period of time. It approximates the Optimal Page Replacement strategy by assuming that pages used recently are more likely to be used again in the near future.

How LRU Works:

1. **Page Fault Handling**: When a page fault occurs:
    - If there's a free frame, the new page is loaded into memory.
    - If all frames are full, the least recently used page is replaced.
2. **Tracking Access**: LRU can be implemented using a counter or a stack to keep track of page access order.

Example:

Consider 3 page frames and a reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3

- **Reference 7**: Load 7 → Memory: [7]
- **Reference 0**: Load 0 → Memory: [7, 0]
- **Reference 1**: Load 1 → Memory: [7, 0, 1]
- **Reference 2**: Replace 7 (7 is least recently used) → Memory: [2, 0, 1]

- **Reference 0**: No fault → Memory: [2, 0, 1]
- **Reference 3**: Replace 1 (1 is least recently used) → Memory: [2, 0, 3]
- **Reference 0**: No fault → Memory: [2, 0, 3]
- **Reference 4**: Replace 2 (2 is least recently used) → Memory: [4, 0, 3]
- **Reference 2**: Replace 3 (3 is least recently used) → Memory: [4, 0, 2]
- **Reference 3**: No fault → Memory: [4, 0, 2]

Result: 6 page faults.

Advantages:

- Fairly simple to implement and works well in most scenarios.

Disadvantages:

- May require significant overhead for tracking the "least recently used" page.

13.4.5 LRU-Approximation Page Replacement

The **LRU-Approximation** algorithm is a simplified approach to implement the Least Recently Used (LRU) page replacement, which avoids the high overhead of keeping track of every page's access time. It approximates the behavior of LRU with less memory and processing cost.

How LRU-Approximation Works:

- **Clock Algorithm**: One common method of approximating LRU is the **Clock algorithm**. Each page is assigned a "reference bit" (or "use bit"). This bit is set to 1 when a page is accessed. The page replacement process proceeds like a circular queue (like a clock) where each page has its reference bit checked.
    - If the reference bit is 1, it is cleared, and the algorithm moves to the next page.
    - If the reference bit is 0, that page is selected for replacement.
- This provides an approximation of LRU because it doesn't require full access history, only recent accesses.

Example:

Consider a system with 3 page frames and a reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, using a simple Clock Algorithm:

- **Initially**: Pages: [empty, empty, empty], Reference Bits: [0, 0, 0]
- **Reference 7**: Load 7 → Pages: [7, empty, empty], Reference Bits: [1, 0, 0]
- **Reference 0**: Load 0 → Pages: [7, 0, empty], Reference Bits: [1, 1, 0]
- **Reference 1**: Load 1 → Pages: [7, 0, 1], Reference Bits: [1, 1, 1]

- **Reference 2**: Check page 7 → Reference bit is 1, clear bit, check 0 → Reference bit is 1, clear bit, check 1 → Reference bit is 1, clear bit, load 2 → Pages: [2, 0, 1], Reference Bits: [0, 0, 1]
- **Reference 0**: Check page 2 → Reference bit is 0, select page 2 for replacement → Pages: [2, 0, 1], Reference Bits: [1, 0, 1]
- **...continue similarly**.

Result: 7 page faults.

Advantages:

- Simpler and more efficient than full LRU, with less memory usage.
- Suitable for systems where access patterns are not easily predictable.

Disadvantages:

- Provides an approximation, so it may not be as optimal as true LRU.
- Still requires some overhead to track the reference bits.

13.4.5.1 Additional-Reference-Bits Algorithm

The **Additional-Reference-Bits (ARB) Algorithm** is another approximation of the **LRU (Least Recently Used)** page replacement algorithm. It is used to reduce the overhead of maintaining the exact history of page references by using an additional reference bit for each page. This bit helps to track whether a page has been recently accessed or not.

How the Additional-Reference-Bits Algorithm Works:

1. Each page in memory has a reference bit, which is set when the page is accessed.
2. The reference bits are periodically shifted (usually at regular intervals or after each memory access cycle). The shifting operation allows the algorithm to track the relative recency of access.
3. When a page needs to be replaced, the algorithm looks for the page with the lowest "history" of access. A page with a reference bit of 0 and a shifted reference bit of 0 is replaced first.

This mechanism is similar to the **Clock** algorithm but provides better approximations for LRU by tracking recent history.

Example:

Consider a system with 4 frames and the following reference string: 1, 2, 3, 4, 1, 2, 5, and assume that reference bits are cleared after every 4 memory accesses.

- **Initially**: Pages: [empty, empty, empty, empty], Reference Bits: [0, 0, 0, 0]
- **Reference 1**: Load 1 → Pages: [1, empty, empty, empty], Reference Bits: [1, 0, 0, 0]
- **Reference 2**: Load 2 → Pages: [1, 2, empty, empty], Reference Bits: [1, 1, 0, 0]
- **Reference 3**: Load 3 → Pages: [1, 2, 3, empty], Reference Bits: [1, 1, 1, 0]
- **Reference 4**: Load 4 → Pages: [1, 2, 3, 4], Reference Bits: [1, 1, 1, 1]
- **After 4 accesses**, shift reference bits: [1, 1, 1, 1] becomes [0, 0, 0, 0]
- **Reference 1**: Update Reference Bits → Pages: [1, 2, 3, 4], Reference Bits: [1, 0, 0, 0]

- **Reference 2**: Update Reference Bits → Pages: [1, 2, 3, 4], Reference Bits: [1, 1, 0, 0]
- **Reference 5**: Replace the least recently used page (4) → Pages: [1, 2, 3, 5], Reference Bits: [1, 1, 0, 1]

Result: The page replacement follows the logic of Least Recently Used (LRU) without maintaining the complete access history.

Advantages:

- Reduces the overhead of full LRU by tracking the recency of access with a simple reference bit mechanism.
- More efficient than true LRU, but still provides good approximation.

Disadvantages:

- It may still incur overhead due to shifting reference bits, and does not fully replicate the behavior of LRU.

13.4.5.2 Second-Chance Algorithm

The **Second-Chance Algorithm** is a page replacement algorithm that is an enhancement to the **FIFO (First-In-First-Out)** algorithm. It addresses the problem of FIFO replacing pages that are still actively used. The second-chance algorithm gives pages that have been in memory a "second chance" before being replaced.

How the Second-Chance Algorithm Works:

1. Each page frame has a **reference bit** (also called a use bit).
2. When a page is first loaded into memory, its reference bit is set to 1.
3. When a page needs to be replaced, the algorithm checks the page in the FIFO queue:
    - If the reference bit is **1**, it is given a second chance. The reference bit is reset to **0**, and the page is moved to the end of the FIFO queue.
    - If the reference bit is **0**, the page is replaced with the new page.
4. This process continues until a page is replaced.

This mechanism provides a simple approximation of the **LRU (Least Recently Used)** algorithm by keeping recently used pages in memory longer.

Example:

Assume we have a system with **4 frames** and the following reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0.

- **Initially**: Pages: [empty, empty, empty, empty], Reference Bits: [0, 0, 0, 0]

1. **Reference 7**: Load page 7 → Pages: [7, empty, empty, empty], Reference Bits: [1, 0, 0, 0]
2. **Reference 0**: Load page 0 → Pages: [7, 0, empty, empty], Reference Bits: [1, 1, 0, 0]
3. **Reference 1**: Load page 1 → Pages: [7, 0, 1, empty], Reference Bits: [1, 1, 1, 0]
4. **Reference 2**: Load page 2 → Pages: [7, 0, 1, 2], Reference Bits: [1, 1, 1, 1]
5. **Reference 0**: Page 0 is accessed, and reference bit is set → Pages: [7, 0, 1, 2], Reference Bits: [0, 1, 1, 1]
6. **Reference 3**: Replace page 7 (it has reference bit 0) → Pages: [3, 0, 1, 2], Reference Bits: [0, 1, 1, 1]
7. **Reference 0**: Page 0 is accessed again, so its reference bit is set → Pages: [3, 0, 1, 2], Reference Bits: [0, 1, 1, 1]
8. **Reference 4**: Replace page 1 (its reference bit is 0) → Pages: [3, 0, 4, 2], Reference Bits: [0, 1, 0, 1]

Result:

In this example, the second-chance algorithm has given "second chances" to pages that were frequently accessed (such as page 0), keeping them in memory for longer than FIFO would have done.

Advantages:

- Provides a better approximation to LRU than FIFO.
- Simple and efficient, requiring only a reference bit and a queue for page replacement.

Disadvantages:

- Does not perform as well as true LRU in all cases.
- Can still suffer from overhead if the number of pages in memory is large.

13.4.5.3 Enhanced Second-Chance Algorithm

The **Enhanced Second-Chance Algorithm** is an improvement over the basic **Second-Chance Algorithm**. It improves decision-making by using **two reference bits** instead of just one. This modification helps distinguish between different types of page usage (e.g., if a page was recently accessed but has not been modified).

How the Enhanced Second-Chance Algorithm Works:

1. Each page frame has **two reference bits**:
    - **R bit (Reference bit)**: Indicates if the page has been accessed.
    - **M bit (Modified bit)**: Indicates whether the page has been modified (written to) since it was loaded into memory.
2. When a page is accessed, the **R bit** is set to 1. If the page is modified, the **M bit** is set to 1.

3. During page replacement:
    - If the page has both **R = 0** and **M = 0** (not recently used and not modified), it is replaced immediately.
    - If **R = 1** and **M = 0** (recently accessed but not modified), it is given a second chance and moved to the back of the queue, and the **R bit** is reset to 0.
    - If **R = 0** and **M = 1** (not recently accessed but modified), the page is written back to disk, and then it is replaced.
    - If **R = 1** and **M = 1** (recently accessed and modified), the page is given a second chance and moved to the back of the queue, and the **R bit** is reset to 0.

This algorithm provides a more intelligent way of handling page replacements, giving higher priority to pages that are not only recently accessed but also modified.

Example:

Assume we have a system with **4 frames** and the following reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0. We will also track the **R** (reference) and **M** (modified) bits.

- **Initially**: Pages: [empty, empty, empty, empty], Reference Bits: [0, 0, 0, 0], Modified Bits: [0, 0, 0, 0]

1. **Reference 7**: Load page 7 → Pages: [7, empty, empty, empty], R: [1, 0, 0, 0], M: [0, 0, 0, 0]
2. **Reference 0**: Load page 0 → Pages: [7, 0, empty, empty], R: [1, 1, 0, 0], M: [0, 0, 0, 0]
3. **Reference 1**: Load page 1 → Pages: [7, 0, 1, empty], R: [1, 1, 1, 0], M: [0, 0, 0, 0]
4. **Reference 2**: Load page 2 → Pages: [7, 0, 1, 2], R: [1, 1, 1, 1], M: [0, 0, 0, 0]
5. **Reference 0**: Page 0 accessed → Pages: [7, 0, 1, 2], R: [0, 1, 1, 1], M: [0, 0, 0, 0]
6. **Reference 3**: Replace page 7 (R = 0, M = 0) → Pages: [3, 0, 1, 2], R: [0, 1, 1, 1], M: [0, 0, 0, 0]
7. **Reference 0**: Page 0 accessed again → Pages: [3, 0, 1, 2], R: [0, 1, 1, 1], M: [0, 0, 0, 0]
8. **Reference 4**: Replace page 1 (R = 0, M = 0) → Pages: [3, 0, 4, 2], R: [0, 1, 0, 1], M: [0, 0, 0, 0]

In this example, the Enhanced Second-Chance Algorithm gives "second chances" to pages that were recently used (R = 1), while also taking into account whether the page has been modified (M = 1), which helps in deciding which page to replace.

Advantages:

- More intelligent page replacement by considering both access and modification history.

- Better performance than the simple second-chance algorithm because it incorporates more information.

Disadvantages:

- Requires additional bits (R and M) for each page frame, which increases memory overhead.
- More complex than the basic second-chance algorithm.

13.4.6 Counting-Based Page Replacement

The **Counting-Based Page Replacement** algorithm is a simple page replacement strategy that uses a **counter** to track how many times a page has been accessed. The page with the lowest access count is the one chosen for replacement. This algorithm is based on the principle that the least frequently accessed page is less likely to be needed in the future.

How It Works:

1. **Counter for Each Page**: Each page has a counter associated with it. The counter keeps track of how often the page has been accessed.
2. **Page Access**: Each time a page is referenced, its counter is incremented.
3. **Page Replacement**: When a page fault occurs and a replacement is necessary, the page with the smallest counter value (least frequently used) is selected for replacement.

Example:

Assume we have 4 page frames and the following reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0. Initially, all counters are set to 0.

- **Initially**: Pages: [empty, empty, empty, empty], Counters: [0, 0, 0, 0]

1. **Reference 7**: Load page 7 → Pages: [7, empty, empty, empty], Counters: [1, 0, 0, 0]
2. **Reference 0**: Load page 0 → Pages: [7, 0, empty, empty], Counters: [1, 1, 0, 0]
3. **Reference 1**: Load page 1 → Pages: [7, 0, 1, empty], Counters: [1, 1, 1, 0]
4. **Reference 2**: Load page 2 → Pages: [7, 0, 1, 2], Counters: [1, 1, 1, 1]
5. **Reference 0**: Page 0 accessed → Pages: [7, 0, 1, 2], Counters: [2, 1, 1, 1]
6. **Reference 3**: Replace page 7 (least accessed) → Pages: [3, 0, 1, 2], Counters: [2, 1, 1, 1]
7. **Reference 0**: Page 0 accessed → Pages: [3, 0, 1, 2], Counters: [3, 1, 1, 1]
8. **Reference 4**: Replace page 1 (least accessed) → Pages: [3, 0, 4, 2], Counters: [3, 1, 2, 1]

In this example, the page with the lowest counter (least accessed) is selected for replacement when necessary.

Advantages:

- Simple and easy to implement.
- Fairly effective in workloads where certain pages are accessed infrequently.

Disadvantages:

- May not always give the best performance because it does not account for the recency of access (i.e., it only tracks frequency, not when the page was last accessed).
- Can be inefficient in scenarios where some pages are accessed very frequently but are still replaced because their access count may be low compared to other pages.

13.4.7 Page-Buffering Algorithms

Page-buffering algorithms are used to enhance the efficiency of **page replacement** by reducing the number of I/O operations needed when a page fault occurs. In these algorithms, the system maintains a pool of **buffers** (usually pages) that can be quickly accessed and used to handle page faults.

How It Works:

1. **Buffered Pages**: The system maintains a buffer pool, which consists of pages that are preloaded from disk or are available for quick use.
2. **Page Fault Handling**: When a page fault occurs, the system first checks if the page is available in the buffer pool. If it is, the page is transferred to the process. If the page is not in the buffer, it is brought in from disk, and the buffer pool is updated accordingly.
3. **Reduced Disk I/O**: By using page buffers, disk I/O operations can be minimized as multiple page faults can be resolved by reusing pages from the buffer pool.

Example:

Consider a system with 3 page frames and a page buffer pool of 2 pages. The reference string is: 7, 0, 1, 7, 0, 3, 4.

Step-by-Step:

1. **Page 7**: Miss, load page 7 into a buffer → Pages: [7, empty, empty], Buffer: [7]
2. **Page 0**: Miss, load page 0 into a frame → Pages: [7, 0, empty], Buffer: [7]
3. **Page 1**: Miss, load page 1 into a frame → Pages: [7, 0, 1], Buffer: [7]
4. **Page 7**: Hit, page 7 is in the buffer → Pages: [7, 0, 1], Buffer: [7]
5. **Page 0**: Hit, page 0 is already in a frame → Pages: [7, 0, 1], Buffer: [7]
6. **Page 3**: Miss, replace page 1 in a frame → Pages: [7, 0, 3], Buffer: [7, 3]
7. **Page 4**: Miss, replace page 0 in a frame → Pages: [7, 4, 3], Buffer: [7, 3]

In this example, whenever a page fault occurs, the buffer helps reduce the number of disk accesses by keeping frequently accessed pages in the buffer pool. When a page is requested, the buffer is checked first before reading from the disk.

Advantages:

- **Reduced Disk I/O**: By storing frequently accessed pages in buffers, the number of I/O operations is minimized.
- **Improved Performance**: Buffering improves system responsiveness and reduces the time needed to handle page faults.
- **Better Handling of Page Faults**: The buffer pool can provide a temporary solution to resolve page faults quickly without accessing the slower disk storage.

Disadvantages:

- **Buffer Management**: Managing the buffer pool adds complexity to the system, especially in determining which pages should be kept in the buffer.
- **Buffer Pool Size**: The size of the buffer pool is limited, and once it is full, the system must decide which pages to evict to make room for new ones.

Page-buffering algorithms are commonly used in systems where page faults happen frequently, and minimizing I/O operations is crucial for system performance.

13.4.8 Applications and Page Replacement

Page replacement algorithms play a critical role in operating systems that utilize **virtual memory**. These algorithms help manage how pages are swapped between **main memory** and **disk storage** (swap space) to optimize system performance, especially when physical memory is insufficient to hold all the pages of active processes.

Key Applications of Page Replacement:

1. **Multiprogramming**: Allows multiple processes to run concurrently despite limited physical memory. Page replacement algorithms help manage which pages should stay in memory when multiple processes are running.
2. **Efficient Memory Usage**: By replacing less frequently used pages with those that are currently needed, page replacement ensures that memory is used efficiently.
3. **Improved System Throughput**: Optimizing page replacement can minimize page faults, thus improving the throughput of the system, making the execution of processes faster.

4. **Real-Time Systems**: In systems with real-time constraints, page replacement can help ensure that critical processes have quick access to their required pages without excessive delays.

Example: A Simple Application of Page Replacement

Consider a system with **three frames** and a page reference string:
7, 0, 1, 2, 0, 3, 0, 4, 5

We will use the **FIFO (First-In-First-Out)** page replacement algorithm.

Steps:

1. **Page 7**: Page fault, load page 7 → Frames: [7, -, -]
2. **Page 0**: Page fault, load page 0 → Frames: [7, 0, -]
3. **Page 1**: Page fault, load page 1 → Frames: [7, 0, 1]
4. **Page 2**: Page fault, replace page 7 (oldest) with page 2 → Frames: [2, 0, 1]
5. **Page 0**: No page fault, page 0 is already in memory → Frames: [2, 0, 1]
6. **Page 3**: Page fault, replace page 1 with page 3 → Frames: [2, 0, 3]
7. **Page 0**: No page fault, page 0 is still in memory → Frames: [2, 0, 3]
8. **Page 4**: Page fault, replace page 2 with page 4 → Frames: [4, 0, 3]
9. **Page 5**: Page fault, replace page 3 with page 5 → Frames: [4, 0, 5]

Summary:

- **Page faults**: 9
- **Final page frame contents**: [4, 0, 5]

In this example, the FIFO page replacement algorithm replaces pages in the order in which they were loaded into memory. Despite its simplicity, FIFO may not always be the most efficient algorithm, as older pages are evicted without considering whether they are frequently accessed or not.

In real-world applications, more sophisticated algorithms like **LRU (Least Recently Used)** or **Optimal Page Replacement** are typically used to minimize page faults and enhance system performance.

13.5 ALLOCATION OF FRAMES

Memory allocation for processes involves dividing the available free frames among them. In a single-user system with 93 free frames, the operating system allocates the required frames for system tasks, and the remaining frames are used for the user process. If the user process exceeds the available frames, page replacement occurs. Variations include reserving frames for system tasks or ensuring a few free frames remain available during page swaps.

13.5.1 Minimum Number of Frames

Frame allocation strategies must balance performance and system constraints. We cannot allocate more frames than available, and a minimum number of frames is required for process execution. Allocating too few frames increases the page-fault rate, slowing execution. The minimum number of frames depends on the architecture, considering factors like instruction complexity and indirection levels. The maximum number is constrained by available physical memory, and efficient frame allocation lies between these limits.

13.5.2 Allocation Algorithms

In virtual memory management, allocation algorithms are used to manage how memory pages are assigned to physical memory (RAM) and how virtual memory is mapped to it. These algorithms are essential to optimize the use of memory and ensure efficient process execution.

13.5.3 Global versus Local Allocation

Page replacement algorithms can be classified into **global replacement** and **local replacement**. In global replacement, a process can select a replacement frame from any process, potentially taking frames from lower-priority processes. Local replacement restricts processes to selecting frames from their own allocation. Global replacement may lead to unpredictable performance due to external factors, while local replacement ensures stability but may limit flexibility. Global replacement often results in better system throughput and is therefore more commonly used.

13.6 THRASHING

If a low-priority process is allocated fewer frames than required, it is suspended, and its pages are swapped out to free frames. This introduces a swap-in, swap-out mechanism. A process with insufficient frames will repeatedly page-fault, replacing pages that are needed immediately, leading to thrashing. Thrashing occurs when a process spends more time paging than executing, severely affecting performance.

13.6.1 Cause of Thrashing

Thrashing occurs when a system spends most of its time swapping pages in and out of memory instead of executing processes. This is typically caused by **insufficient memory allocation** for processes, leading to excessive page faults. When processes do not have enough frames to keep their working set in memory, they continuously replace pages that are immediately needed again.

Example:

Consider two processes, A and B.

- Process A needs 4 pages, but only 2 frames are allocated.

- Process B needs 3 pages, but only 2 frames are allocated.

Both processes keep page-faulting as they continuously replace pages that are still in use. This constant swapping causes thrashing, slowing down the system significantly.

13.6.2 Working-Set Model

The **Working-Set Model** is a memory management technique that aims to optimize the allocation of frames by ensuring that a process keeps its frequently used pages in memory. The "working set" refers to the set of pages that a process is actively using within a given time window. The goal is to keep this working set in memory to reduce page faults and improve performance.

Example:

Consider a process with 10 pages. If the time window (or working-set window) is set to 3 pages, the system will track the 3 most recently used pages at any given time. As the process accesses pages, it will load the pages within the working set into memory and evict less frequently used pages. This way, the process minimizes page faults by keeping its actively used pages in memory.

This model helps prevent thrashing by adjusting the number of pages allocated based on the process's actual usage pattern.

13.6.3 Page-Fault Frequency

The **Page-Fault Frequency (PFF) algorithm** is a memory management technique used to adjust the number of pages allocated to a process based on its page fault rate. If a process experiences too many page faults, it is allocated more frames; if page faults are low, it may have some frames taken away. The idea is to keep the page fault rate within an acceptable range for efficient execution.

Example:

Consider a process with 4 frames allocated:

- If the page fault rate exceeds a threshold (e.g., more than 1 page fault every 10 ms), the system allocates an additional frame to the process.
- If the page fault rate is low (e.g., less than 1 page fault every 10 ms), it may reduce the number of frames allocated to the process.

This dynamic adjustment helps optimize memory usage and performance by maintaining an appropriate balance between memory allocation and page faults.

13.7 MEMORY-MAPPED FILES

Memory-Mapped Files allow files to be mapped directly into the virtual memory space of a process. This means that the content of the file is treated as part of the process's memory, allowing faster access to file data and easier sharing of data between processes. The operating system automatically handles the mapping of file contents into memory.

Example:

If a process needs to read from a large file, rather than using traditional I/O operations (like read()), it can use memory-mapping. For instance, in a C program, the mmap() function can map a file into the process's address space. Once mapped, the process can access the file contents just like it would access any other part of memory.

```
int fd = open("example.txt", O_RDWR);
char *file_content = mmap(NULL, file_size, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
```

This method improves performance by eliminating the need for explicit read and write operations, making file I/O more efficient.

13.7.1 Basic Mechanism

The basic mechanism behind **memory-mapped files** involves mapping the contents of a file directly into a process's address space. This allows the process to access file data as if it were part of its memory, providing faster access to the data without needing explicit read or write system calls. The operating system handles the mapping, ensuring that the file's contents are read into memory when needed.

Example:

1. **Mapping a File**: A file is mapped into the process's virtual memory using system calls like mmap() in Unix/Linux.
2. **Accessing Data**: Once mapped, the process can access the file data just like any other memory region, by directly reading or writing to the mapped memory.
3. **Synchronization**: Changes made to the mapped memory are automatically written to the file (if using MAP_SHARED).

```
int fd = open("data.txt", O_RDWR);  // Open the file for reading and writing
char *mapped_data = mmap(NULL, file_size, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);  // Map file into memory
// Now, mapped_data can be accessed directly as memory.
```

This method improves performance, especially for large files, by avoiding the overhead of repeated I/O operations.

13.7.2 Shared Memory in the Win32 API

In the **Win32 API**, shared memory is implemented through memory-mapped files. This allows multiple processes to map a file or a portion of memory into their address space and share data efficiently. Shared memory is used for inter-process communication (IPC) by mapping the same memory region in different processes.

Basic Mechanism:

1. **Create or Open a Memory-Mapped File**: A memory-mapped file is created using CreateFileMapping() or opened using OpenFileMapping().
2. **Map the File into Memory**: The file is mapped into the address space of the process using MapViewOfFile().
3. **Access Shared Memory**: Multiple processes can access the same mapped memory, allowing them to share data.

Example:

This example shows how to create shared memory and map it into a process's address space in Windows:

```
#include <windows.h>
#include <stdio.h>
int main() {
   // Create a memory-mapped file
   HANDLE hMapFile = CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
PAGE_READWRITE, 0, sizeof(int), "SharedMemory");

   if (hMapFile == NULL) {
      printf("Could not create file mapping. Error: %ld\n", GetLastError());
      return 1;
   }
   // Map the file into memory
   int* pSharedMemory = (int*)MapViewOfFile(hMapFile, FILE_MAP_ALL_ACCESS, 0,
0, sizeof(int));

   if (pSharedMemory == NULL) {
      printf("Could not map view of file. Error: %ld\n", GetLastError());
      return 1;
   }
   // Write to shared memory
   *pSharedMemory = 1234;
   printf("Written to shared memory: %d\n", *pSharedMemory);
```

```
    // Clean up
    UnmapViewOfFile(pSharedMemory);
    CloseHandle(hMapFile);
    return 0;
}
```

Explanation:

1. CreateFileMapping() creates a named memory-mapped file that can be shared between processes.
2. MapViewOfFile() maps this file into the process's address space.
3. The shared memory is accessed and modified by the process directly.

This mechanism allows efficient communication between multiple processes by accessing common memory, eliminating the need for more complex IPC methods like pipes or message queues.

13.7.3 Memory-Mapped I/O

Memory-Mapped I/O allows a process to interact with hardware devices by mapping device memory or registers directly into the system's address space. This means the device's memory can be accessed using standard memory operations (like reading and writing), rather than using slower I/O operations.

The operating system maps a range of physical memory addresses to virtual memory addresses, enabling the CPU to access the device directly via memory instructions, improving speed and efficiency.

Example:

In systems with hardware like graphics cards or sensors, memory-mapped I/O might map the device's memory space (e.g., video buffer) to the program's virtual memory. The program can then read from or write to this memory as if it were regular RAM, allowing faster interaction with the hardware

13.8 SUMMARY

This section covers key concepts of virtual memory systems, starting with an introduction to their role in separating logical and physical memory for efficient management. **Demand paging** is highlighted as a method to load pages only when needed, enhancing memory usage, with performance influenced by page-fault rates. **Copy-on-Write** is presented as a technique for shared memory optimization until modifications are required. Various **page replacement algorithms** (e.g., FIFO, Optimal, LRU, and approximations like second-chance) are discussed to manage memory effectively during page faults. The allocation of memory frames, including minimum frame requirements and global versus local strategies,

ensures balanced performance. **Thrashing**, caused by excessive page faults, is mitigated through the working-set model and page-fault frequency techniques. Finally, **memory-mapped files** provide efficient file access and interprocess communication mechanisms, with applications such as shared memory in the Win32 API.

13.9 KEY TERMS

Virtual Memory, Demand Paging, Page Replacement, FIFO Replacement, LRU, Replacement, Thrashing, Frame Allocation.

13.10 SELF-ASSESSMENT QUESTIONS

1. What is the primary purpose of a virtual memory system, as introduced in the background section?
2. How does demand paging optimize memory usage, and what factors affect its performance?
3. What is the concept of Copy-on-Write, and how does it reduce memory overhead?
4. What are the differences between FIFO, Optimal, and LRU page replacement algorithms?
5. How do global and local frame allocation strategies impact process performance?
6. What causes thrashing, and how can the working-set model help to prevent it?

13.11 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”, Seventh Edition, Wiley.

2. William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3. Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4. Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5. Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008).

NAME OF THE AUTHOR

Dr U Surya Kameswari

CHAPTER 14

FILE SYSTEM

AIM AND OBJECTIVES:

- Understand how file systems work.
- Learn how to interact with file systems.
- Explore design choices in file systems, such as file access, sharing, locking, and directory organization.
- Learn how to protect file systems.

STRUCTURE:

14.1 FILE CONCEPT

Computers store information on media like magnetic disks, tapes, and optical disks. The operating system provides a logical view of storage by defining files, abstracting the physical storage devices. Files, stored on non-volatile devices, retain data even after power failures or reboots.

A **file** is a named collection of related information on secondary storage, representing the smallest unit of logical storage. Files can store programs (source, object, or executable) or data (numeric, text, binary, etc.) and may be free-form (text files) or formatted (executable files).

File types include:

- **Text files**: Characters organized into lines or pages.
- **Source files**: Subroutines/functions with declarations and executable statements.
- **Object files**: Bytes in blocks for the system's linker.
- **Executable files**: Code sections for loading and execution.

The file's content and structure depend on its creator and intended purpose.

14.1.1 File Attributes

 File Naming:

- Files have human-readable names (e.g., example.c).
- Some systems differentiate between uppercase and lowercase characters in names.

- File names make files independent of the process, user, or system that created them and allow easy transfer or modification.

 File Attributes:

1. **Name**: Human-readable symbolic name.
2. **Identifier**: Unique system-level tag for identifying the file.
3. **Type**: Specifies the file type (if supported).
4. **Location**: Pointer to the file's location on the storage device.
5. **Size**: Current and possibly maximum file size.
6. **Protection**: Access controls for read, write, and execute permissions.
7. **Time, Date, User**: Metadata for creation, modification, and last use, useful for security and monitoring.

 Directories:

- Store information about files, including names and unique identifiers.
- Directory size can grow significantly in systems with many files.
- Directories are stored on secondary storage and loaded into memory as needed.

14.1.2 File Operations

A file is an abstract data type defined by the operations it supports. Operating systems provide system calls for six basic file operations:

1. **Create**: Allocates space and adds an entry in the directory.
2. **Write**: Locates the file, maintains a write pointer, and updates it after writing.
3. **Read**: Locates the file, uses a read pointer to fetch data, and updates the pointer after reading.
4. **Reposition (Seek)**: Adjusts the file position pointer without performing I/O.
5. **Delete**: Removes the directory entry and frees up file space.
6. **Truncate**: Clears file contents while retaining its attributes.

The six basic file operations—create, write, read, reposition, delete, and truncate—are essential. Additional operations include appending data, renaming files, copying files, and modifying file attributes (e.g., file length, owner).

To reduce directory searches during file operations, systems use an **open()** system call, which creates an entry in an **open-file table**. This table stores file metadata like pointers and access modes, avoiding repetitive lookups.

File operations require **open()** before access and **close()** afterward to remove entries. Opened files are managed via a **per-process table** (tracking file pointers and access rights) and a **system-wide table** (containing file metadata like disk location

and size). The system tracks the number of processes accessing a file via an open count, removing the table entry when it reaches zero.

An open file is associated with several key pieces of information:

- **File Pointer**: Tracks the current read/write position, unique to each process.
- **File-Open Count**: Tracks the number of processes with the file open. The entry is removed once the count reaches zero.
- **Disk Location**: Stores the file's disk location in memory to avoid repeated disk lookups.
- **Access Rights**: Defines the access mode (e.g., read, write) for each process, used to manage permissions.

Some systems also support **file locks**, which prevent concurrent access to a file or its sections by multiple processes, useful for shared files like system logs.

File locks function similarly to reader-writer locks. A **shared lock** allows multiple processes to access a file concurrently, like a reader lock, while an **exclusive lock** restricts access to one process at a time, like a writer lock. Some systems only support exclusive locks.

File locking mechanisms can be **mandatory** or **advisory**:

- **Mandatory locking** prevents other processes from accessing a locked file, ensuring locking integrity.
- **Advisory locking** requires processes to manually acquire locks before accessing the file, relying on software developers to manage the lock.

Windows typically uses mandatory locks, while UNIX systems use advisory locks. Care must be taken to avoid **deadlocks** and to ensure exclusive locks are held only while needed to avoid blocking other processes.

14.1.3 File Types

When designing a file system, file types can help the OS handle files appropriately, such as preventing errors like printing binary files. File types are often implemented using extensions (e.g., .doc, .exe), which indicate the file's purpose and operations allowed. For instance, .com and .exe are executable, while .bat is a batch file.

Applications and operating systems use extensions as "hints." For example, assemblers expect .asm, and word processors use .doc. Some systems, like TOPS-20, ensure updated object files by recompiling if the source file is modified. Mac OS X supports file types (e.g., TEXT, APPL) and tracks the creator program, enabling seamless file handling.

UNIX uses "magic numbers" in some files to identify types and allows optional extensions for user convenience. However, UNIX does not enforce extensions or track creating programs, leaving file type management to applications and users.

14.1.4 File Structure

File types indicate a file's internal structure, helping the OS and applications manage files appropriately. For example, executable files must have a specific structure to load into memory and identify the starting instruction. Some OSs, like VMS, support predefined file structures with specialized operations, but this increases system complexity and size.

A minimal file structure approach, used by UNIX and MS-DOS, treats files as sequences of bytes, offering flexibility but requiring applications to interpret file structures. However, all OSs must support at least an executable file structure to run programs.

Macintosh OS divides files into a **resource fork** (e.g., user-editable UI labels) and a **data fork** (e.g., program code). This simplifies customization compared to UNIX or MS-DOS, where changes may require source code modification and recompilation. Balancing the number of file structures is key—too few limit functionality, while too many increase system complexity and confusion.

14.1.5 Internal File Structure

Locating offsets in a file involves managing logical records (user-defined units) and physical blocks (disk sectors). Disk I/O operates in fixed-size blocks, typically 512 bytes, which may not align with logical record sizes. Logical records are packed into physical blocks by either the OS or the application.

For instance, UNIX treats files as byte streams, with each byte addressable by its offset. The OS handles packing and unpacking bytes into blocks. The packing method determines how logical records fit into blocks, with I/O operations performed at the block level.

Disk space allocation in blocks leads to **internal fragmentation**, where unused space in the last block of a file is wasted. For example, a 1,949-byte file would occupy 2,048 bytes across 4 blocks, wasting 99 bytes. Larger block sizes increase internal fragmentation.

14.2 ACCESS METHODS

Files store information that must be accessed and read into memory for use. Some systems offer only one access method, while others, like IBM systems, support

multiple methods. Selecting the appropriate access method is crucial for efficient application design.

14.2.1 Sequential Access

Sequential access is the simplest and most common file access method, where information is processed in order, record by record. Operations like **read next** and **write next** automatically advance the file pointer. Files can be reset to the beginning, and limited skipping of records may be supported. This method, based on a tape model, works for both sequential and random-access devices.

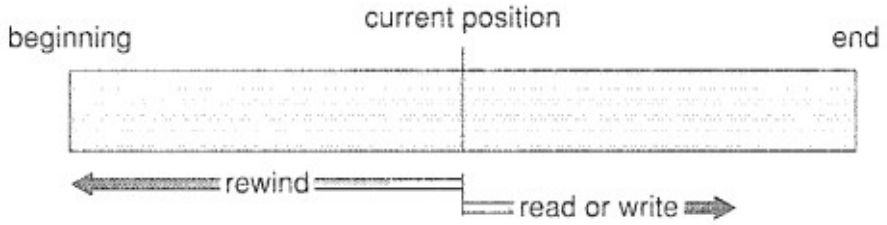


Figure 14.1 Sequential-access file

14.2.2 Direct Access

Direct access (or relative access) enables rapid read and write operations in any order, based on a disk model. Files consist of fixed-length logical records, treated as numbered blocks. Operations like **read n** and **write n** use block numbers, or a **position to n** operation can be used before sequential reads/writes. Relative block numbers, starting at 0 or 1, simplify allocation and prevent unauthorized access to the file system.

Direct access is ideal for large datasets, such as databases or airline reservation systems, where specific records can be directly retrieved. For example, flight 713 data may reside in block 713. Logical record lengths allow efficient record read, write, or deletion. While some systems support both sequential and direct access, others restrict files to one mode, defined at creation. Sequential access can be simulated on direct-access files, but simulating direct access on sequential files is inefficient.

14.2.3 Other Access Methods

Indexing methods build on direct access by creating an index that points to file blocks, enabling efficient record retrieval. For example, in a retail-price file, the index contains the first UPC of each block, allowing binary search to locate the desired block. If the file has 120,000 records (2,000 blocks), the index (20,000 bytes) can be kept in memory, reducing I/O operations.

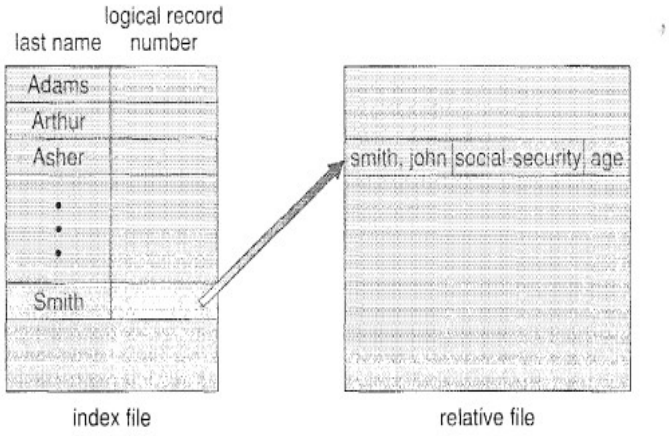


Figure 14.2 Example of index and relative files.

For larger files, a multi-level index can be used. A primary index points to secondary index blocks, which then reference the actual file blocks. Systems like IBM's ISAM use a master index to locate secondary index blocks, which then guide access to the file blocks. This structure enables efficient record retrieval with minimal direct-access reads, as shown in systems like VMS.

14.3 DIRECTORY STRUCTURE

Modern systems can have multiple file systems of different types, such as UFS, VFS, and NFS in Solaris. With millions of files stored across terabytes of disk, organizing this data is crucial. Directories play a key role in managing and structuring files effectively. Before exploring directory structures, it's essential to understand the basics of storage organization.

14.3.1 Storage Structure

A disk can be used entirely for a file system or divided into partitions (also called slices or minidisks) for multiple file systems, swap space, or raw storage. File systems can also be created on larger structures like volumes, which combine partitions.

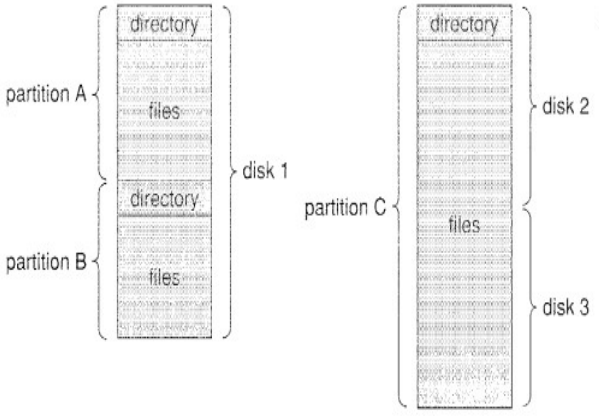


Figure 14.3 A typical file-system organization.

Each volume, acting as a virtual disk, can store multiple operating systems and must include a directory (or volume table of contents) to store file details like name, location, size, and type. Figure 14.3 shows a typical file-system organization.

14.3.2 Directory Overview

A directory acts as a symbol table, mapping file names to their entries. It supports key operations such as:

- **Search for a file**: Locate files by name or pattern.
- **Create a file**: Add new files to the directory.
- **Delete a file**: Remove unused files.
- **List a directory**: Display all files and their details.
- **Rename a file**: Change a file's name or position in the directory.
- **Traverse the file system**: Access all directories and files for backups or freeing disk space by archiving unused files.

The directory's logical structure can be organized in various ways to support these operations efficiently. Common directory structures are discussed in the following sections.

14.3.3 Single-Level Directory

A single-level directory is simple, storing all files in the same directory. While easy to manage initially, it has significant limitations as the number of files or users increases. All files must have unique names, which can be challenging, especially with short name limits (e.g., MS-DOS allows 11 characters, UNIX allows 255). For multiple users or a single user managing hundreds of files, remembering and organizing unique names becomes increasingly difficult, making this structure impractical for complex systems.

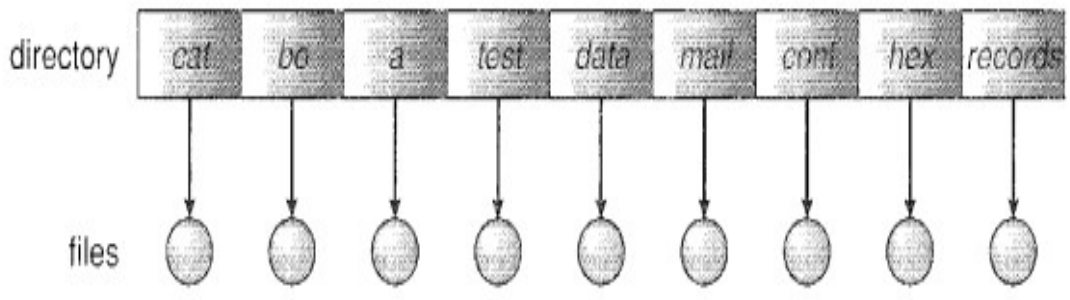


Figure 14.4 Single-level directory.

14.3.4 Two-Level Directory

The two-level directory structure solves file name conflicts by providing each user with their own User File Directory (UFD), managed under a Master File Directory (MFD). Each UFD stores a user's files, allowing different users to have files with the same name. To access another user's file, a path specifying the user and file name (e.g., /user_b/test) is required. This structure resembles a two-level tree, with the MFD at the root, UFDs as branches, and files as leaves.

However, it isolates users, making file sharing difficult unless naming conventions or access permissions are implemented. System files, shared among all users, are stored in a special directory. The operating system first searches the user's UFD and, if not found, searches the system directory, using a search path. This approach minimizes storage duplication and supports flexible file naming, as seen in UNIX and MS-DOS.

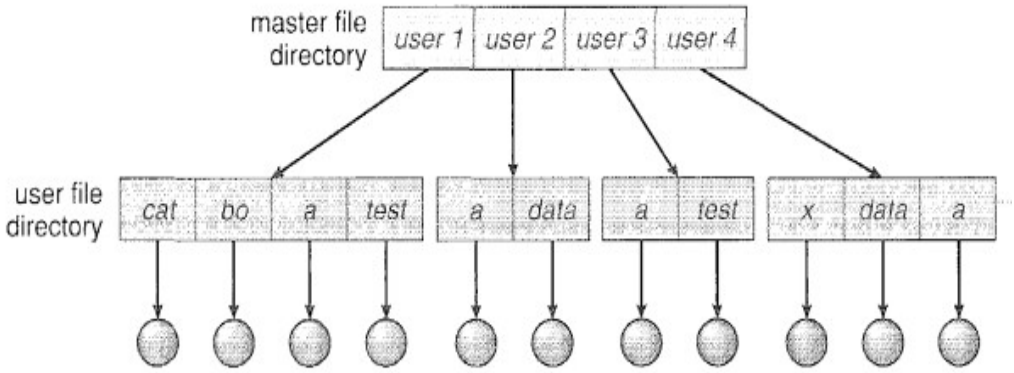


Figure 14.5 Two-level directory structure.

14.3.5 Tree-Structured Directories

A **tree-structured directory** extends the two-level directory into a hierarchy of arbitrary depth, allowing users to create subdirectories and organize files more flexibly. It has a **root directory**, with unique path names for each file. Directories can contain files or subdirectories, and each directory entry specifies whether it is a file or subdirectory.

Each process operates within a **current directory**, which can be changed using system calls. File paths can be **absolute** (starting from the root) or **relative** (from the current directory). Users can structure their files into separate directories based on topics or file types.

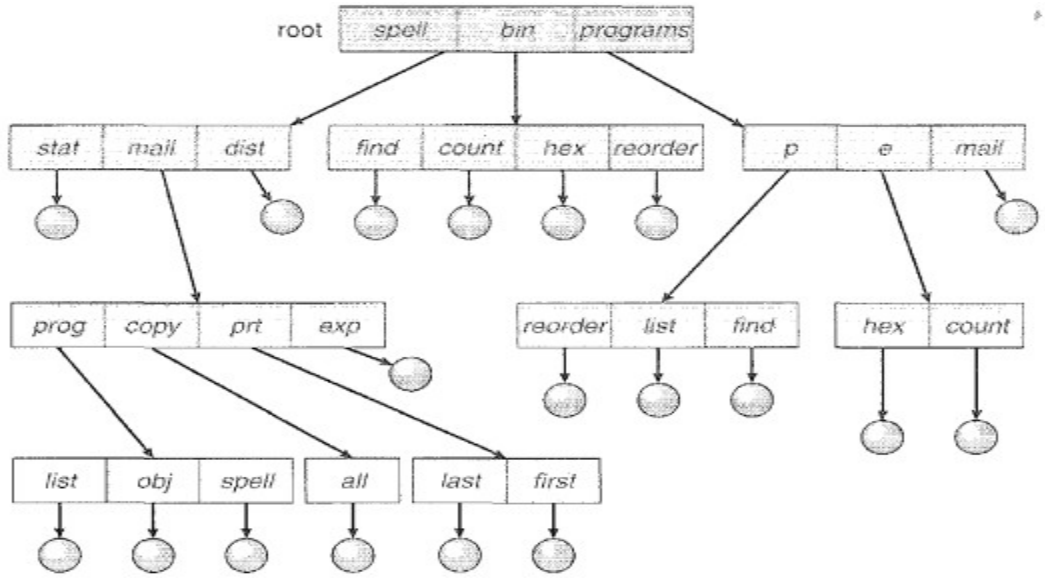


Figure 14.6 Tree-structured directory structure.

When deleting a directory, systems either require it to be empty (e.g., MS-DOS) or allow recursive deletion of all files and subdirectories (e.g., UNIX), with the latter being more convenient but riskier.

Tree structures support user collaboration by enabling access to others' files via specified paths. However, longer paths can complicate program execution. Systems like **Macintosh** simplify this with a **Desktop File** that tracks executable programs, enabling easy execution with a double-click. **Windows** uses an extended two-level directory structure with drive letters for devices and volumes.

14.3.6 Acyclic-Graph Directories

When two programmers collaborate, they may require shared subdirectories to organize files. A **tree-structured directory** does not support sharing, but an **acyclic graph structure** allows files and subdirectories to exist in multiple locations, ensuring changes are immediately visible to all users.

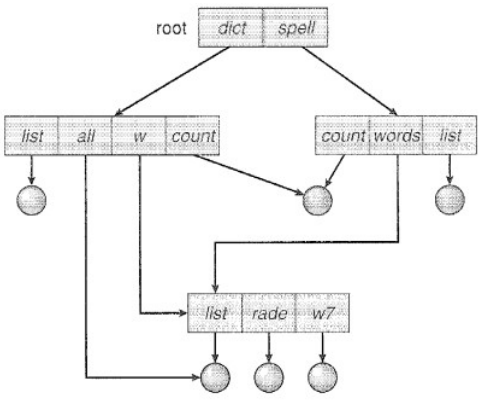


Figure 14.7 Acyclic-graph directory structure

Sharing methods:

1. **Links**: Used in UNIX systems, links are pointers to actual files or directories. They preserve the acyclic structure and simplify deletion since removing a link does not affect the original file.
2. **Duplicate Entries**: Copies the file information into multiple directories, making both entries identical. However, maintaining consistency during modifications is challenging.

Challenges:

- **Multiple Path Names**: A single file may have different absolute paths, complicating operations like system traversal or backups.
- **Deletion Issues**: Removing a file could leave dangling pointers. Symbolic links mitigate this by leaving broken links upon file deletion, but they must be handled carefully.
- **Reference Counting**: UNIX uses a reference count to manage file deletion. A file is deleted only when all references to it are removed, ensuring no dangling links remain.

To avoid these complexities, systems like MS-DOS enforce a tree structure without shared directories or links.

14.3.7 General Graph Directory

A major challenge with an **acyclic graph directory structure** is preventing cycles. Adding links to a tree-structured directory transforms it into a graph, potentially introducing cycles.

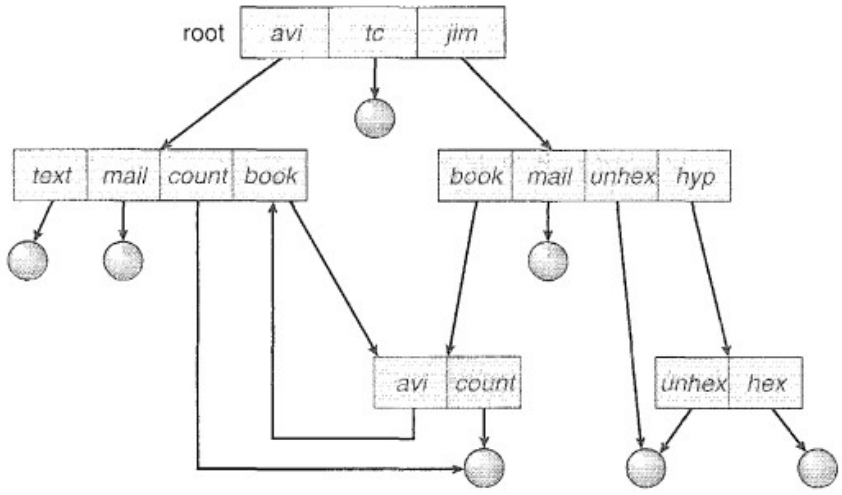


Figure 14.8 General graph directory

Advantages of Acyclic Graphs:

- Simple algorithms for traversal and deletion.
- Avoids redundant searches in shared subdirectories, improving performance.

Problems with Cycles:

- Searching may lead to infinite loops, affecting correctness and performance.
- Deletion issues arise as reference counts may not reach zero due to self-referencing cycles.

Solutions:

1. **Garbage Collection**: Identifies unreferenced files by marking accessible files and collecting unmarked ones. However, this is time-consuming and rarely used.
2. **Cycle Detection**: Algorithms exist to detect cycles but are computationally expensive, especially with disk-based graphs.
3. **Bypassing Links**: During traversal, skipping links avoids cycles and requires no extra overhead, making acyclic structures easier to manage.

14.4 FILE-SYSTEM MOUNTING

Before using a file system, it must be **mounted** to integrate it into the system's directory structure.

Mounting Process:

1. The OS is provided with the device name and a **mount point** (an empty directory in the existing structure).
2. The OS verifies the device has a valid file system by reading and checking its directory structure.
3. The file system is attached at the mount point, allowing seamless traversal across multiple file systems.

Example: A user's home directory file system could be mounted at /home, making paths like /home/jane accessible. If mounted at /users, the same directory becomes /users/jane.

Key Considerations:

- Mounting can obscure existing files at the mount point until unmounted.
- Systems may limit or allow multiple mounts of the same file system at different points.

This approach supports flexible file system integration while maintaining directory structure integrity.

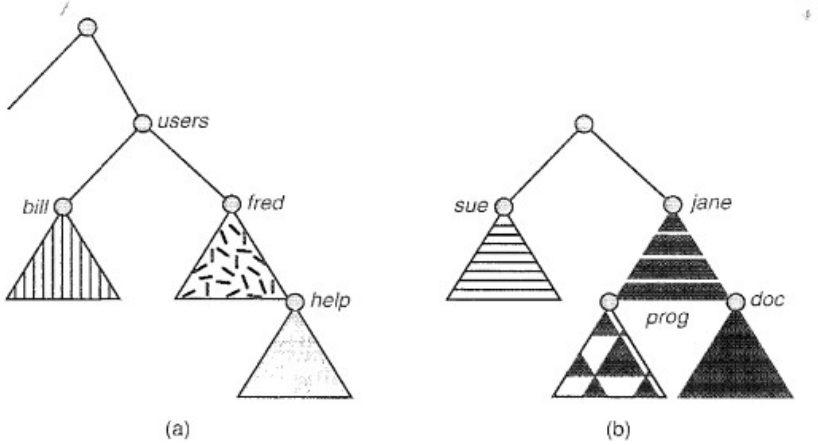


Figure 14.9 File system. (a) Existing system. (b) Unmounted volume.

When the Macintosh OS encounters a disk for the first time (e.g., a hard disk at boot or a floppy disk when inserted), it searches for a file system on the device. If found, the system automatically mounts it at the root level and displays a folder icon labeled with the file system's name. The user can then click the icon to access the newly mounted file system.

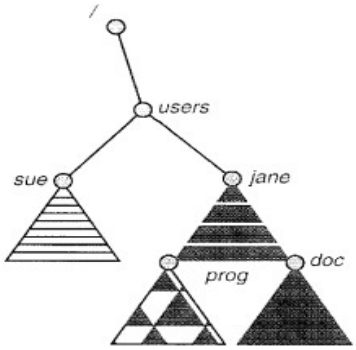


Figure 14.10 Mount point.

Microsoft Windows operating systems (95, 98, NT, 2000, XP) use a two-level directory structure with devices and volumes assigned drive letters. The file path format is drive-letter:\path\file. Recent versions of Windows allow file systems to be mounted anywhere in the directory tree, similar to UNIX. Windows automatically discovers and mounts all devices and file systems at boot time, while additional mounts can be done manually or through a configuration file for automatic mounting at boot.

14.5 FILE SHARING

This section explores the challenges of file sharing in user-oriented operating systems, which is essential for collaboration and reducing effort. It discusses issues arising when multiple users share files, extending sharing across multiple file systems, including remote ones, and managing conflicting actions on shared files. Specifically, it addresses whether all writes to a shared file should be allowed or if the operating system should protect users from interfering with each other's actions.

14.5.1 Multiple Users

When an operating system supports multiple users, file sharing, naming, and protection become crucial. The system must manage file access, either by default or through user-granted permissions. Most systems use file ownership and groups to control access. The owner has the most control, while group members can access a subset of operations. Permissions are defined by the owner and stored with the file attributes. When a user requests an operation, the system checks the user's ID against the owner and group IDs to determine which permissions apply. This process is straightforward in systems with multiple local file systems once they are mounted.

14.5.2 Remote File Systems

With the rise of networking, sharing resources like files across computers became possible. Initially, files were manually transferred using programs like FTP. Later, Distributed File Systems (DFS) allowed remote directories to appear local. The World Wide Web reverts to the FTP model, requiring a browser to access remote files, using operations similar to FTP for file transfer. FTP supports both anonymous and authenticated access, with anonymous access allowing file transfers without an account. DFS offers tighter integration between accessing and providing machines, adding complexity to the system.

14.5.2.1 The Client-Server Model

Remote file systems allow a client machine to mount and access files from a server machine. The server specifies which files and which clients can access them, with client identification typically based on network identifiers or IP addresses. However, spoofing can lead to unauthorized access, so secure authentication methods using encrypted keys are often employed, though they present compatibility and security challenges. In UNIX with NFS, authentication is based on matching user IDs between the client and server. Once mounted, file operations are performed via DFS protocols, with access checks handled by the server before granting or denying access. Afterward, the client can read, write, and close the file.

14.5.2.2 Distributed Information Systems

Distributed information systems, such as the Domain Name System (DNS), simplify client-server management by providing unified access to remote resources. DNS translates host names to network addresses, making Internet navigation scalable. Other systems, like UNIX's NIS and NIS+, manage user and group information, but NIS uses unsecure methods, while NIS+ offers better security. Microsoft's CIFS uses user authentication to manage file access, requiring matching usernames across machines. Modern systems, including Active Directory (based on LDAP), provide secure and centralized user authentication. LDAP is increasingly adopted as a secure method for managing user and resource data, offering benefits like single sign-on and easier system administration across multiple machines in an organization.

14.5.2.3 Failure Modes

Local file systems can fail due to disk issues, hardware failures, or user errors, often requiring human intervention to fix. Remote file systems (OFS) are more prone to failures due to network issues, such as hardware failure, poor configuration, or interruptions between hosts. When a network or server failure occurs, the client system must handle the disruption either by terminating or delaying operations, with the hope that the remote server becomes reachable again.

To recover from failures, state information is maintained on both the client and server. For example, NFS uses a stateless approach, where it assumes a file request is legitimate if the file system is mounted and the file was previously open. This makes NFS resilient but insecure. To address this, NFS version 4 introduces a stateful design, improving security, performance, and functionality by tracking client requests and enforcing proper authentication and permission checks.

14.5.3 Consistency Semantics

Consistency semantics define how shared files are accessed by multiple users, particularly when modifications by one user become visible to others. These semantics are important for file systems supporting file sharing and are related to process-synchronization algorithms. However, due to high latencies in disk and network operations, complex synchronization algorithms (like those in Chapter 6) are generally not implemented for file I/O.

The Andrew File System is a notable example of successful implementation of complex sharing semantics. In file systems, file accesses by a user, which occur between open() and close() operations, are considered a file session. Various consistency semantics are used to manage how these accesses are handled and synchronized across users.

14.5.3.1 UNIX Semantics

The UNIX file system uses the following consistency semantics:

- Writes to an open file by one user are immediately visible to other users with the file open.
- One sharing mode allows users to share the file's current pointer location, so advancing the pointer by one user affects all users sharing the file.

In UNIX, a file has a single physical image that is accessed exclusively, and contention for this image can cause delays in user processes.

14.5.3.2 Session Semantics

The Andrew File System (AFS) uses the following consistency semantics:

- Writes to an open file by one user are not immediately visible to others with the same file open.
- Changes are visible only in new sessions after the file is closed; open instances do not reflect these changes.

This allows multiple users to concurrently read and write to different images of the file without delay, with minimal access scheduling constraints.

14.5.3.3 Immutable-Shared-Files Semantics

Immutable shared files cannot be modified once declared shared by their creator. They have two key properties: their name cannot be reused, and their contents cannot be altered. This read-only sharing simplifies implementation in a distributed system.

14.6 PROTECTION

To ensure information safety, systems focus on reliability and protection. Reliability is achieved by creating duplicate copies of files, often using regular backups to prevent data loss from hardware failures, power issues, or software bugs. Protection involves securing access to files, which can be as simple as physical security in single-user systems or require more complex mechanisms in multiuser systems.

14.6.1 Types of Access

File protection is necessary to control access to files, balancing between no access and unrestricted access. Protection mechanisms limit the types of operations that can be performed on files,

- Read: Read from the file.
- Write: Write or rewrite the file.
- Execute: Load the file into memory and execute it.

- Append: Write new information at the end of the file.
- Delete: Delete the file and free its space for possible reuse.
- List: List the name and attributes of the file

Higher-level operations like renaming or copying may also be controlled, often implemented through lower-level system calls. Protection mechanisms vary based on system size and application, with different needs for small research systems versus large corporate systems.

14.6.2 Access Control

The most common method for file protection is to base access on user identity, implemented via access-control lists (ACLs). An ACL associates each file or directory with user permissions, specifying who can perform certain actions. When access is requested, the system checks the ACL to determine whether access should be granted. While this allows for detailed access control, maintaining lengthy ACLs can be tedious, especially for systems with many users, and it complicates storage management.

To simplify, many systems use a condensed scheme with three user classifications: **Owner**, **Group**, and **Universe**. This allows default permissions for these groups but supports detailed ACLs for specific files or directories. For example, Solaris combines the simplified owner-group-universe model with ACLs for granular control.

Example Scenario:

Sara, writing a book, needs specific permissions:

- **Sara (Owner):** Full access to the file book.
- **Jim, Dawn, Jill (Group):** Read and write access, but not delete.
- **Others (Universe):** Read-only access for feedback.

A group named text can be created for Jim, Dawn, and Jill, and associated with the file. If Sara wants to grant a visitor access to only Chapter 1, ACL functionality allows fine-grained access without altering group memberships.

In UNIX, only administrators can manage groups, while in VMS, file owners control ACLs. Protection schemes often use fields like rwx (read, write, execute) for Owner, Group, and Universe. For example, book permissions might look like:

- **Owner (Sara):**rwx
- **Group (text):**rw-
- **Universe:**r--

Combining ACLs with traditional schemes offers flexibility but requires clear user interfaces. For instance, Solaris appends a "+" to indicate ACL extensions in file permissions.

In systems like Solaris, commands such as setfacl and getfacl are used to manage ACLs. On Windows XP, ACLs are managed via the GUI, as shown in the NTFS file permission interface.

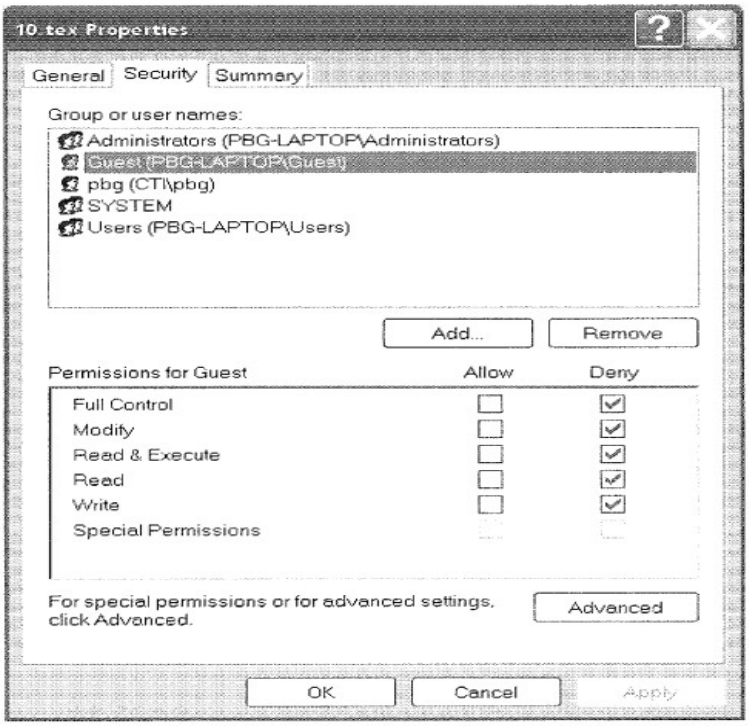


Figure14.11 Windows XP access-control list management.

A key challenge is resolving conflicts between general permissions and ACLs. For instance, if a user belongs to a group with only read permissions but an ACL grants them read and write access, which takes precedence? Solaris prioritizes ACLs due to their finer granularity, following the rule that more specific permissions override broader settings.

14.6.3 Other Protection Approaches

Another approach to file protection involves associating passwords with files. While effective if passwords are random and frequently changed, this method has drawbacks: users may struggle to remember multiple passwords, and a single password for all files poses an all-or-nothing risk. Some systems, like TOPS-20, address this by assigning passwords to directories instead of individual files, while IBM VM/CMS allows separate passwords for read, write, and multiwrite access.

Older single-user operating systems like MS-DOS and pre-Mac OS X offered minimal file protection. Adding protection mechanisms to such systems, especially in networked environments, is less effective and harder than designing them into new systems.

In multilevel directory structures, directory protection is critical. Control must be provided over creating, deleting, and listing files in directories, as file existence and names can carry significance. Access to a file also requires access to its directory. In systems with multiple path names for files, user access rights can vary based on the path used.

14.7 SUMMARY

A file, managed by the operating system, is a sequence of logical records that map to physical storage devices. Directories help organize files, with structures like single-level (prone to naming conflicts), two-level (user-specific directories), tree-structured (subdirectories), and graph-based (enabling file sharing but complicating management).

Disks are divided into volumes containing file systems that can be mounted for access or unmounted for maintenance. File sharing depends on system-defined semantics, such as limits on readers and writers, and distributed systems face challenges like reliability and security.

File protection is essential and can be enforced through passwords, access lists, or other mechanisms, controlling operations like reading, writing, and executing.

14.8 KEY TERMS

File Concept, Access Methods, Directory Structure, File System Mounting, File Sharing, File Protection, Sequential Access, Random Access, Hierarchical Directory, File Permissions, File Access Control.

14.9 SELF-ASSESSMENT QUESTIONS

1. What is the file concept, and why is it essential in operating systems?
2. Explain different access methods for files in a file system.
3. Describe the types of directory structures used in operating systems.
4. What is file system mounting, and how is it performed?
5. Discuss the significance of file sharing and its challenges.
6. How does file protection ensure security in a file system?

14.10 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. "Operating System Principles", Seventh Edition, Wiley.
2. William Stallings, "Operating Systems – Internals and Design Principles", Fifth Edition, Pearson Education (2007)
3. Achyut S Godbole, "Operating Systems", Second Edition, TMH (2007).
4. Flynn/McHoes, "Operating Systems", Cengage Learning (2008).
5. Deitel&Deitel, "Operating Systems",Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Dr U Surya Kameswari

CHAPTER 15

IMPLEMENTING FILE SYSTEMS

AIM AND OBJECTIVES:

- Explain the implementation of local file systems and directory structures.
- Describe the functionality and operation of remote file systems.
- Discuss block allocation methods in file systems.
- Explain free-block management techniques.
- Analyze the trade-offs in block allocation and free-block management.

STRUCTURE:

15.1 FILE-SYSTEM STRUCTURE

Disks are the primary medium for secondary storage, offering two key advantages:

1. **Rewrite Capability**: Blocks can be read, modified, and rewritten in the same location.
2. **Direct Access**: Any block can be accessed directly, enabling both sequential and random file access. Switching files involves moving the read-write heads and waiting for disk rotation.

To enhance I/O efficiency, data is transferred between memory and disk in blocks, typically 512 bytes each. File systems enable efficient data storage, retrieval, and organization by defining file attributes, operations, and directory structures while mapping logical files to physical storage.

A layered structure simplifies management: the I/O control layer uses device drivers to translate commands into hardware actions, and the basic file system handles block-level data access by numeric disk addresses. This design ensures modular and efficient file handling.

The file-organization module manages logical and physical blocks, translating logical block addresses to physical ones based on file allocation and location. It includes a free-space manager to track and allocate unallocated blocks.

The logical file system handles metadata, directory structures, and file control blocks (FCBs) containing file details like ownership, permissions, and data locations. It also ensures security and protection.

A layered file system minimizes code duplication, allowing shared use of I/O control and basic file-system code across multiple systems. Examples include UNIX's UFS, Windows' FAT/NTFS, Linux's ext2/ext3, and distributed file systems for server-client use.

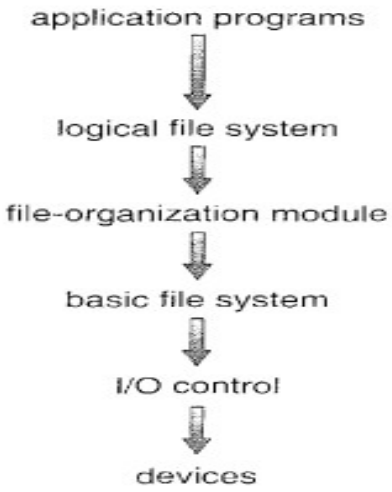


Figure 15.1 Layered file system.

15.2 FILE-SYSTEM IMPLEMENTATION

Operating systems use open() and close() system calls to manage process access to file contents. This section explores the structures and operations behind these file system functions.

15.2.1 Overview

File systems use both on-disk and in-memory structures, varying by operating system. Key on-disk components include:

- **Boot Control Block**: Contains boot-related information, like the boot block in UFS or the partition boot sector in NTFS.
- **Volume Control Block**: Stores volume details such as block size, free blocks, and pointers (e.g., superblock in UFS or master file table in NTFS).
- **Directory Structure**: Organizes files, mapping names to identifiers (inode in UFS, stored in NTFS master file table).
- **File Control Block (FCB)**: Holds file metadata, including permissions, size, and block locations (inode in UFS, row in NTFS master file table).

In-memory structures enhance file-system management and performance via caching. These include:

- **In-memory mount table**: Holds information about mounted volumes.
- **Directory-structure cache**: Stores recently accessed directory information.
- **System-wide open-file table**: Contains FCBs of open files.
- **Per-process open-file table**: Points to the corresponding entry in the system-wide table.

To create a new file, an application calls the logical file system, which allocates a new FCB, updates the directory with the new file name and FCB, and writes it back to disk. Some systems treat directories as files, while others handle them separately. The logical file system uses the file-organization module to map directory I/O to disk blocks.

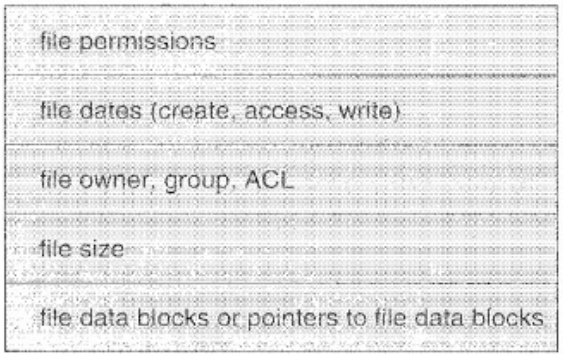


Figure 15.2 A typical file-control block.

When a file is created, it must be opened for I/O operations. The open() system call checks if the file is already in use by searching the system-wide open-file table. If in use, a per-process table entry points to the existing entry. The file's directory structure is then searched (often cached), and the file control block (FCB) is copied into the open-file table, tracking the number of processes accessing it. Each process has an entry in the per-process table, which includes information like the file's current position and access mode. Operations are performed through this table until the file is closed, at which point the entry is removed and metadata is saved to disk.

Systems may also use the file system for other tasks like networking, and caching is crucial for performance. For example, the BSD UNIX system uses caching to improve efficiency, with a high cache hit rate of 85 percent.

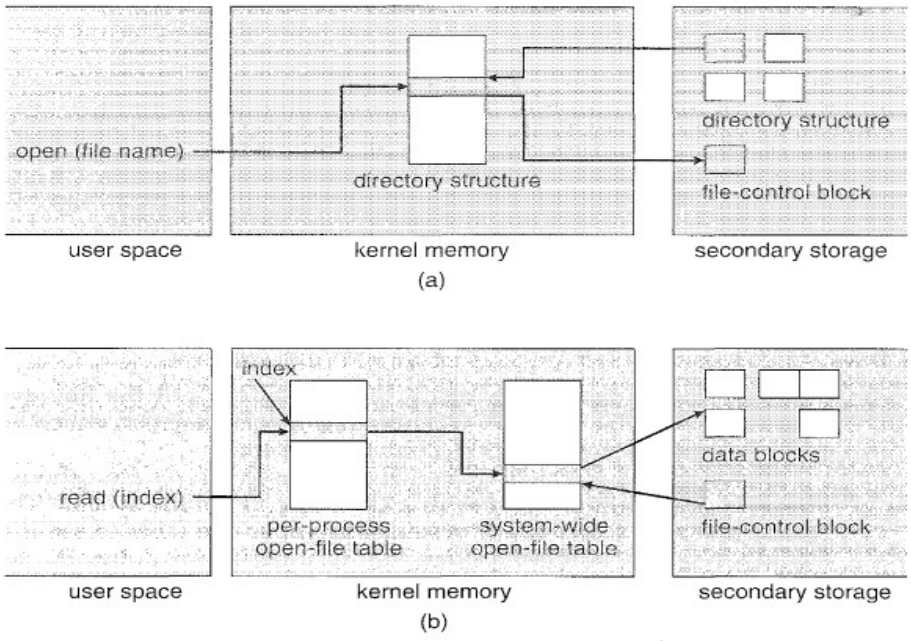


Figure 15.3 In-memory file-system structures. (a) File open. (b) File read.

15.2.2 Partitions and Mounting

Disk layout can vary depending on the operating system. A disk can be divided into multiple partitions, which can either be "raw" (no file system) or "cooked" (with a file system). Raw disks are used where no file system is required, such as in UNIX swap space or some databases, which format the data themselves. Raw disks may also store RAID-related information, like block status or RAID configuration.

Boot information is typically stored in a separate partition in a custom format, as the system cannot interpret file-system formats before booting. This boot partition may include a boot loader that allows the system to choose between multiple operating systems on a disk.

The root partition, containing the operating system kernel, is mounted at boot time, and other volumes can be mounted either automatically or manually. The system verifies the file system format by reading the device directory, checking for validity, and correcting errors if needed. In Windows, each volume is mounted in a separate namespace (e.g., F:), while UNIX mounts file systems at any directory using a mount table and inode flags.

15.2.3 Virtual File Systems

Modern operating systems support multiple file system types by using object-oriented techniques to integrate them into a unified directory structure. Instead of writing separate routines for each file system type, systems like UNIX organize the implementation into three major layers. These layers isolate basic functionality from the details, allowing users to access files across various file systems, including network file systems like NFS. The first layer provides a file-system interface with system calls like open(), read(), write(), and close(), along with file descriptors.

The second layer of the file system architecture is the Virtual File System (VFS), which serves two key purposes:

1. It separates file-system-generic operations from specific implementations, allowing transparent access to multiple file systems.
2. It provides a mechanism for uniquely identifying files across a network using vnodes, which represent network-wide unique files (unlike UNIX inodes, which are only unique within a single file system).

In Linux, the VFS includes four main object types: inode (file), file (open file), superblock (file system), and dentry (directory entry). Each object has a function table with operations like open(), read(), write(), and mmap(). The VFS calls the appropriate function from this table for any object, allowing operations to be performed without knowing the object's specific type.

15.3 DIRECTORY IMPLEMENTATION

The choice of directory-allocation and directory-management algorithms impacts the efficiency, performance, and reliability of the file system. This section discusses the trade-offs involved in selecting these algorithms.

15.3.1 Linear List

A linear list of file names with pointers to data blocks is a simple but slow directory implementation. To add or delete files, the directory must be searched, which can be time-consuming. A directory entry can be marked as unused, linked to a free list, or replaced with the last entry to optimize space. However, finding a file requires a linear

search, leading to inefficiency. To improve this, a software cache can store recent directory information, and a sorted list or tree structure (e.g., a 6-tree) can reduce search time but may complicate file creation and deletion.

15.3.2 Hash Table

A hash table for directory management uses a hash function to quickly locate file entries in a linear list, improving search time. However, it faces challenges like fixed size and collision handling. When the table is full, resizing and rehashing are required, which can be complex. Alternatively, a chained-overflow hash table resolves collisions by linking entries, although lookup speed may decrease due to traversing the linked list. Despite this, a hash table is typically faster than a linear search.

15.4 ALLOCATION METHODS

Disk space allocation for files can be done using three main methods: contiguous, linked, and indexed. Each method has its pros and cons. Some systems support all three, but typically, a system uses one method for all files within a given file system type to ensure efficient space utilization and fast access.

15.4.1 Contiguous Allocation

Contiguous allocation stores files in consecutive blocks on the disk, offering efficient access with minimal disk seeks, as blocks are accessed sequentially or directly. However, it has challenges. First, finding free space for new files can be difficult, requiring efficient management to avoid fragmentation. External fragmentation occurs when free space is scattered, making it hard to allocate large files. Older systems used repacking routines to consolidate free space, but this is time-consuming and inefficient, especially on large disks. Additionally, estimating the required space for a file at creation can be tricky, and allocating too little space can prevent file expansion, particularly with best-fit allocation.

When a file exceeds its allocated space, two options arise: the program can terminate with an error, requiring the user to reallocate space and rerun, which can be costly, or the system can move the file to a larger space, though this can be time-consuming. In some cases, overestimating space needs leads to wasted space, while preallocating too much space can result in internal fragmentation. To address this, modified contiguous allocation uses extents—additional contiguous blocks linked to the original space. While this reduces fragmentation, it can still lead to inefficiencies and external fragmentation. Systems like Veritas use extents for optimized performance.

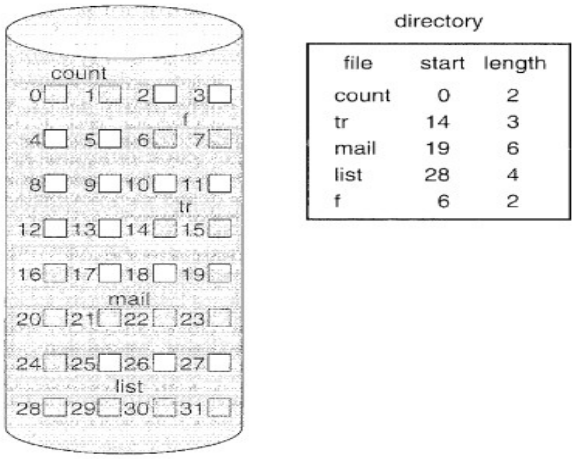


Figure 15.5 Contiguous allocation of disk space.

15.4.2 Linked Allocation

Linked allocation addresses issues of contiguous allocation by organizing files as linked lists of disk blocks, which can be scattered across the disk. Each file's directory entry contains pointers to its first and last blocks. For example, a file might span blocks 9 → 16 → 1 → 10 → 25, with each block linking to the next.

Blocks store 508 bytes of user data, with 4 bytes reserved for the pointer. To create a file, a directory entry is initialized with a nil pointer and size 0. Writing to the file adds a free block, which is linked to the file's end. Reading involves following block pointers sequentially. Linked allocation avoids external fragmentation, supports dynamic file growth, and uses any available free blocks without requiring disk space compaction.

Linked allocation has several drawbacks. It is suitable mainly for sequential-access files since finding the i-th block requires traversing the file from the beginning, causing inefficiency for direct access due to additional disk reads and seeks. Space is also wasted on pointers; for example, a 4-byte pointer per 512-byte block uses 0.78% of disk space.

To address this, clusters (groups of blocks) are used to reduce pointer overhead, improve disk throughput, and simplify mapping. However, this increases internal fragmentation as partially filled clusters waste more space than blocks.

Reliability is another concern, as damaged pointers can corrupt files, link into the free-space list, or connect to the wrong file. Partial solutions include doubly linked lists or storing file metadata in each block, though these add further overhead.

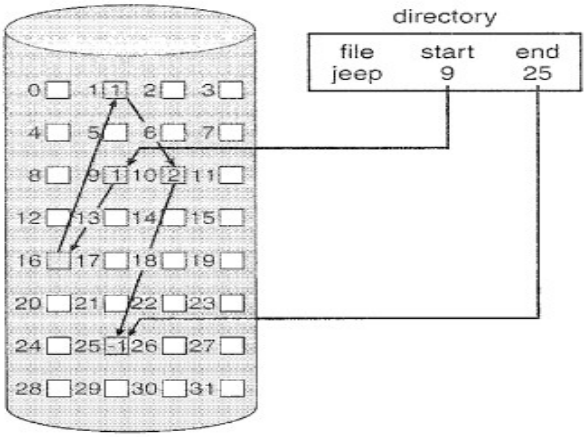


Figure 15.6 Linked allocation of disk space.

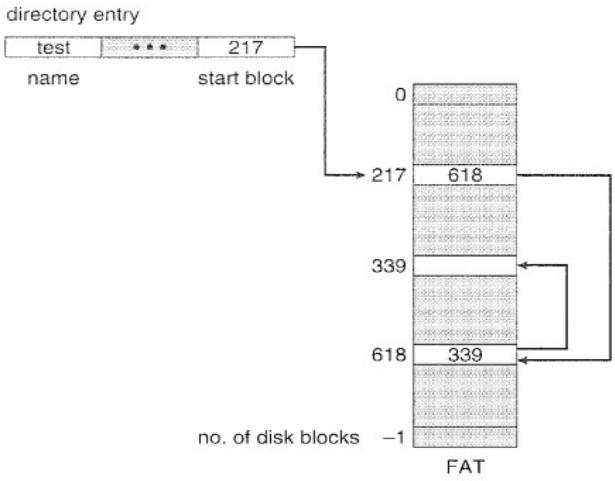


Figure 15.7 File-allocation table

The File Allocation Table (FAT) is a variation of linked allocation used in MS-DOS and OS/2. A FAT, stored at the start of each disk volume, has entries for each disk block indexed by block number. The directory entry points to the first block of a file, and the FAT stores the pointer to the next block, continuing until an end-of-file marker. Unused blocks are marked as 0. Allocating a new block involves updating the FAT with the new block's address and marking it with an end-of-file value.

FAT improves random access as the location of any block can be determined directly from the table. However, it may cause frequent disk head movements unless the FAT is cached, as accessing a block often requires seeking the FAT and then the block.

15.4.3 Indexed Allocation

Indexed allocation addresses the limitations of linked allocation by consolidating all pointers into a single index block. Each file has its own index block, an array of disk-block addresses. The *i-th* entry in the index block points to the *i-th* block of the file.

The directory contains the address of the index block, enabling direct access to any block without sequential traversal.

When a file is created, index block pointers are set to nil. Writing to the *i-th* block allocates a free block and updates the corresponding index entry. Indexed allocation eliminates external fragmentation and allows flexible use of free space. However, it incurs more overhead than linked allocation since even small files require a full index block, resulting in wasted space for files with few blocks.

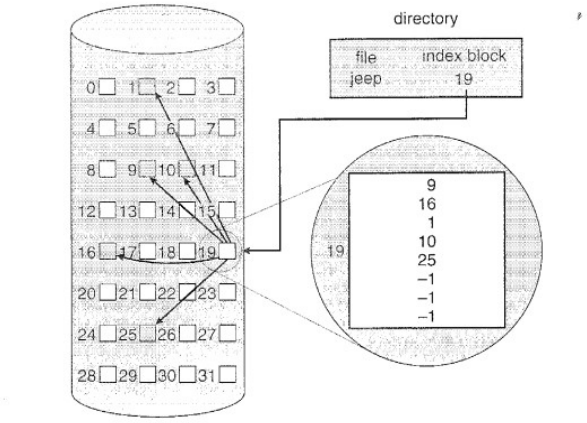


Figure 15.8 Indexed allocation of disk space.

Determining the size of an index block is crucial in indexed allocation. Small index blocks save space but may not hold enough pointers for large files. Solutions include:

1. **Linked Scheme**: Index blocks can link together for large files. Each block contains pointers to data blocks and, if needed, a pointer to the next index block.
2. **Multilevel Index**: Uses hierarchical levels of index blocks. For example, a first-level index points to second-level indexes, which point to data blocks. With 4 KB blocks and 4-byte pointers, a two-level index supports files up to 4 GB.
3. **Combined Scheme**: Used in systems like UFS, where small files (e.g., 12 blocks) use direct pointers in the file's inode. Larger files use single, double, or triple indirect blocks for additional levels of indexing. This method supports very large file sizes, especially with 64-bit pointers.

While indexed allocation avoids external fragmentation, data blocks may still be scattered across the disk, leading to performance issues unless cached.

15.4.4 Performance

Allocation methods differ in storage efficiency and access speed, which depend on the system's access patterns.

- **Contiguous Allocation**: Ideal for direct access, as the disk address of any block can be calculated directly using the starting address. However, it requires the file size to be fixed at creation.
- **Linked Allocation**: Suitable for sequential access, as the next block address is stored in memory. It is inefficient for direct access, as accessing the *i-th* block may require multiple reads.

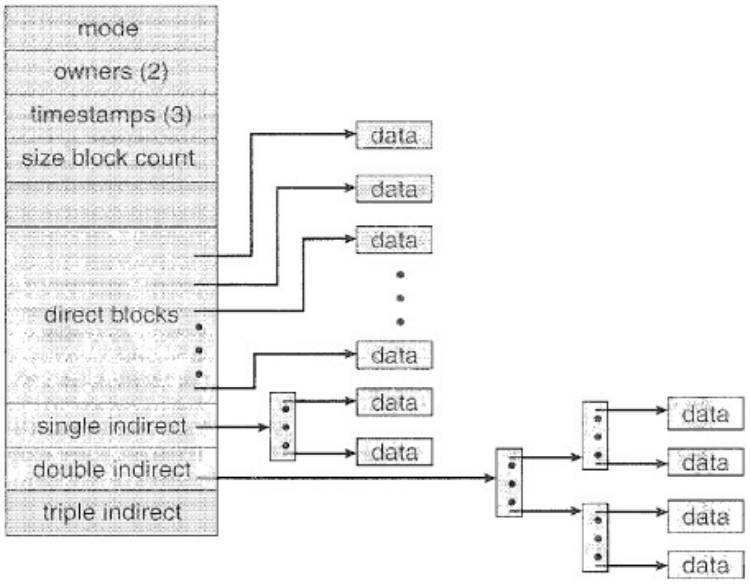


Figure 15.9 The UNIX inode.

Some systems combine methods: sequential files use linked allocation, while direct-access files use contiguous allocation. File type (sequential or direct) must be declared during creation. Files can be converted between types by copying to a new file of the desired type, deleting the old file, and renaming the new one.

Indexed allocation offers direct access if the index block is in memory, but storing it requires significant space. Without sufficient memory, accessing a block might involve reading the index block first, or multiple index blocks for large files with multilevel indexing. Performance depends on the index structure, file size, and block position.

To optimize, some systems use contiguous allocation for small files and switch to indexed allocation as files grow. For example, Sun Microsystems improved UNIX performance by allocating space in 56 KB clusters, reducing fragmentation and improving disk throughput while keeping the inode structure unchanged. Disk-reading routines were optimized, resulting in 25% less CPU usage and better performance. Modern systems often employ thousands of CPU instructions to minimize disk-head movements due to the increasing gap between CPU and disk speeds.

15.5 FREE-SPACE MANAGEMENT

To manage limited disk space, the system uses a free-space list to track unallocated disk blocks. When creating a file, space is allocated from this list and removed. When a file is deleted, its space is returned to the list. The free-space list may not always be implemented as a literal list, depending on the system design.

15.5.1 Bit Vector

In a bit-vector approach, each disk block is represented by a single bit: 1 for free and 0 for allocated. For example, if blocks 2–5 and 8–13 are free, the bit map would look like 001111001111.... This method simplifies locating free blocks, especially with hardware instructions that identify the first 1 bit in a word.

To find the first free block, the system scans for the first non-zero word and calculates the block number based on its offset. While efficient for small disks, this method requires significant memory for large disks. For instance, a 1.3-GB disk with 512-byte blocks needs 332 KB for the bit map, while clustering reduces it to 83 KB. A 40-GB disk with 1-KB blocks needs over 5 MB.

15.5.2 Linked List

In linked free-space management, free blocks are linked together using pointers, with the first free block pointer stored in a special location and cached in memory. Each free block points to the next. For example, block 2 points to block 3, which points to block 4, and so on. This method is simple but inefficient for traversing the list, as it requires reading each block, leading to high I/O overhead. However, since free list traversal is rare, the first block is typically allocated directly. The FAT system integrates free-block management into its allocation structure, eliminating the need for a separate method.

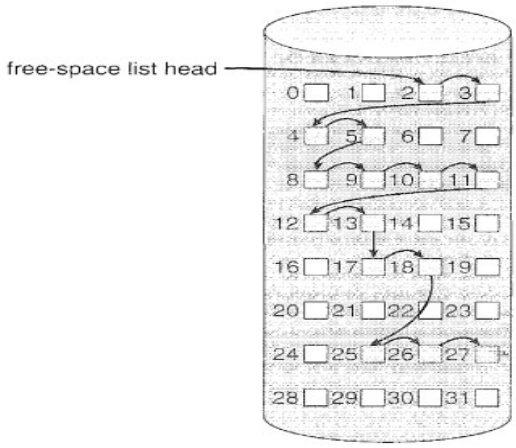


Figure 15.10 Linked free-space list on disk

15.5.3 Grouping

A modified free-list approach stores the addresses of **n free blocks** in the first free block. The first n−1n-1n−1 blocks are free, while the last block points to another group of nnn free blocks. This method enables quick access to a large number of free blocks, improving efficiency compared to the standard linked-list approach.

15.5.4 Counting

An alternative approach is to track contiguous free blocks by storing the address of the first free block and the count of contiguous free blocks. This method reduces the length of the free-space list, as each entry consists of an address and a count, making it more efficient when multiple contiguous blocks are freed or allocated at once.

15.6 EFFICIENCY AND PERFORMANCE

Disks are often a performance bottleneck due to their slower speed compared to other computer components. This section explores various techniques aimed at improving the efficiency and performance of secondary storage.

15.6.1 Efficiency

Efficient disk space usage depends on allocation and directory algorithms. For example, UNIX preallocatesinodes across a volume, improving performance by keeping file data blocks near their inode blocks, reducing seek time. Clustering schemes, like those in BSD UNIX, improve file-seek performance by adjusting cluster sizes to reduce fragmentation. Directory entries, such as "last write" or "last access" dates, require writing back to disk when accessed, which can be inefficient for frequently accessed files. The trade-off between these benefits and performance must be considered when designing a file system.

Efficiency in file systems is affected by pointer sizes used to access data. Systems typically use 16- or 32-bit pointers, limiting file size to 64KB or 4GB. Some systems use 64-bit pointers to support larger files, but they require more storage, increasing space usage for allocation and free-space management. As technology advances, pointer sizes must adapt, as seen with MS-DOS, where larger disk capacities required changes to the file system. Sun’s ZFS uses 128-bit pointers, which are theoretically sufficient forever. Similarly, Sun's Solaris evolved from fixed-length tables to dynamic allocation, allowing more flexibility but at the cost of slower performance.

15.6.2 Performance

File-system performance can be improved with disk controller caches, which store entire tracks in memory to reduce latency. Once data is transferred from the disk

controller, the operating system may further cache it in main memory. Some systems use a dedicated buffer cache for file data, while others use a page cache that stores data as pages using virtual memory techniques, which is more efficient. Many systems, including Solaris, Linux, and Windows, use unified virtual memory to cache both process pages and file data. A unified buffer cache eliminates the need for double caching (copying data between two caches), improving memory use, reducing CPU cycles, and preventing inconsistencies that could corrupt files.

LRU (Least Recently Used) is a common block/page replacement algorithm, but choosing the right algorithm is complex, as shown by Solaris' evolving page-caching methods. Earlier versions used shared memory for processes and page caches, leading to inefficient memory use during high I/O activity. Later versions prioritized process pages over the cache, applied fixed memory limits, and optimized memory use to reduce thrashing.

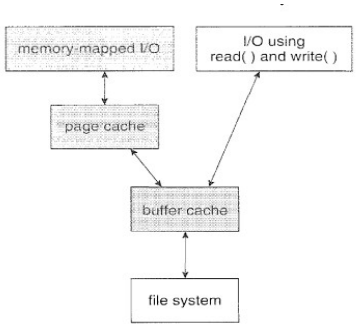


Figure 15.11 I/O without a unified buffer cache.

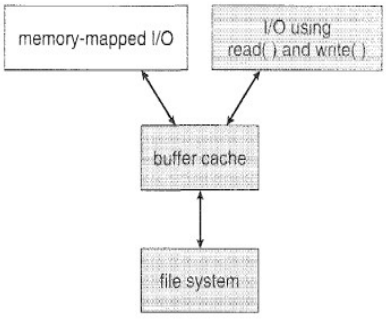


Figure 15.12 I/O using a unified buffer cache.

I/O performance is also influenced by whether writes are synchronous (wait for disk write to complete) or asynchronous (writes cached and returned immediately). Databases often use synchronous writes for transaction consistency. Sequential file access can be optimized with techniques like free-behind (removing

unused pages) and read-ahead (loading subsequent pages), improving I/O speed. Despite track caching, read-ahead remains beneficial for reducing transfer overhead.

The interaction between the page cache, file system, and disk driver optimizes disk writes by minimizing seek time and writing during disk rotation, making asynchronous writes faster than reads in large transfers.

15.7 RECOVERY

Files and directories are stored in both main memory and on disk, requiring precautions to prevent data loss or inconsistency during system failures. These issues are addressed in the following sections.

15.7.1 Consistency Checking

Directory information is stored in both main memory and on disk, with cached data often being more up-to-date than the disk version. A system crash can result in the loss of cache and buffer contents, leaving the file system in an inconsistent state. To address this, a consistency checker (e.g., fsck or chkdsk) is run at reboot to compare and fix discrepancies between the directory structure and data blocks. The effectiveness of the checker depends on the allocation method. For example, linked allocation allows file reconstruction, while indexed allocation can cause issues if a directory entry is lost. UNIX ensures metadata changes, like space allocation, are written synchronously to prevent such inconsistencies.

15.7.2 BackUp and Restore

To protect against data loss from magnetic disk failures, system programs can back up data to other storage devices like floppy disks, tapes, optical disks, or additional hard disks. Recovery involves restoring data from these backups. To optimize this process, the backup program can use directory information, such as the last backup date and the file's last write date, to avoid unnecessary copies. A typical backup schedule includes:

- **Day 1**: Full backup of all files.
- **Day 2**: Incremental backup of files changed since Day 1.
- **Day 3**: Incremental backup of files changed since Day 2.

.

.

.

- **Day N**: Incremental backup of files changed since Day N-1, then restart the cycle from Day 1.

The backup cycle can overwrite previous backups or use new media for each cycle. To restore an entire disk, start with the full backup and follow with the incremental backups. The longer the cycle (higher N), the more backup media are needed for a complete restore. A shorter cycle with full backups followed by daily incremental backups reduces the number of media needed for a restore. A full backup is also periodically taken and stored permanently, away from regular backups, to protect against disasters. Care must be taken not to reuse backup media too often, as they can wear out and become unusable for future restores.

15.8 LOG-STRUCTURED FILE SYSTEMS

Log-based recovery techniques, originally used in database systems, are now applied to file systems to improve consistency and prevent data loss during crashes. These techniques, implemented in journaling file systems, log all metadata changes sequentially before they are applied to disk structures like directories and file control blocks (FCBs). This ensures that even if a system crashes mid-operation, the changes can be replayed from the log to restore consistency. The log operates as a circular buffer, overwriting old data once it's no longer needed. Systems like NIFS, Veritas, and UFS on Solaris use this method to prevent inconsistencies and minimize recovery time, making it a common practice in modern file systems.

In the event of a system crash, any incomplete transactions in the log must be completed to maintain file system consistency. If a transaction was aborted, its changes must be undone. This process ensures the file system remains consistent without the need for time-consuming consistency checks.

Additionally, using log-based updates for metadata improves performance by converting costly random writes into faster sequential writes to the log. These changes are later applied to the file system asynchronously, resulting in better performance for metadata operations like file creation and deletion.

15.9 SUMMARY

The file system is stored on secondary storage, typically disks, which may be partitioned to support multiple file systems. These file systems are integrated into a logical file system structure for easier access. File systems are layered, with lower levels managing physical storage and higher levels dealing with file names and logical properties. A Virtual File System (VFS) interface allows uniform access to different file systems, including remote ones.

Files can be allocated using contiguous, linked, or indexed allocation, each with its pros and cons, such as fragmentation or inefficiency. These methods can be optimized, for example, by using extents for contiguous allocation or clustering for indexed allocation. Free-space allocation and directory management also impact disk efficiency and reliability, with hash tables commonly used for directory management.

Backup tools help protect against data loss by allowing copies to be made for recovery in case of failures.

15.10 KEY TERMS

File System Structure, File System Implementation, Directory Implementation, Allocation Methods, Free Space Management, Efficiency, Performance, Recovery, Log-structured File Systems, File System Optimization

15.11 SELF-ASSESSMENT QUESTIONS

1. What is the structure of a file system, and how is it organized?
2. Explain the steps involved in implementing a file system.
3. How are directories implemented in file systems, and what are the common methods?
4. Describe the different allocation methods for files and their advantages.
5. How is free space managed in file systems to ensure efficient utilization?
6. What is a log-structured file system, and how does it aid in recovery and performance optimization?

15.12 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”, Seventh Edition, Wiley.
2. William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3. Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4. Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5. Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Dr U Surya Kameswari

CHAPTER 16

MASS-STORAGE STRUCTURE

Aim and objectives:

- Explain the physical structure and functionality of secondary storage devices.
- Describe the physical structure and uses of tertiary storage devices.
- Analyze the performance characteristics of mass storage devices.
- Discuss operating system services for RAID systems.
- Explain the role of operating system services in hierarchical storage management (HSM).

Structure:

16.1 OVERVIEW OF MASS-STORAGE STRUCTURE

16.1.1 Magnetic Disks

Magnetic disks are the main form of secondary storage in modern computers. Each disk contains platters, similar in shape to CDs, coated with a magnetic layer for storing data. A read-write head floats above each platter surface, mounted on a disk arm that moves the heads as a unit. Platters are divided into circular tracks and sectors, and the tracks aligned vertically form cylinders, providing high storage capacity in gigabytes.

When in use, the disk spins at high speeds (60-200 rotations per second), enabling data transfer between the disk and the computer. Disk speed depends on transfer rate (data movement speed) and positioning time (seek time for locating the track and rotational latency for sector alignment). Common transfer speeds are in megabytes per second, with positioning delays in milliseconds.

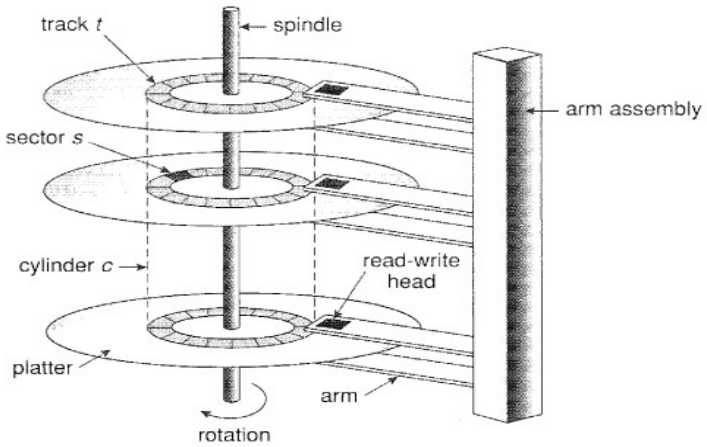


Figure 16.1 Moving-head disk mechanism.

The read-write head floats on a thin air cushion, but if it touches the platter surface, it can cause a head crash, damaging the disk permanently.

Disks can be removable, like floppy disks (1.44 MB) or higher-capacity hard disk alternatives. Disk drives connect to computers via I/O buses like SATA, USB, or SCSI, and each drive has a controller to manage commands from the computer's host controller. Disk controllers often include a cache, optimizing data transfer between the disk and the host.

16.1.2 Magnetic Tapes

Magnetic tape was an early form of secondary storage, known for its large capacity and durability but slow access times compared to memory and magnetic disks. Tapes are mainly used for backups, infrequently accessed data, and data transfer between systems due to their slow random access speed. Tapes are wound on spools, taking time to reach specific data locations, though they write data at speeds similar to disks once positioned. Capacities range from 20 GB to 200 GB, with some tapes supporting compression for higher storage. Tapes come in various widths (e.g., 4 mm, 8 mm) and types like LTO-2 and SDLT.

16.2 DISK STRUCTURE

Modern disk drives organize data as a one-dimensional array of logical blocks, the smallest transfer unit, typically 512 bytes in size. These blocks are mapped sequentially to disk sectors, starting from the outermost cylinder. Logical block numbers can theoretically be converted to disk addresses (cylinder, track, and sector), but this is complicated by defective sectors and varying numbers of sectors per track.

On constant linear velocity (CLV) drives, like CD-ROMs, bit density is uniform, so outer tracks hold more sectors and require faster rotation speeds as the head moves inward. Hard disks use constant angular velocity (CAV), maintaining consistent rotation speed but reducing bit density from inner to outer tracks.

Advances in disk technology have increased sectors per track (outer zones holding up to 40% more than inner zones) and the number of cylinders, with modern disks containing tens of thousands of cylinders.

16.3 DISK ATTACHMENT

Computers access disk storage either through I/O ports (host-attached storage) on small systems or via a remote host in a distributed file system, called network-attached storage.

16.3.1 Host-Attached Storage

Host-attached storage is accessed via local I/O ports using technologies like IDE/ATA, SATA, SCSI, or Fiber Channel (FC). IDE/ATA supports up to two drives per bus, while SATA simplifies cabling. High-end systems often use SCSI or FC.

SCSI is a bus-based system supporting up to 16 devices, including a host controller and storage targets, with logical unit addressing for devices like RAID arrays. FC, a high-speed serial architecture, operates over optical fiber or copper cables. It supports large switched fabrics for storage-area networks (SANs) or arbitrated loops (FC-AL) for up to 126 devices.

Common host-attached storage devices include hard drives, RAID arrays, and optical or tape drives, with data transfers initiated by read/write commands to specified storage units.

16.3.2 Network-Attached Storage

Network-attached storage (NAS) is a remote storage system accessed over a network using protocols like NFS for UNIX or CIFS for Windows. It uses remote procedure calls (RPCs) over TCP/IP, typically on the same LAN as client devices. NAS is often implemented as a RAID array with software supporting the RPC interface, providing shared storage for multiple devices.

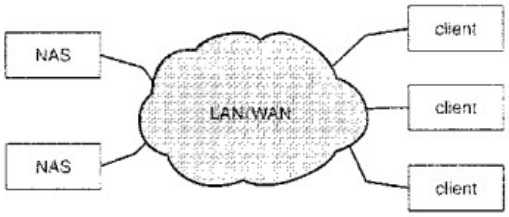


Figure 16.2 Network-attached storage.

NAS simplifies storage sharing but may be less efficient than direct-attached options. The SCSI protocol enhances NAS by carrying SCSI commands over IP networks, allowing hosts to treat distant storage as if it were locally attached.

16.3.3 Storage-Area Network

NAS systems can increase network latency by using data network bandwidth for storage I/O, creating bottlenecks in large client-server setups.

SANs address this by using a private network with storage-specific protocols to connect servers and storage devices. They are highly flexible, allowing multiple hosts and storage arrays to share resources. SAN switches manage access, enabling dynamic storage allocation, such as adding space to a server when needed.

SANs are efficient, supporting shared storage for server clusters with more affordable and numerous ports, typically using Fiber Channel (FC). InfiniBand is a newer high-speed alternative for server and storage interconnection.

16.4 DISK SCHEDULING

The operating system ensures efficient hardware use for disk drives by optimizing **access time** (seek time and rotational latency) and **disk bandwidth** (bytes transferred per unit time).

When a process needs disk I/O, it makes a system call with details like:

- Input or output operation
- Disk address
- Memory address
- Number of sectors

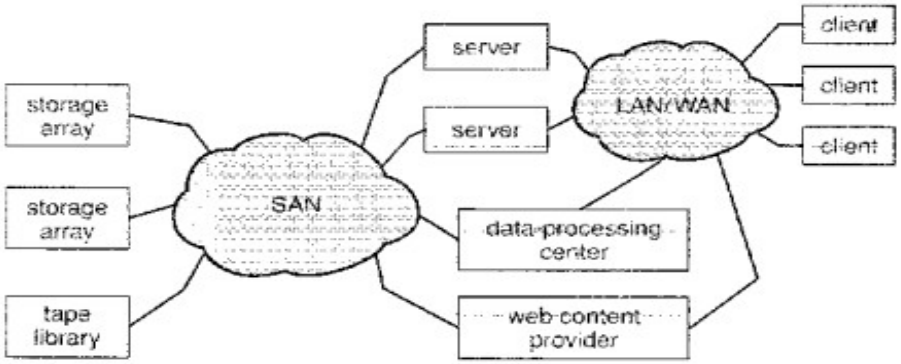


Figure 16.3 Storage-area network.

If the disk drive is available, the request is serviced immediately; otherwise, it's queued. In multiprogramming systems, the OS decides the order to service pending disk requests to improve access time and bandwidth.

16.4.1 FCFS Scheduling

The **First-Come, First-Served (FCFS)** disk scheduling algorithm is simple and fair but often inefficient. For example, servicing disk requests in the order **98, 183, 37, 122, 14, 124, 65, 67** with the head starting at cylinder 53 results in **640 cylinders of head movement**. Inefficient scheduling, such as moving from 122 to 14 and back to 124, highlights the need for optimizing the order to reduce total head movement and improve performance.

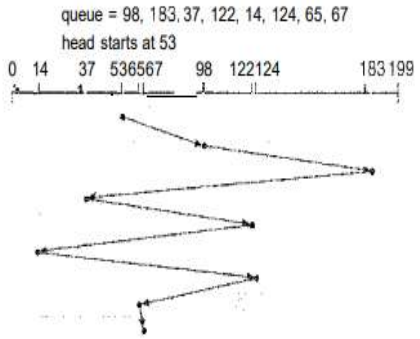


Figure 16.4 FCFS disk scheduling.

16.4.2 SSTF Scheduling

The Shortest-Seek-Time-First (SSTF) algorithm prioritizes servicing the request closest to the current head position, reducing seek time by minimizing cylinder traversal. For example, with an initial head position at cylinder 53, SSTF processes requests in the order: 65, 67, 37, 14, 98, 122, 124, and 183, resulting in a total head movement of 236 cylinders, significantly outperforming FCFS scheduling. However, like Shortest-Job-First (SJF) scheduling, SSTF may cause starvation if new nearby requests continually arrive, potentially delaying distant requests indefinitely.

While SSTF improves over FCFS, it is not optimal. For instance, moving the head from 53 to 37, then 14, before servicing 65, 67, 98, 122, 124, and 183 reduces total head movement to 208 cylinders, outperforming SSTF.

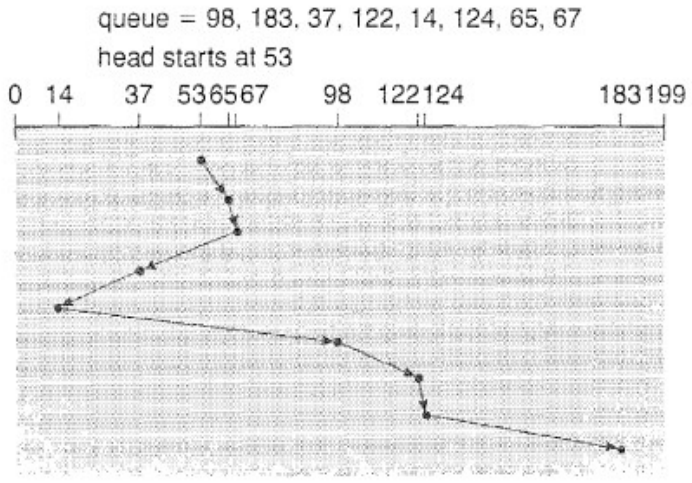


Figure 16.5 SSTF disk scheduling.

16.4.3 SCAN Scheduling

The SCAN algorithm, also known as the elevator algorithm, moves the disk arm in one direction, servicing requests along the way, until it reaches the end of the disk. It then reverses direction and continues servicing requests. For example, starting at cylinder 53 and moving down, SCAN services 37 and 14 before reversing direction to handle 65, 67, 98, 122, 124, and 183. Requests in front of the head are serviced quickly, while those behind must wait until the arm reverses. This approach balances servicing requests across the disk but leads to fewer requests near the reversal points, inspiring alternative algorithms.

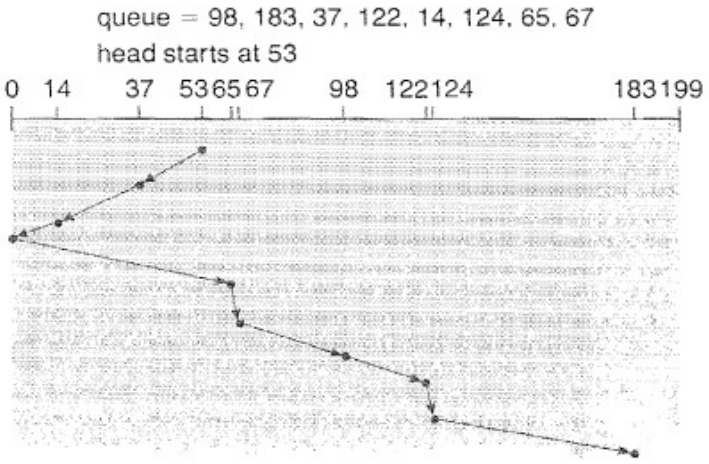


Figure 16.6 SCAN disk scheduling

16.4.4 C-SCAN Scheduling

Circular SCAN (C-SCAN) improves uniform wait times by moving the disk head in one direction, servicing requests along the way. Upon reaching the end, the head returns to the start without servicing requests on the return trip, treating cylinders as a circular list.

16.4.5 LOOK Scheduling

LOOK and C-LOOK are variations of SCAN and C-SCAN that optimize movement by stopping the disk arm at the last request in each direction instead of going to the disk's end. These algorithms "look" for pending requests before continuing, reducing unnecessary movement.

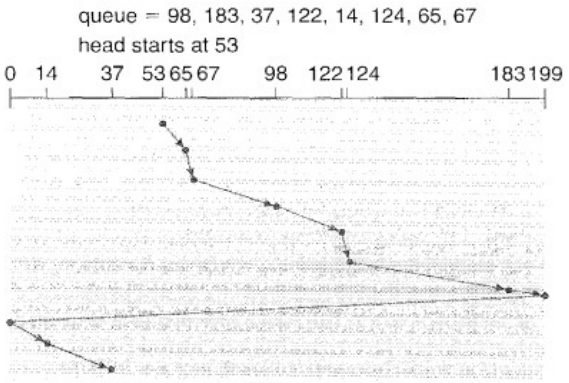


Figure 16.7 C-SCAN disk scheduling

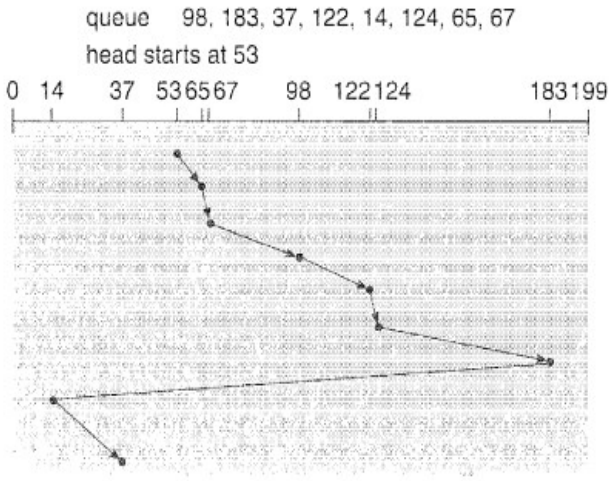


Figure 16.8 C-LOOK disk scheduling.

16.4.6 Selection of a Disk-Scheduling Algorithm

Choosing the best disk-scheduling algorithm depends on the workload and request patterns. SSTF is commonly used for its simplicity and performance, while SCAN and C-SCAN are better for heavy loads as they reduce starvation. Though optimal schedules exist, their computation may not justify the savings over simpler methods like SSTF or SCAN.

Disk scheduling behavior is influenced by file allocation methods. Contiguous files result in minimal head movement, while scattered files increase it. Directory and index block locations are also critical. Placing directories centrally or caching them in memory minimizes disk-arm movement, especially for frequent read operations.

Due to disk scheduling complexities, it's best to implement it as a separate OS module for easy replacement. SSTF or LOOK are good default choices. Modern disks face challenges with rotational latency, as logical block locations are often hidden. Disk controllers now handle scheduling, optimizing both seek time and latency for batches of requests.

However, OS-level scheduling may still be necessary to prioritize tasks like demand paging, urgent writes, or ensuring write order for system robustness. In such cases, the OS may manage scheduling and send requests individually to the disk controller.

16.5 DISK MANAGEMENT

The operating system manages disk-related tasks such as initialization, booting, and bad-block recovery.

16.5.1 Disk Formatting

A new magnetic disk requires low-level formatting to divide it into sectors. Each sector has a header, a data area (usually 512 bytes), and a trailer. The header and trailer include metadata like the sector number and an error-correcting code (ECC). During I/O, the ECC is updated based on the data. When read, the ECC verifies the data's integrity, detecting and correcting minor errors automatically, or reporting soft errors if recoverable.

Most hard disks are low-level formatted at the factory to map logical blocks to defect-free sectors. Formatting determines sector sizes (e.g., 256, 512, or 1,024 bytes), balancing data capacity and overhead. After formatting, disks are partitioned into groups of cylinders, with each partition treated as a separate disk. One partition might hold the OS, while another stores user files.

Logical formatting creates file-system structures like free-space maps (e.g., FAT, inodes) and directories. File systems often group blocks into clusters to optimize I/O for sequential access.

Some systems use raw disks—partitions without file-system structures—for direct, efficient data control, such as in databases. However, regular file-system services are typically better for most applications, providing caching, locking, and space management.

16.5.2 Boot Block

When a computer is powered on or rebooted, it needs an initial program to start. This program, called the bootstrap, initializes the system and loads the operating system kernel into memory. The bootstrap is typically stored in read-only memory (ROM), which is fixed and immune to viruses. However, modifying the bootstrap in ROM is difficult. Therefore, most systems store a small bootstrap loader in ROM, which loads the full bootstrap program from the disk. The full program is stored in the "boot blocks" on the disk, and the disk with this partition is called the boot or system disk.

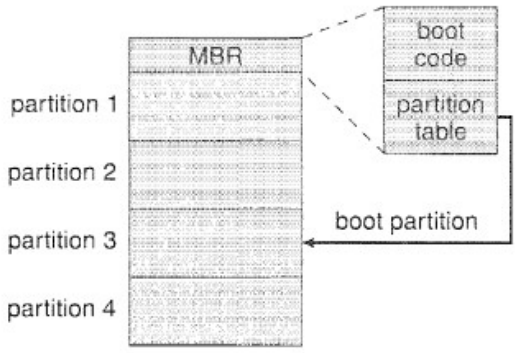


Figure 16.9 Booting from disk in Windows 2000

The boot ROM code instructs the disk controller to read the boot blocks into memory and then execute the code. This more advanced bootstrap program loads the entire operating system from the disk and starts it. In Windows 2000, the boot code is stored in the master boot record (MBR) of the hard disk, which also contains partition information and a boot partition flag. The system starts by running the ROM code, which directs it to read the MBR. After identifying the boot partition, it loads the boot sector from that partition and continues the boot process by loading subsystems and system services.

16.5.3 Bad Blocks

Disks are prone to failure due to their moving parts, with some sectors becoming defective over time. In cases of complete failure, the disk is replaced and data restored from backups. For defective sectors, simple disks may handle bad blocks manually. When formatting, the disk scans for bad blocks and marks them in the FAT to prevent their use. If blocks fail during operation, a program like chkdsk must be run to find and lock the bad blocks, though data on those blocks is often lost.

Sophisticated disks, like SCSI, handle bad-block recovery more efficiently. The controller keeps a list of bad blocks, updated over time, and sets aside spare sectors during low-level formatting. When a bad sector is detected, the controller can replace it with a spare sector through a process called sector sparing. Alternatively, sector slipping can move data from the bad sector to a new location, shifting subsequent sectors. While sector sparing and slipping help avoid data loss, the data in the bad block is usually lost, and manual intervention may be needed for file repair, especially in cases of unrecoverable errors.

16.6 SWAP-SPACE MANAGEMENT

Swapping, originally used to move entire processes between disk and memory, is now combined with virtual memory techniques. Modern systems swap pages instead of full processes, often using "swapping" and "paging" interchangeably. Swap-space management is a key task for the operating system, as virtual memory extends main memory to disk. Since disk access is slower, efficient swap-space design aims to optimize throughput for virtual memory, balancing performance and memory management.

16.6.1 Swap-Space Use

Swap space usage varies by operating system and memory management algorithms. Systems using swapping may store entire process images, while paging systems store individual pages. The required swap space depends on physical and virtual memory sizes and usage, ranging from a few megabytes to gigabytes. Overestimating swap space is safer than underestimating, as running out can cause system crashes.

Some systems, like Solaris, recommend swap space based on virtual memory exceeding pageable physical memory, while Linux has historically recommended double the physical memory. Linux now allows multiple swap spaces across different disks to distribute I/O load.

16.6.2 Swap-Space Location

Swap space can be located in either the normal file system or a separate disk partition. Using a file system for swap space is easy but inefficient, as it requires navigating directory structures, which can cause fragmentation and slow performance. Performance can be improved with caching and allocating contiguous blocks, but the cost of file-system overhead remains.

Alternatively, a raw partition can be used for swap space, bypassing the file system and using a separate manager optimized for speed. While this reduces fragmentation, it requires fixed partitioning and reinitialization at boot. Some systems, like Linux, allow swapping in both raw partitions and file-system space, offering a trade-off between ease of management and performance.

16.6.3 Swap-Space Management: An Example

UNIX systems evolved from using full process swapping to a combination of swapping and paging as paging hardware became available. In Solaris 1 (SunOS), the system optimized efficiency by reading text-segment pages (code) from the file system directly instead of swapping them. Swap space was used only for anonymous memory, such as the stack, heap, and uninitialized data. In later versions of Solaris, swap space is allocated only when a page is forced out of physical memory, improving performance on systems with more physical memory and less frequent paging.

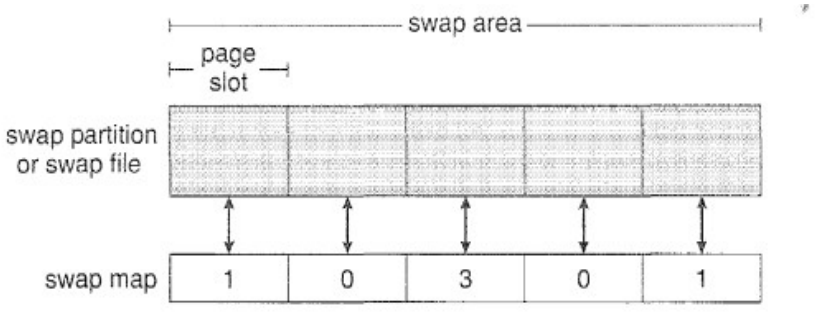


Figure 16.10 The data structures for swapping on Linux systems.

Linux uses swap space for anonymous memory and shared memory regions between processes. It allows multiple swap areas, which can be either swap files on a regular file system or raw swap partitions. Each swap area consists of 4-KB page slots, with a corresponding swap map. If a counter is 0, the page slot is free; if greater

than 0, it indicates the page is occupied, with the counter showing the number of mappings to the swapped page, such as shared pages across multiple processes.

16.7 RAID STRUCTURE

With the decrease in cost and size of disk drives, it is now practical to attach many disks to a system. This allows for parallel data access, improving read/write speeds and enhancing data reliability through redundancy. Redundant Arrays of Independent Disks (RAID) are used to address performance and reliability concerns. While RAIDs were once a cost-effective alternative to large disks, they are now valued more for their higher reliability and data transfer rates.

16.7.1 Improvement of Reliability via Redundancy

In a RAID setup, the failure rate of a disk increases as the number of disks in the array grows. For instance, with 100 disks, the mean time to failure drops to 1,000 hours. To prevent data loss from frequent disk failures, redundancy is introduced. The simplest method is mirroring, where each disk has a duplicate. Data is written to both disks, ensuring that if one fails, the other can still provide the data. Data loss occurs only if both disks fail before one is replaced.

The reliability of a mirrored disk system depends on the mean time to failure of individual disks and the mean time to repair. Assuming independent disk failures, a mirrored system can have a mean time to data loss of about 57,000 years. However, factors like power failures, natural disasters, manufacturing defects, and aging disks can affect reliability. Power failures are especially concerning, as they can leave disks in an inconsistent state if writes are not completed. Solutions include writing to one disk first or using a nonvolatile RAM (NVRAM) cache to protect data during power outages.

16.7.2 Improvement in Performance via Parallelism

Parallel access to multiple disks enhances performance through techniques like **disk mirroring** and **data striping**.

- **Disk Mirroring**: Read requests can be distributed across mirrored disks, doubling the rate of handling read operations while maintaining the same transfer rate per request.
- **Data Striping**: Data is distributed across multiple disks to improve transfer rates.
    - In **bit-level striping**, bits of each byte are split across disks, enabling simultaneous reads/writes, which significantly increases data transfer rates per access.

- In **block-level striping** (more common), file blocks are distributed across disks using a modular arithmetic approach, allowing for efficient parallelism.

Goals of Disk Parallelism:

1. Enhance throughput for multiple small accesses through load balancing.
2. Reduce response time for large data accesses.

16.7.3 RAID Levels

Mirroring ensures high reliability but is costly, while striping boosts data transfer rates without enhancing reliability. To balance cost and redundancy, **RAID levels** combine striping with parity bits for error correction. These levels offer varying cost-performance trade-offs. In RAID, redundant information is stored on extra disks to enable failure recovery, as shown in Figure 16.11, where "P" represents error-correcting bits and "C" denotes a data copy.

- **RAID Level 0**: Uses block-level striping without redundancy, providing high performance but no fault tolerance (Figure 16.11(a)).
- **RAID Level 1**: Implements disk mirroring, duplicating data for high reliability (Figure 16.11(b)).
- **RAID Level 2**: Uses memory-style error-correcting code (ECC) with bit-level striping and additional disks for parity. It detects and corrects single-bit errors. For four data disks, only three additional disks are required for redundancy, unlike RAID 1, which needs four (Figure 16.11(c)).
- **RAID Level 3**: Uses bit-interleaved parity. A single parity disk allows for error detection and correction by computing the parity of bits from other disks. It requires only one parity disk, reducing overhead compared to RAID 2. While read/write transfer rates are high due to N-way striping, fewer I/O operations are supported because all disks participate in every request. Parity computation slows writes, but dedicated hardware controllers with caching can mitigate this issue (Figure 16.11(d)).
- **RAID Level 4**: Implements block-level striping with a dedicated parity disk. It supports parallel read operations for higher I/O rates and efficient large read/write transfers. However, small writes trigger a read-modify-write cycle, requiring multiple disk accesses (two reads and two writes). This design allows seamless addition of new disks by initializing them with zeros, maintaining parity integrity (Figure 16.11(e)).
- **RAID Level 5**: Uses block-level striping with distributed parity across all disks. Parity for each block is stored on a different disk to balance load and avoid overuse of a single parity disk. It is the most common parity RAID level, offering a good balance of performance and reliability (Figure 16.11(f)).

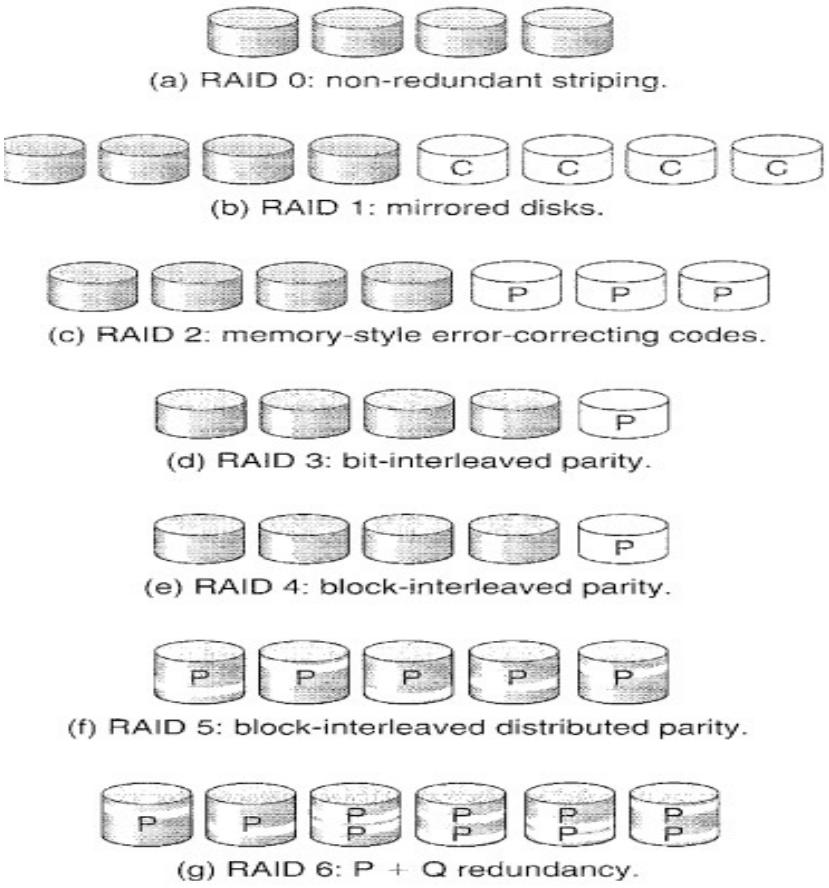


Figure 16:11 RAI D levels.

- **RAID Level 6**: Extends RAID 5 with additional redundancy (e.g., Reed-Solomon codes) to protect against two disk failures. It stores two bits of redundant data for every four bits of data, enhancing fault tolerance (Figure 16.11(g)).
- **RAID Level 0+1**: Combines RAID 0 (striping) and RAID 1 (mirroring) to achieve high performance and reliability. Data is striped first and then mirrored. However, it requires double the number of disks, making it expensive.
- **RAID Level 1+0**: Mirrors disks in pairs and then stripes the mirrored pairs. It offers better fault tolerance than RAID 0+1, as a single disk failure only affects its pair, leaving other disks accessible (Figure 16.12).

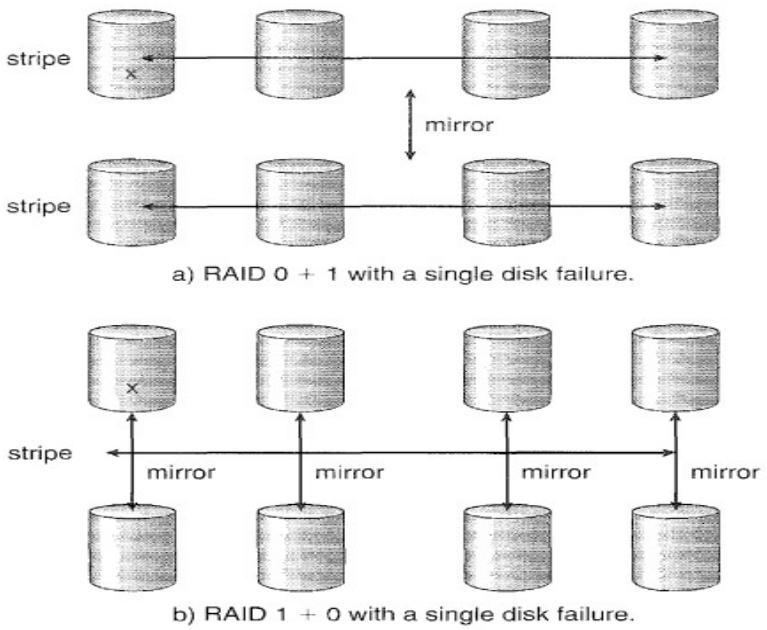


Figure 16.12 RAID a+ 1 and 1 + O

RAID implementations vary across several layers, leading to some confusion about RAID level definitions:

- **Software RAID**: Implemented via volume-management software in the OS kernel. Often limited to RAID 0, 1, or 0+1 due to slow parity computation.
- **Host Bus Adapter (HBA) RAID**: Implemented in HBA hardware, connecting directly to disks. It is cost-effective but lacks flexibility.
- **Storage Array RAID**: Hardware within storage arrays handles RAID, allowing flexible RAID set creation and slicing into volumes. Suitable for SANs and multi-host access.
- **SAN Layer RAID**: Disk virtualization devices in SANs manage RAID, offering features like mirroring by writing data to separate devices.

RAID implementations can include additional features like **snapshots** and **replication** for redundancy and disaster recovery:

- **Replication**: Automatically duplicates writes between sites.
  - **Synchronous**: Writes are completed both locally and remotely, ensuring no data loss but slower.
  - **Asynchronous**: Writes are batched and periodically updated, faster but risks data loss if the primary site fails.
  - Implementation varies: software-based RAID requires host management, while storage arrays or SAN interconnects handle replication centrally.

Hot Spares: Unused disks configured as replacements in case of disk failure. They enable automatic RAID rebuilding and can support multiple failures without manual intervention.

16.7.4 Selecting a RAID Level

System designers choose a RAID level based on factors like rebuild performance and reliability. Rebuilding is easiest in RAID 1, as data can be copied from another disk, while other levels require accessing multiple disks. Rebuild performance is critical for systems requiring continuous data availability, like high-performance databases. RAID 0 is used for high performance without data loss concerns, RAID 1 for high reliability, and RAID 0+1 or 1+0 for both performance and reliability. RAID 5 is preferred for large data volumes due to its lower space overhead, while RAID 6 offers better reliability but is less supported.

Designers also consider the number of disks in an array and the amount of data protected by each parity bit. More disks increase transfer rates but cost more, while more data protected by a single parity bit lowers overhead but increases the risk of data loss if another disk fails before repair.

16.7.5 Extensions

RAID concepts extend to other storage and broadcast systems.

- **Tape Arrays**: RAID can recover data from damaged tapes.
- **Data Broadcast**: Data blocks split into units with parity enable reconstruction of missing units.
- **Tape-Drive Robots**: Stripe data across multiple drives to boost throughput and reduce backup time.

16.7.6 Problems with RAID

RAID protects against physical media errors but not software or hardware issues like incorrect file pointers, incomplete writes, or accidental overwrites.

The **Solaris ZFS file system** addresses these by:

- Using internal checksums for all data and metadata.
- Storing checksums with pointers, not the blocks themselves, for robust error detection.
- Automatically repairing corrupted blocks using mirrored copies with correct checksums.
- Ensuring higher consistency and error correction compared to standard file systems or RAID.

Despite added overhead, ZFS maintains high performance.

16.8 SUMMARY

Disk drives are the main secondary-storage devices, typically magnetic disks or tapes. They are organized as large arrays of blocks, usually 512 bytes. Disks can be connected through local I/O ports or network connections.

Disk I/O requests use logical block numbers, and scheduling algorithms like SSTF, SCAN, and C-SCAN help improve performance. Fragmentation can reduce performance, but systems like UNIX reduce this through efficient space allocation.

Disk management includes formatting, partitioning, and creating file systems. Corrupted blocks are either locked or replaced. For better performance, systems often use raw disk access for swap space. RAID algorithms provide redundancy, improving reliability and performance, even during disk failures.

16.9 KEY TERMS

Mass Storage, Disk Structure, Disk Attachment, Disk Scheduling, Disk Management, Swap Space, RAID, Storage Efficiency, Data Redundancy, Performance Optimization

16.10 SELF-ASSESSMENT QUESTIONS

1. What is the mass-storage structure, and how does it support data storage?
2. Explain the components and organization of disk structures.
3. What are the different types of disk attachments, and how do they affect system performance?
4. Discuss various disk scheduling algorithms and their advantages.
5. How is swap space managed in operating systems, and why is it important?
6. What is RAID, and how does it enhance data reliability and storage performance?

16.11 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”, Seventh Edition, Wiley.
2. William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3. Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4. Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5. Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Dr U Surya Kameswari

CHAPTER 17

I/O SYSTEMS

Aim and Objectives:

At the end of the lesson students will be able to understand the concepts of:

- Understand the structure of the I/O subsystem in an operating system.
- Learn how I/O hardware works and its complexity.
- Understand the performance factors of I/O hardware.
- Examine how I/O software improves performance.
- Explore the relationship between I/O hardware and software performance.

Structure:

17.1 OVERVIEW

The operating system's design must carefully control the different devices connected to a computer, like a mouse, hard disk, or CD-ROM, each with unique functions and speeds. To manage these differences, the OS uses an I/O subsystem, which shields the rest of the OS from the complexities of handling varied devices. While more standardized interfaces make it easier to integrate new devices, the increasing variety of devices still requires specialized hardware and software. Device drivers in the OS provide a uniform way for the system to interact with any device, much like system calls provide a standard way for applications to interact with the OS.

17.2 I/O HARDWARE

Computers use various devices, including storage devices (like disks), transmission devices (like network cards), and human-interface devices (like keyboards and mice). Some devices, like those in fighter jets, are specialized, using inputs like joysticks to control specific parts such as rudders. Despite their variety, all devices connect and communicate with computers using a few core concepts.

Devices connect to computers via ports (like serial ports) or shared connections called buses. A bus allows multiple devices to communicate by sending messages through shared wires. When devices are connected in sequence to one port, it’s called a daisy chain. PCs commonly use buses like the PCI bus for fast devices and slower expansion buses for peripherals like keyboards.

Controllers manage device communication. For instance, a simple controller handles serial port signals, while a complex SCSI controller manages multiple disks using its own processor. Disk drives have a controller that performs tasks like mapping bad sectors, buffering, caching, and prefetching data.

The CPU communicates with these controllers through registers using two methods:

1. **I/O instructions**: Data is sent to specific I/O ports.
2. **Memory-mapped I/O**: The controller's registers are part of the CPU's memory address space.

Many systems, like PCs, use both methods. For example, a graphics controller might use I/O ports for basic controls and memory-mapped I/O for faster data transfer to display images efficiently.

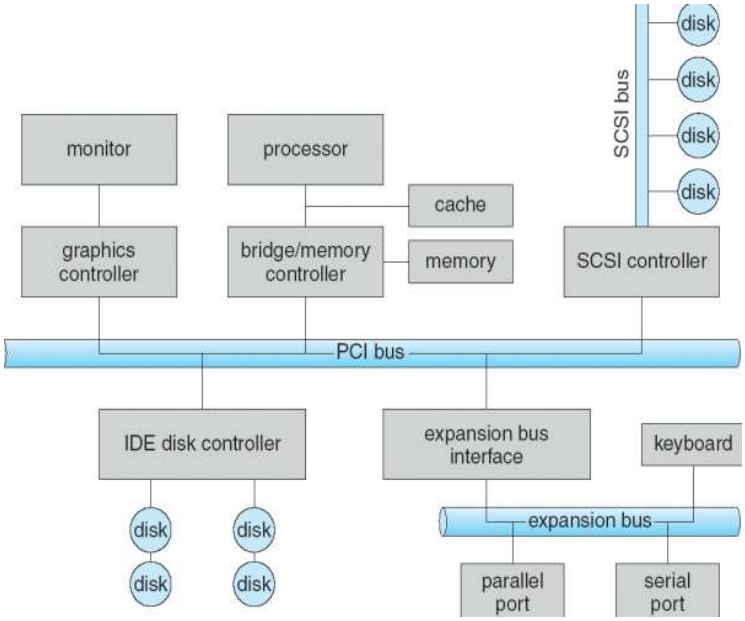


Figure 17.1 A typical PC bus Structure

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Figure 17.2 Device I/O part locations on PCs (partial).

An I/O port generally has four main registers: **status**, **control**, **data-in**, and **data-out**.

- The **data-in register** is read by the CPU to receive input
- The **data-out register** is written to send output.
- The **status register** shows device states, such as command completion or errors
- The **control register** is used to initiate commands or change device settings, like communication mode or speed.

Data registers are small (1–4 bytes), but some controllers use FIFO chips to hold extra data temporarily, enabling smoother data flow between the device and the host.

17.2.1 Polling

- Polling is a simple handshaking method where the CPU checks a device's **busy** status until it's ready.
- The CPU sets a **command-ready** bit to tell the controller a command is available.
- The controller sets the **busy** bit while processing and clears it when done, allowing the CPU to send the next command.
- Polling works well if the device is fast but can waste CPU time if waiting is long.

In those cases, it's more efficient for the device to signal the CPU when it's ready, using a mechanism called an **interrupt**, which alerts the CPU without constant checking.

17.2.2 Interrupts

The **interrupt mechanism** allows the CPU to pause its current tasks and handle important events, such as signals from device controllers, through an **interrupt-request line**. When an interrupt is detected, the CPU saves its state, jumps to a designated **interrupt handler**, processes the event, and then resumes its previous task.

For advanced needs, such as handling **critical processing**, modern systems support:

1. **Deferred interrupts** to manage timing in critical tasks.
2. **Efficient dispatching** to the correct handler without polling each device.
3. **Multilevel interrupts** to prioritize high-importance tasks.

To manage interrupts, CPUs typically have **nonmaskable** (for serious errors) and **maskable** (for device requests) lines. The **interrupt vector** maps specific interrupts to handlers, and **interrupt chaining** helps systems with more devices than vector slots by linking related handlers.

To ensure proper handling order, modern CPUs use **priority levels** to allow high-priority interrupts to preempt lower ones. During **boot**, the OS sets up these handlers based on connected devices, and during I/O, device controllers raise interrupts for events like input completion or error detection.

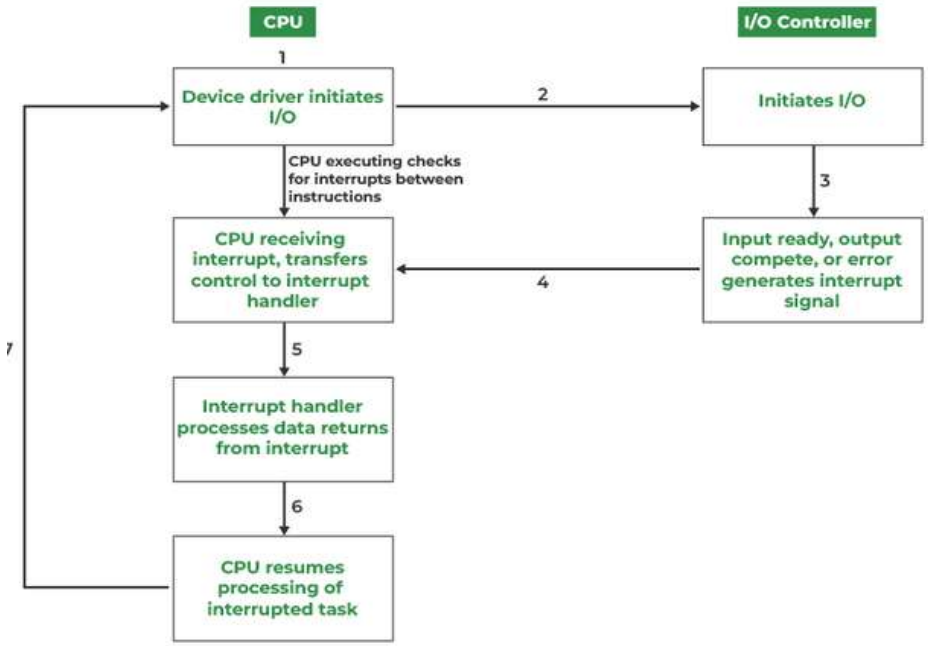


Figure 17.3 Interrupt-driven I/O cycle.

Interrupts are also used for **exception handling**—for example, when a **page fault** occurs in virtual memory, the CPU suspends the current process to address it. The interrupt mechanism is critical for real-time responses in operating systems, ensuring swift handling of devices and errors.

The interrupt handler's job in the kernel includes saving the process state, moving it to a **wait queue**, managing **page-cache**, initiating I/O for a page, scheduling other processes, and returning from the interrupt. **System calls** are handled by issuing a software interrupt (or trap) where a library checks arguments, prepares data, and sends a request to the kernel. The CPU then saves the state, switches to supervisor mode, and executes the system call, prioritizing it below device interrupts, as device service is often more urgent.

Interrupts also manage control flow within the kernel. For example, in **disk read completion**, a high-priority handler records I/O status, initiates the next I/O, and triggers a low-priority interrupt. This low-priority handler later copies data to user space when no high-priority tasks are pending, and places the application in the **ready queue**.

In **threaded kernels** like Solaris, interrupt handlers run as high-priority threads, ensuring they preempt lower-priority tasks and can execute on multiple processors. This enables efficient, **priority-based interrupt management**, where high-priority interrupts (like device I/O) preempt lower-priority ones.

17.2.3 Direct Memory Access

For devices that handle large data transfers, like disk drives, **programmed I/O (PIO)**, where the CPU monitors data transfers byte by byte, is inefficient. Instead, many systems offload this work to a **Direct Memory Access (DMA) controller**. To start a DMA transfer, the CPU sets up a **DMA command block** in memory, which includes pointers to the source and destination addresses and the data size. The CPU then instructs the DMA controller to handle the transfer, allowing the CPU to focus on other tasks.

The **DMA controller** manages data movement independently by placing addresses on the memory bus directly, without CPU involvement. Standard DMA controllers are common in PCs, and bus-mastering I/O boards often include advanced DMA hardware.

Handshaking between the DMA and device controllers occurs via **DMA-request** and **DMA-acknowledge** signals. When data is ready, the device controller raises the DMA-request signal, prompting the DMA controller to seize the memory bus and send the address to memory. After receiving the DMA-acknowledge signal, the device transfers data to memory and removes the request signal.

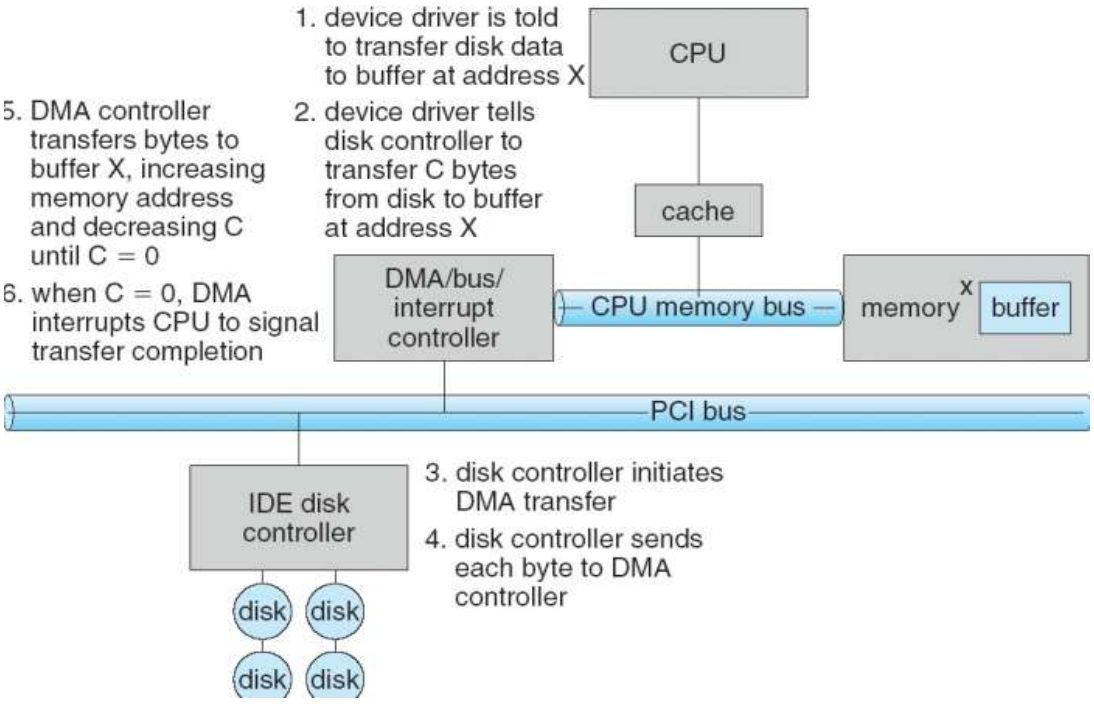


Figure 17.4 Steps in a DMA transfer.

When a DMA transfer is complete, the DMA controller **interrupts the CPU**. During data transfer, the DMA controller may temporarily prevent the CPU from accessing main memory by **seizing the memory bus**, though the CPU can still access its caches. This process, known as **cycle stealing**, may slow CPU computations

slightly but generally **improves overall system performance** by offloading data-transfer tasks.

Some systems use **Direct Virtual Memory Access (DVMA)**, where transfers use virtual addresses that translate to physical addresses, allowing data movement between memory-mapped devices without CPU or main memory intervention.

In **protected-mode kernels**, the operating system restricts direct device access to **protect system stability and security**. Only privileged processes can use system-provided functions to perform low-level operations on hardware. However, kernels without memory protection allow processes direct access to device controllers, which improves performance by bypassing kernel layers but risks system security and reliability. Most modern operating systems prioritize **memory and device protection** to prevent malicious or erroneous applications from compromising system stability.

17.3 APPLICATION I/O INTERFACE

This section discusses how operating systems structure I/O interfaces to allow devices to be managed uniformly. Using **abstraction, encapsulation, and software layering**, the OS can standardize device access. Applications, for example, can open a file on a disk without knowing the disk type, and new devices can be added without OS disruption.

The **device-driver layer** abstracts device differences for the I/O subsystem by encapsulating specific device behaviors into kernel modules called **device drivers**. These drivers provide a standardized interface, regardless of device specifics, allowing the OS to manage diverse hardware more efficiently. This standardization simplifies tasks for OS developers and benefits hardware manufacturers. New devices can either conform to existing controller standards or include device drivers, enabling integration with existing operating systems without needing direct OS support for every new device.

Device-hardware manufacturers face challenges because each operating system has unique standards for device drivers, requiring multiple drivers for compatibility across systems (e.g., MS-DOS, Windows, Solaris). Devices differ in several key aspects:

- **Character-stream vs. Block**: Character-stream devices transfer data byte by byte, while block devices transfer in blocks.
- **Sequential vs. Random Access**: Sequential devices transfer data in a set order, while random-access devices allow data retrieval from any location.
- **Synchronous vs. Asynchronous**: Synchronous devices transfer data at predictable times; asynchronous devices do not.
- **Sharable vs. Dedicated**: Sharable devices can be used by multiple processes, whereas dedicated devices are limited to one.

- **Speed of Operation**: Device speeds vary widely, from a few bytes to gigabytes per second.
- **Read-Write Modes**: Devices may support both reading and writing, or only one direction.

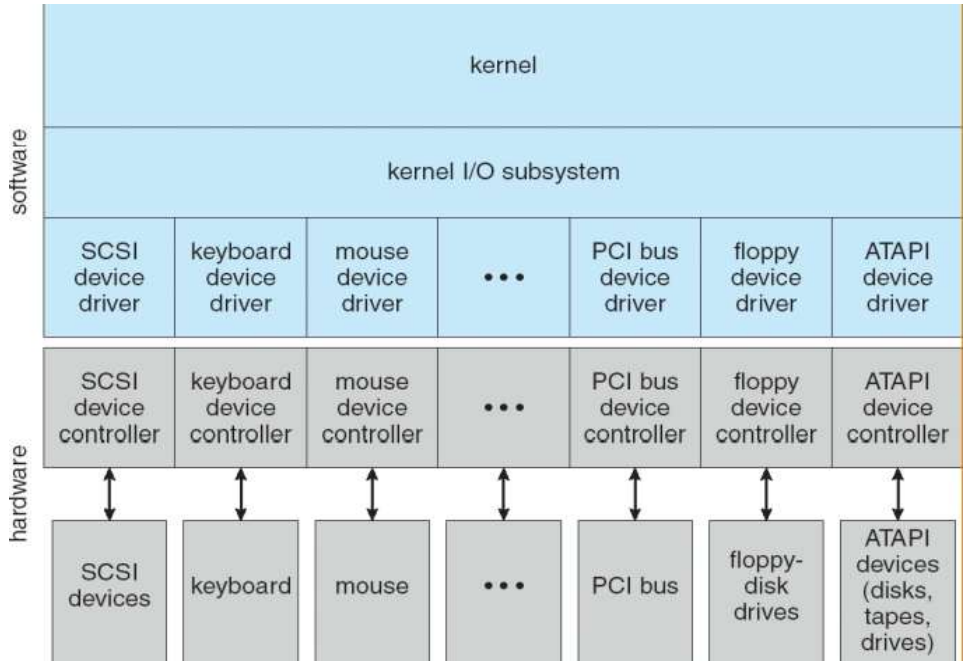


Figure 17.5 A kernel I/O structure

17.3.1 Block and Character Devices

The block-device interface standardizes disk access with commands like read(), write(), and seek(), abstracting device details through a file system. For apps managing their own buffering/locking, "raw I/O" provides direct access, while "direct I/O" disables file-system buffering/locking selectively.

Memory-mapped files allow efficient access by mapping files to memory, supporting on-demand transfers. For character-stream devices (e.g., keyboards), get() and put() commands handle unpredictable data input/output, with libraries for line-based access and buffering. This interface also supports linear output for devices like printers.

17.3.2 Network Devices

Network I/O uses a separate interface from disk I/O due to differences in performance and addressing. The socket interface, common in UNIX and Windows NT, allows applications to create, connect, listen, send, and receive data via sockets. The select() function manages multiple sockets efficiently, avoiding busy waiting.

This standardized interface simplifies distributed applications, making them compatible with various network setups. Systems like Windows NT and UNIX offer additional inter-process communication options, such as pipes, FIFOs, and message queues, enhancing flexibility in network communication.

17.3.3 Clocks and Timers

Computers use hardware clocks and timers for three main functions: giving the current time, tracking elapsed time, and setting future actions. A programmable interval timer, set for single or periodic interrupts, supports these functions, aiding the scheduler in process pre-emption and I/O subsystems for caching and network management. Although timer system calls vary across operating systems, virtual clocks help manage multiple timer requests by tracking interrupts in a queue. Hardware clocks tick between 18 and 60 times per second, limiting precision, but high-frequency counters on some systems provide high-resolution time measurements without interrupts.

17.3.4 Blocking and Nonblocking IO

System calls can be **blocking** or **nonblocking**:

- **Blocking I/O:** The application pauses until the I/O operation is finished.
- **Nonblocking I/O:** The application keeps running while the I/O operation happens in the background.

In multithreaded programs, blocking I/O can run in one thread while others continue processing. Some operating systems offer nonblocking calls that return immediately with information about the transfer status.

Asynchronous I/O is different—it also returns immediately, but the system notifies the application later (using signals or callbacks) when the operation is done.

An example of nonblocking I/O is the select() system call, which lets programs check if I/O is ready without stopping. However, this approach can add extra processing overhead.

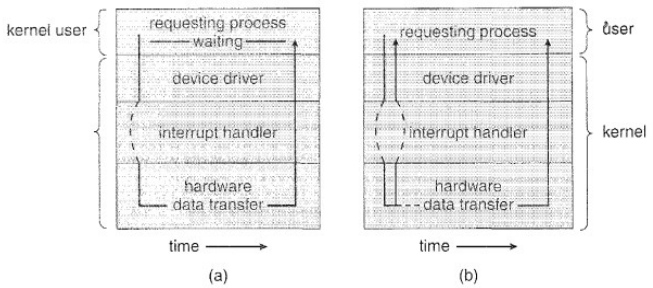


Figure 17.6 Two I/O methods: (a) synchronous and (b) asynchronous

17.4 KERNEL I/O SUBSYSTEM

Kernels offer I/O services like scheduling, buffering, caching, spooling, device reservation, and error handling, all built on hardware and device-driver infrastructure, while also protecting against faulty processes and malicious users.

17.4.1 I/O Scheduling

I/O scheduling optimizes the order of I/O requests to improve system performance, fairness, and response time. The OS maintains a wait queue for each device, rearranges requests for efficiency, and may prioritize delay-sensitive tasks.

The OS prioritizes certain I/O requests, like virtual memory, and manages multiple requests via a device-status table showing device type, address, and state. Efficiency is further enhanced by scheduling, buffering, caching, and spooling.

17.4.2 Buffering

Buffering temporarily stores data to manage differences in speed or size between devices and ensures data is handled correctly during I/O operations.

- **Double buffering**: Helps transfer data between slower devices and faster storage by using two buffers. This is useful in tasks like handling network packets.
- **Copy semantics**: Data is copied to the kernel buffer before being written to the disk. This prevents issues if the application changes the original data but can slightly increase overhead.

17.4.3 Caching

A cache stores copies of data for quick access, improving performance, while a buffer temporarily holds data and may contain the only copy. Operating systems often use buffers as a cache to speed up disk I/O by quickly accessing frequently used files and combining disk writes, reducing the number of direct disk operations.

17.4.4 Spooling and Device Reservation

A spool is a buffer holding output for devices like printers that handle one job at a time, preventing mixed outputs from concurrent applications. Spooling stores each job in a queue and prints them sequentially. Some operating systems use a system daemon or kernel thread to manage spooling, offering controls for managing the print queue. For devices that cannot multiplex I/O, OS provide exclusive access methods, such as Windows NT's wait function for device availability, leaving deadlock prevention to applications.

17.4.5 Error Handling

Protected memory in an OS helps prevent system failures from minor errors. Transient I/O errors are often retried, but permanent hardware failures are harder to

handle. Most OS indicate I/O success or failure simply, with UNIX using errno codes for more detail. SCSI devices offer multi-level error details but are rarely fully utilized by current OS.

17.4.6 I/O Protection

To prevent illegal I/O, the OS makes I/O instructions privileged, requiring user programs to request I/O through system calls. The OS, running in monitor mode, validates and executes these requests. Memory-mapped I/O areas are also protected but can be selectively accessed, as with graphics applications, through kernel-provided locking mechanisms to ensure secure, efficient access.

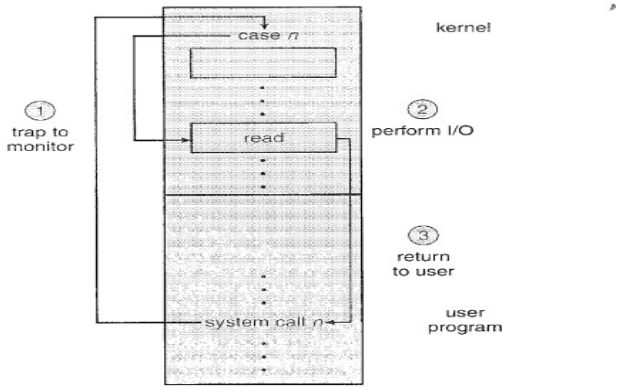


Figure 17.7 Use of a system call to perform I/O.

17.4.7 Kernel Data Structures

The kernel maintains in-kernel data structures, like open-file tables, to track I/O components. I/O requests are converted to messages sent to the I/O manager and device driver, enhancing flexibility through message-passing but with added overhead. UNIX uses a uniform file access structure with a dispatch table for different entities, such as files, raw devices, and process images. Systems like Windows NT rely extensively on message-passing for I/O, trading some efficiency for design simplicity and flexibility.

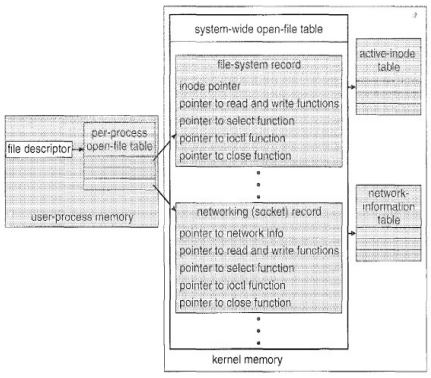


Figure 17.8 UNIX I/O kernel structure.

17.5 Transforming I/O Requests to Hardware Operations

To connect an application request to hardware, OS maps file names to disk space using lookup tables, such as inodes in UNIX or file-access tables in MS-DOS. MS-DOS identifies devices with prefixes like "c:" for primary disks, while UNIX uses a mount table to link path prefixes to devices. In UNIX, device numbers help the OS locate device drivers. Modern OS use multiple lookup stages for flexibility, allowing new devices to be added without kernel recompilation, with some systems loading drivers as needed by probing hardware during boot.

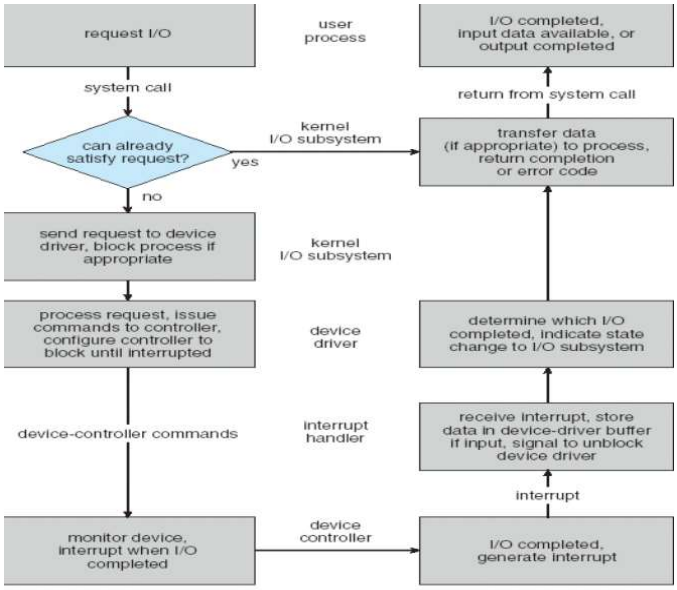


Figure 17.9 The life cycle of an I/O request

The life cycle of a blocking read request involves numerous steps, consuming significant CPU cycles:

1. A process issues a blocking read() system call.
2. The kernel checks parameters; if data is in the buffer cache, it's returned to the process.
3. If not, the process is moved to a wait queue, and the I/O request is scheduled.
4. The device driver allocates kernel buffer space and sends commands to the device controller.
5. The device controller performs the data transfer.
6. Data is transferred via DMA, generating an interrupt upon completion.
7. The interrupt handler stores necessary data and signals the driver.
8. The driver verifies and signals completion to the I/O subsystem.
9. The kernel transfers data to the process and moves it to the ready queue.
10. The scheduler assigns the process to the CPU, where it resumes execution.

17.6 STREAMS

UNIX System V's **STREAMS** is a mechanism that allows applications to build dynamic pipelines for device drivers. A STREAM is a full-duplex connection between a device driver and a user process, comprising a **stream head** (which interfaces with the user), a **driver end** (which controls the device), and optional **stream modules** in between. Each part contains read and write queues, and **message passing** transfers data between these queues. **Flow control** is essential for managing buffer capacity, preventing overflow when modules pass messages.

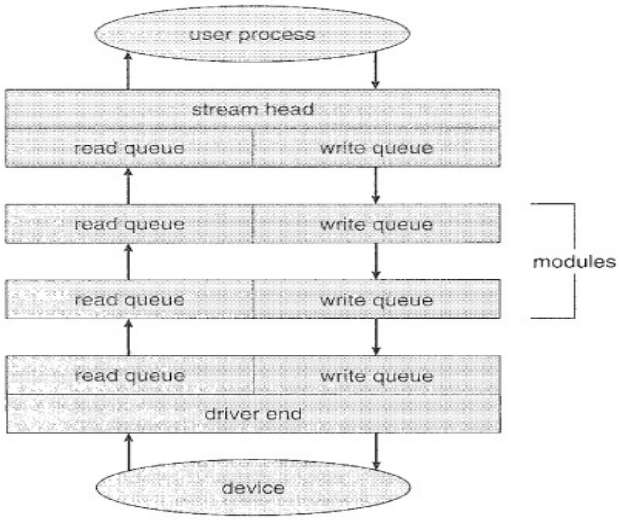


Figure 17.10 The STREAMS structure

User processes interact with STREAMS through **system calls** like write() for raw data and putmsg() to specify messages. The data flows through the stream head, moves through intermediate modules, and reaches the device. Similarly, data from the device is read by the user process through read() or getmsg().

STREAMS I/O is **asynchronous** (nonblocking) except when the user process interacts with the stream head. A process will block while waiting for available buffer space in the next queue if flow control is active. The **driver end** must handle all incoming data without blocking; if buffers are full, it may drop excess data (e.g., a network card drops incoming packets when its buffer is full).

STREAMS provides a **modular approach** for developing device drivers and network protocols, as modules can be reused across multiple devices, like Ethernet and token-ring cards. STREAMS also supports **structured message boundaries** and **control information**. Widely adopted in UNIX variants, STREAMS is used for protocols and device drivers, with System V UNIX and Solaris even implementing socket mechanisms through STREAMS.

17.7 PERFORMANCE

I/O significantly impacts system performance, requiring substantial CPU resources to manage device-driver code, process scheduling, and context switching. These demands put stress on the CPU and cache, reveal kernel interrupt-handling inefficiencies, and create heavy memory-bus traffic due to data transfers between controllers and memory. Modern CPUs can handle many interrupts per second, but **interrupt handling** is costly, as each one requires a state change, running an interrupt handler, and then restoring the system state. In some cases, **programmed I/O** can be more efficient than interrupt-driven I/O, especially when the CPU cycles spent on busy waiting are low.

Network I/O generates frequent context switches, such as in remote logins, where each character typed triggers multiple interrupts, data transfers, and kernel calls across both local and remote systems. This is resource-intensive as each step involves state changes and context switches. In Solaris, in-kernel threads were implemented to reduce context switching, enabling thousands of network logins by handling telnet processing directly in the kernel.

To manage I/O more efficiently, some systems offload tasks to **front-end processors** or **terminal concentrators** that multiplex traffic, reducing the main CPU's interrupt load. **I/O channels** on high-end systems or mainframes offload I/O handling from the main CPU entirely. These channels are specialized CPUs for managing I/O, allowing smooth data flow while freeing the main CPU to perform primary processing tasks, similar to device controllers and DMA controllers on smaller systems but with more advanced capabilities for tuning specific workloads.

To enhance I/O efficiency, several principles can be applied:

- **Reduce context switches** and minimize **memory data copying** between devices and applications.
- **Decrease interrupt frequency** by using larger transfers, smart controllers, or polling (if busy waiting is manageable).
- **Increase concurrency** using DMA-aware controllers or channels to offload data transfers from the CPU.
- **Integrate processing primitives into hardware** to allow concurrent operations between device controllers and CPU.
- **Balance performance across CPU, memory, bus, and I/O** to prevent bottlenecks.

Device complexity varies significantly; for example, a mouse is simple, while disk drivers, like the Windows NT driver, are complex, managing disks and RAID arrays with performance optimization, error handling, and data recovery.

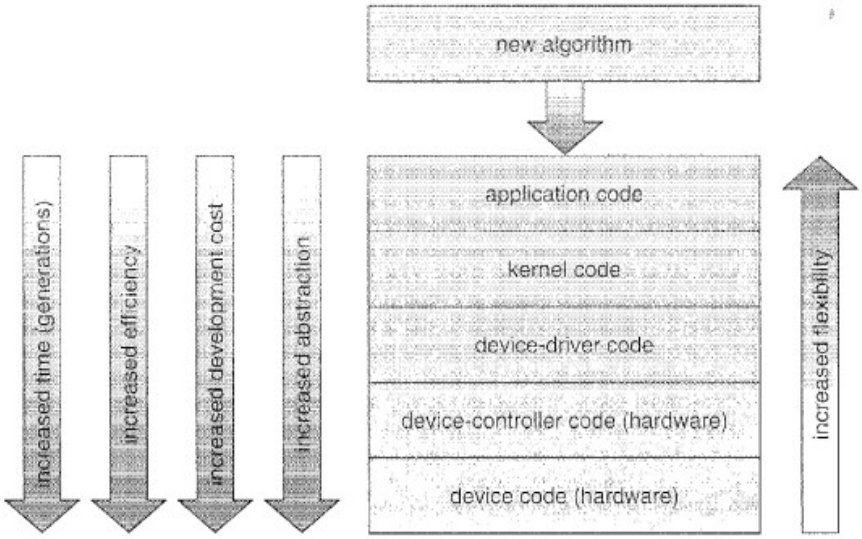


Figure 17.11 Device functionality progression.

I/O functionality placement (application, kernel, or hardware) affects performance:

- **Application level**: Useful for experimenting with new I/O algorithms, as it offers flexibility and reduces crash risks, though it has higher overhead and lacks direct kernel support.
- **Kernel level**: Boosts performance once an algorithm has proven effective, but development is challenging and must be stable to avoid data issues or system crashes.
- **Hardware level**: Provides the highest performance, though improvements and bug fixes are difficult and costly, development is slower, and flexibility is limited, as the hardware may restrict the kernel's control over certain tasks like block read/write order.

17.8 SUMMARY

The key hardware elements in I/O systems are buses, device controllers, and devices. Data transfer between devices and memory is handled by the CPU through programmed I/O or offloaded to a DMA controller. Device control is managed by device drivers, while the system-call interface supports block devices, character devices, memory-mapped files, network sockets, and timers. System calls often block processes but can be nonblocking for specific use cases.

The kernel's I/O subsystem provides services like I/O scheduling, buffering, caching, spooling, device reservation, error handling, and name translation, which maps device names to physical addresses. STREAMS allows drivers to be reusable and stackable for bidirectional data processing.

I/O system calls are resource-intensive due to context switching, signal and interrupt handling, and data copying between kernel buffers and application memory. These processes introduce significant CPU overhead.

17.9 KEY TERMS

I/O Hardware, Application I/O Interface, Kernel I/O Interface, Hardware Operations, I/O Requests, Streams, Performance, Device Drivers, I/O Scheduling, Buffering

17.10 SELF-ASSESSMENT QUESTIONS

1. What are the key components of I/O hardware, and how do they function?
2. Explain the role of the application I/O interface in an operating system.
3. How does the kernel I/O interface transform I/O requests into hardware operations?
4. What are streams, and how are they used in I/O systems?
5. Discuss the performance challenges in I/O systems and how they are addressed.
6. Describe the process of I/O scheduling and its importance in an operating system.

17.11 FURTHER READINGS

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. “Operating System Principles”, Seventh Edition, Wiley.
2. William Stallings, “Operating Systems – Internals and Design Principles”, Fifth Edition, Pearson Education (2007)
3. Achyut S Godbole, “Operating Systems”, Second Edition, TMH (2007).
4. Flynn/McHoes, “Operating Systems”, Cengage Learning (2008).
5. Deitel&Deitel, “Operating Systems”,Third Edition, Pearson Education (2008)

NAME OF THE AUTHOR

Dr U Surya Kameswari