



ALAGAPPA UNIVERSITY

[Accredited with 'A+' Grade by NAAC (CGPA:3.64) in the Third Cycle
and Graded as Category-I University by MHRD-UGC]

(A State University Established by the Government of Tamil Nadu)

KARAIKUDI – 630 003



Vishal Dr. R.M. Alagappa Chettiar

Directorate of Distance Education

B.Sc. (Computer Science)

V - Semester

130 51

OPERATING SYSTEMS

"The copyright shall be vested with Alagappa University"

All rights reserved. No part of this publication which is material protected by this copyright notice may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the Alagappa University, Karaikudi, Tamil Nadu.

Information contained in this book has been published by VIKAS® Publishing House Pvt. Ltd. and has been obtained by its Authors from sources believed to be reliable and are correct to the best of their knowledge. However, the Alagappa University, Publisher and its Authors shall in no event be liable for any errors, omissions or damages arising out of use of this information and specifically disclaim any implied warranties or merchantability or fitness for any particular use.



Vikas® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

VIKAS® PUBLISHING HOUSE PVT. LTD.
E-28, Sector-8, Noida - 201301 (UP)
Phone: 0120-4078900 • Fax: 0120-4078999
Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44
• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

Work Order No.AU/DDE/DE12-27/Preparation and Printing of Course Materials/2020 Dated 12.08.2020 Copies.....

SYLLABI-BOOK MAPPING TABLE

Operating Systems

Syllabi	Mapping in Book
BLOCK I: INTRODUCTION	
UNIT 1: Introduction, Components and Goals, Operating System Architecture	Unit 1: Introduction to Operating Systems (Pages 1-26);
UNIT 2: Process Concepts: Introduction, Process States, Process Management	Unit 2: Processes (Pages 27-40);
UNIT 3: Interrupts, Interprocess Communication	Unit 3: Interrupts and Interprocess Communication (Pages 41-65)
BLOCK II: ASYNCHRONOUS CONCURRENT EXECUTION	
UNIT 4: Introduction, Mutual Exclusion, Implementing Mutual Exclusion Primitives	Unit 4: Mutual Exclusion (Pages 66-77);
UNIT 5: Software Solution to the Mutual Exclusion Problem, Hardware Solution to Mutual Exclusion Problem, Semaphores	Unit 5: Mutual Exclusion Problem and Semaphores (Pages 78-106);
UNIT 6: Concurrent Programming, Introduction, Monitors	Unit 6: Concurrent Programming (Pages 107-126)
BLOCK III: DEADLOCK AND INFINITE POSTPONEMENT	
UNIT 7: Introduction: Examples of Deadlock, Related Problem Indefinite Postponement, Resource Concepts	Unit 7: Introduction to Deadlocks (Pages 127-140);
UNIT 8: Conditions for Deadlock: Deadlock Solution, Prevention, Avoidance with Dijkstra's Banker Algorithm, Deadlock Detection, Recovery	Unit 8: Conditions for Deadlock (Pages 141-159);
UNIT 9: Processor Scheduling: Introduction, Scheduling Levels, Preemptive vs Nonpreemptive Scheduling Priorities, Scheduling Criteria, Scheduling Algorithms	Unit 9: Process Scheduling (Pages 160-193)
BLOCK IV: REAL MEMORY AND VIRTUAL MEMORY MANAGEMENT	
UNIT 10: Introduction, Memory Organization, Memory Management, Hierarchy, Management Strategies	Unit 10: Memory Management (Pages 194-217);
UNIT 11: Contiguous vs Non-Contiguous Memory Allocation, Fixed Partition Multiprogramming, Variable Partition Multiprogramming	Unit 11: Memory Management Strategies (Pages 218-242);
UNIT 12: Virtual Memory Management: Introduction, Page Replacement, Strategies, Page Fault Frequency, Page Replacement, Page Release, Page Size	Unit 12: Virtual Memory Management (Pages 243-285)
BLOCK V: DISK PERFORMANCE AND FILE, DATABASE SYSTEMS	
UNIT 13: Introduction, Disk Scheduling Strategies, Rotational Optimization	Unit 13: Disk Scheduling (Pages 286-306);
UNIT 14: File and Database System: Introduction, Data Hierarchy, Files, File Systems, File Optimization, File Allocation, Free Space Management, File Access Control	Unit 14: File and Database System (Pages 307-350)

CONTENTS

BLOCK I: INTRODUCTION**UNIT 1 INTRODUCTION TO OPERATING SYSTEMS 1-26**

- 1.0 Introduction
- 1.1 Objectives
- 1.2 Operating Systems – Definition, Purpose and Goals
- 1.3 Components of An Operating System
- 1.4 Evolution of Operating System
- 1.5 Architecture of Operating Systems
- 1.6 Answers to Check Your Progress Questions
- 1.7 Summary
- 1.8 Key Words
- 1.9 Self Assessment Questions and Exercises
- 1.10 Further Readings

UNIT 2 PROCESSES 27-40

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Process Management
- 2.3 Process Concept and Threads
 - 2.3.1 OS View of Processes
 - 2.3.2 Process States
- 2.4 Answers to Check Your Progress Questions
- 2.5 Summary
- 2.6 Key Words
- 2.7 Self Assessment Questions and Exercises
- 2.8 Further Readings

UNIT 3 INTERRUPTS AND INTERPROCESS COMMUNICATION 41-65

- 3.0 Introduction
- 3.1 Objectives
- 3.2 Introduction to Interrupts in Operating System
- 3.3 Interprocess Communication
- 3.4 Answers to Check Your Progress Questions
- 3.5 Summary
- 3.6 Key Words
- 3.7 Self Assessment Questions and Exercises
- 3.8 Further Readings

BLOCK II: ASYNCHRONOUS CONCURRENT EXECUTION**UNIT 4 MUTUAL EXCLUSION 66-77**

- 4.0 Introduction
- 4.1 Objectives
- 4.2 Mutual Exclusion
- 4.3 Implementing Mutual Exclusion Primitives

- 4.4 Answers to Check Your Progress Questions
- 4.5 Summary
- 4.6 Key Words
- 4.7 Self Assessment Questions and Exercises
- 4.8 Further Readings

UNIT 5 MUTUAL EXCLUSION PROBLEM AND SEMAPHORES

78-106

- 5.0 Introduction
- 5.1 Objectives
- 5.2 Critical Section Problem
 - 5.2.1 Software Solutions for Critical Section Problem
 - 5.2.2 Hardware Solutions for Critical Section Problem
 - 5.2.3 Mutual Exclusion Algorithms
- 5.3 Semaphores: Definition
- 5.4 Answers to Check Your Progress Questions
- 5.5 Summary
- 5.6 Key Words
- 5.7 Self Assessment Questions and Exercises
- 5.8 Further Readings

UNIT 6 CONCURRENT PROGRAMMING

107-126

- 6.0 Introduction
- 6.1 Objectives
- 6.2 Sequential and Concurrent Process
- 6.3 Precedence Graph
- 6.4 Bernstein's Condition
- 6.5 Time Dependency
- 6.6 Monitors
- 6.7 Answers to Check Your Progress Questions
- 6.8 Summary
- 6.9 Key Words
- 6.10 Self Assessment Questions and Exercises
- 6.11 Further Readings

BLOCK III: DEADLOCK AND INFINITE POSTPONEMENT

UNIT 7 INTRODUCTION TO DEADLOCKS

127-140

- 7.0 Introduction
- 7.1 Objectives
- 7.2 What is Deadlock?
- 7.3 Deadlock Characterization
 - 7.3.1 Example of Deadlock Conditions
- 7.4 Indefinite Postponement
- 7.5 Resource Concept
- 7.6 Answers to Check Your Progress Questions
- 7.7 Summary
- 7.8 Key Words
- 7.9 Self Assessment Questions and Exercises
- 7.10 Further Readings



UNIT 8 CONDITIONS FOR DEADLOCK **141-159**

- 8.0 Introduction
- 8.1 Objectives
- 8.2 Handling Deadlocks
- 8.3 Deadlock Prevention
- 8.4 Deadlock Avoidance
- 8.5 Deadlock Detection
- 8.6 Deadlock Recovery
- 8.7 Answers to Check Your Progress Questions
- 8.8 Summary
- 8.9 Key Words
- 8.10 Self Assessment Questions and Exercises
- 8.11 Further Readings

UNIT 9 PROCESS SCHEDULING **160-193**

- 9.0 Introduction
- 9.1 Objectives
- 9.2 Concept of Process Scheduling
- 9.3 Operations on Processes
- 9.4 Scheduling Concepts
- 9.5 Scheduling Criteria
- 9.6 Scheduling Algorithms
- 9.7 Answers to Check Your Progress Questions
- 9.8 Summary
- 9.9 Key Words
- 9.10 Self Assessment Questions and Exercises
- 9.11 Further Readings

BLOCK IV: REAL MEMORY AND VIRTUAL MEMORY MANAGEMENT

UNIT 10 MEMORY MANAGEMENT **194-217**

- 10.0 Introduction
- 10.1 Objectives
- 10.2 Memory Management
 - 10.2.1 Address Binding
 - 10.2.2 Contiguous Allocation
 - 10.2.3 Dynamic Loading and Linking
 - 10.2.4 Overlay
- 10.3 Memory Organization in Operating System
- 10.4 Memory Hierarchy
- 10.5 Memory Management Strategies
- 10.6 Answers to Check Your Progress Questions
- 10.7 Summary
- 10.8 Key Words
- 10.9 Self Assessment Questions and Exercises
- 10.10 Further Readings





UNIT 11 MEMORY MANAGEMENT STRATEGIES **218-242**

- 11.0 Introduction
- 11.1 Objectives
- 11.2 Preliminaries
- 11.3 Contiguous Memory Allocation
- 11.4 Non-Contiguous Memory Allocation
- 11.5 Answers to Check Your Progress Questions
- 11.6 Summary
- 11.7 Key Words
- 11.8 Self Assessment Questions and Exercises
- 11.9 Further Readings

UNIT 12 VIRTUAL MEMORY MANAGEMENT **243-285**

- 12.0 Introduction
- 12.1 Objectives
- 12.2 Allocation
- 12.3 Address Binding, Relocation and Protection
- 12.4 Swapping
- 12.5 Segmentation
- 12.6 Virtual Memory
 - 12.6.1 Basic Concepts
- 12.7 Logical Versus Physical Address Space
- 12.8 Paging
 - 12.8.1 Page Size and Page Realease
- 12.9 Page Table and Its Entries
- 12.10 Page Fault
 - 12.10.1 Demand Paging
 - 12.10.2 Multilevel Page Table
- 12.11 Page Replacement
- 12.12 Answers to Check Your Progress Questions
- 12.13 Summary
- 12.14 Key Words
- 12.15 Self Assessment Questions and Exercises
- 12.16 Further Readings

BLOCK V: DISK PERFORMANCE AND FILE, DATABASE SYSTEMS

UNIT 13 DISK SCHEDULING **286-306**

- 13.0 Introduction
- 13.1 Objectives
- 13.2 Concept of Disk Management
- 13.3 Disk Structure
- 13.4 Disk Scheduling
- 13.5 Disk Scheduling Algorithms
 - 13.5.1 FCFS Scheduling
 - 13.5.2 SSTF Scheduling
 - 13.5.3 SCAN Scheduling
 - 13.5.4 C-SCAN Scheduling
 - 13.5.5 LOOK Scheduling

- 13.6 Rotaional Optimization
- 13.7 Answers to Check Your Progress Questions
- 13.8 Summary
- 13.9 Key Words
- 13.10 Self Assessment Questions and Exercises
- 13.11 Further Readings

UNIT 14 FILE AND DATABASE SYSTEM

307-350

- 14.0 Introduction
- 14.1 Objectives
- 14.2 File Concept
 - 14.2.1 File Attributes; 14.2.2 File Operations
 - 14.2.3 File Types; 14.2.4 File Structure
- 14.3 Access Methods
 - 14.3.1 Sequential Access; 14.3.2 Direct Access
- 14.4 Data Hierarchy
 - 14.4.1 Purpose of the Data Hierarchy
 - 14.4.2 Components of the Data Hierarchy
- 14.5 Directory Structure
 - 14.5.1 Single-Level Directory
 - 14.5.2 Two-Level Directory
 - 14.5.3 Hierarchical Directory
- 14.6 Protection
 - 14.6.1 Types of Access; 14.6.2 Access Control
- 14.7 File System Structure
 - 14.7.1 File System Implementation
- 14.8 Directory Implementation
 - 14.8.1 Linear List; 14.8.2 Hash Table
- 14.9 File Optimization
 - 14.9.1 7 Things You Should know about File Optimization
 - 14.9.2 How File Optimization Works
- 14.10 Allocation Methods
 - 14.10.1 Contiguous Allocation
 - 14.10.2 Linked Allocation
 - 14.10.3 Indexed Allocation
- 14.11 Free Space Management
 - 14.11.1 Bit Vector; 14.11.2 Linked List
 - 14.11.3 Grouping; 14.11.4 Counting
- 14.12 Efficiency and Performance
 - 14.12.1 Efficiency; 14.12.2 Performance
- 14.13 Recovery
 - 14.13.1 Backup and Restore; 14.13.2 Consistency Checking
- 14.14 File Access Control
 - 14.14.1 Setting Permissions
- 14.15 Answers to Check Your Progress Questions
- 14.16 Summary
- 14.17 Key Words
- 14.18 Self Assessment Questions and Exercises
- 14.19 Further Readings

INTRODUCTION

An Operating System (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs. Characteristically, an Operating System or OS is referred as an interface between a computer user and computer hardware. An operating system is a software which typically performs all the basic tasks, such as file management, memory management, process management, handling input and output, and controlling peripheral devices, for example the disk drives and the printers. Some popular Operating Systems include Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc.

As per the definition, “An Operating System or OS is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs”.

There are various types of operating systems which are precisely scheduled to perform different specified tasks. Time-sharing operating systems, for example have scheduled tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, printing, and other resources. For hardware functions, such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware, although the application code is usually executed directly by the hardware and frequently makes system calls to an operating system function or is interrupted by it. Operating systems are found on many devices that contain a computer – from cellular phones and video game consoles to web servers and supercomputers. Consequently, an operating system is the most essential and vital part of any computer system.

Process scheduling is an operating system task that schedules processes of different states like ready, waiting, and running. Process scheduling allows operating system to allocate a time interval of CPU execution for each process. Another important reason for using a process scheduling system is that it keeps the CPU (Central Processing Unit) busy all the time.

In concurrent computing, a deadlock is a state in which each member of a group waits for another member, including itself, to take action, such as sending a message or more commonly releasing a lock. Deadlock is a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to arbitrate shared resources and implement process synchronization. In an operating system, a deadlock occurs when a process or thread enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process,

NOTES

NOTES

then the system is said to be in a deadlock. Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

This book, Operating Systems, is divided into five blocks, which are further subdivided into fourteen units. The topics discussed include operating systems, operating system architecture, process concepts, process states, process management, interrupts, interprocess communication, mutual exclusion, implementing mutual exclusion primitives, software solution to the mutual exclusion problem, hardware solution to mutual exclusion problem, semaphores, concurrent programming, deadlock, indefinite postponement, resource concepts, conditions for deadlock, avoidance with Dijkstra's banker algorithm, deadlock detection, recovery, processor scheduling, scheduling priorities, scheduling criteria, scheduling algorithms, memory organization, memory management, hierarchy, contiguous vs. non-contiguous memory allocation, fixed partition multiprogramming, variable partition multiprogramming, virtual memory management, page replacement, strategies, page fault frequency, page replacement, page release, page size, disk scheduling strategies, rotational optimization, file and database system, data hierarchy, files, file systems, file optimization, file allocation, free space management, and file access control.

The book follows the Self-Instructional Mode (SIM) wherein each unit begins with an 'Introduction' to the topic. The 'Objectives' are then outlined before going on to the presentation of the detailed content in a simple and structured format. 'Check Your Progress' questions are provided at regular intervals to test the student's understanding of the subject. 'Answers to Check Your Progress Questions', a 'Summary', a list of 'Key Words', and a set of 'Self-Assessment Questions and Exercises' are provided at the end of each unit for effective recapitulation. This book provides a good learning platform to the people who need to be skilled in the area of operating system functions. Logically arranged topics, relevant examples and illustrations have been included for better understanding of the topics.

BLOCK - I

INTRODUCTION

*Introduction to
Operating Systems*

UNIT 1 INTRODUCTION TO OPERATING SYSTEMS

NOTES

Structure

- 1.0 Introduction
 - 1.1 Objectives
 - 1.2 Operating Systems – Definition, Purpose and Goals
 - 1.3 Components of An Operating System
 - 1.4 Evolution of Operating System
 - 1.5 Architecture of Operating Systems
 - 1.6 Answers to Check Your Progress Questions
 - 1.7 Summary
 - 1.8 Key Words
 - 1.9 Self Assessment Questions and Exercises
 - 1.10 Further Readings
-

1.0 INTRODUCTION

An Operating System (OS) is an interface between a computer user and the computer hardware. There are different types of operating systems which provide common services for computer programs. Fundamentally, an operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices, such as disk drives and printers. Consequently, an operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware. An additional universal definition is that the operating system is the single unique program that runs continuously on the computer and supports the various types of compatible application programs.

The computer system is a combination of many parts, such as I/O (Input/Output) devices, secondary memory, CPU (Central Processing Unit), etc. In a computer system, the I/O devices include monitor, mouse, keyboards, touch pad, disk drives, display adapters, USB (Universal Serial Bus) devices, bit-mapped screen, LED (Light Emitting Diode), Analog-to-Digital Converter (ADC), ON/OFF switch, network connections, audio I/O, printers, device drivers, etc. An I/O system is required to take instructions/request from the I/O application and send it to the respective physical devices, then process the response whatever comes back from the respective devices and send it back to the application. Device

NOTES

drivers are software modules that can be plugged into an operating system to handle a specific or particular computer device. Basically, the operating system takes help from device drivers for handling all the I/O devices.

All the components of an operating system altogether make the different parts of a computer to work in organised manner. All user software needs to go through the operating system in order to use any of the hardware, whether it be as simple as a mouse or keyboard or as complex as an Internet component. A Central Processing Unit (CPU), also called a central processor, main processor or just processor, is the electronic circuitry within a computer that executes instructions that make up a computer program. The CPU performs basic arithmetic, logic, controlling, and Input/Output (I/O) operations specified by the instructions in the program. Most modern CPUs are implemented on Integrated Circuit (IC) microprocessors, with one or more CPUs on a single Metal-Oxide-Semiconductor (MOS) IC chip.

Operating systems have evolved from slow and expensive systems to present-day technology where computing power has reached exponential speeds and relatively inexpensive costs. Some popular operating systems include UNIX Operating System, Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc. Microsoft Windows is a family of proprietary operating systems designed by Microsoft Corporation and primarily targeted to Intel architecture based computers. Server editions of Windows are widely used. The latest version is Windows 10.

In this unit, you will study about the operating system concept, definition, purpose, goals, evolution and architecture of the operating system.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basic concept of an operating system
- Explain the functions, goals and purpose of an operating system
- Define the components of the operating system
- Discuss about the evolution of the operating system
- Elaborate on the concept of serial processing, batch processing and multiprogramming
- Explain the architecture of an operating system

1.2 OPERATING SYSTEMS – DEFINITION, PURPOSE AND GOALS

An Operating System (OS) is an interface between a computer user and computer hardware. Fundamentally, an operating system is a specific unique program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs. Characteristically, an operating system is a system software which performs all the basic tasks like managing software resources, file management, memory management, process management, handling input and output, managing and controlling computer hardware and peripheral devices, such as disk drives and printers.

Consequently, an operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware. An additional universal definition is that the operating system is the single unique program that runs continuously on the computer and supports the various types of compatible application programs. The principle task of an operating system is concerned with the allocation and management of resources and services, such as a traffic controller, a scheduler, the memory management module, the I/O programs, and a file system. Management tasks include scheduling of resources so that the conflicts and interference between the programs can be avoided. Most of the application programs that run on the computer system get terminated after completing the specified task or as per the instructions specified by the user, whereas an operating system runs for an indefinite period or till further information and terminates only when the computer system is turned off.

There are different types of operating systems which provide common services for computer programs. Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, printing, and other resources. For hardware functions, such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware, although the application code is usually executed directly by the hardware and frequently makes system calls to an OS function or is interrupted by it. Operating systems are found on many devices that contain a computer – from cellular phones and video game consoles to web servers and supercomputers.

The dominant desktop operating system is Microsoft Windows, MacOS by Apple Inc. and the varieties of Linux. In the mobile sector (including smartphones and tablets), Android's type of operating systems are used. Linux distributions are dominant in the server and supercomputing sectors. Other specialized classes of operating systems, such as embedded and real-time systems, exist for many applications. Some popular operating systems include UNIX Operating System,

NOTES

NOTES

Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc. Microsoft Windows is a family of proprietary operating systems designed by Microsoft Corporation and primarily targeted to Intel architecture based computers. Server editions of Windows are widely used. The latest version is Windows 10.

Definition

An operating system is a program that acts as an intermediary between the user of a computer and the computer hardware.

Purpose and Goals

- Executing user programs
- Making the computer system convenient to use
- Using computer hardware efficiently

Functions of an Operating System

An operating system is used to manage user and system programs, and hardware such as CPU (Central Processing Unit), memory and peripherals. It performs scheduling of processes by executing the process with high priority. It also protects the resources by performing authorization, authentication and accounting. It allows a process to communicate with other processes.

An operating system can operate in two modes to protect it from other system components. They are the kernel mode, where the OS gets privileges to control the entire system components, and the user mode, where the components are under the user program and only those privileges that do not affect the OS are granted.

1.3 COMPONENTS OF AN OPERATING SYSTEM

The components of an operating system all exist in order to make the different parts of a computer work together. All user software needs to go through the operating system in order to use any of the hardware, whether it be as simple as a mouse or keyboard or as complex as an Internet component.

The operating system provides an interface between an application program and the computer hardware, so that an application program can interact with the hardware only by obeying rules and procedures programmed into the operating system. The operating system is also a set of services which simplify development and execution of application programs. Executing an application program involves the creation of a process by the operating system kernel which assigns memory space and other resources, establishes a priority for the process in multi-tasking systems, loads program binary code into memory, and initiates execution of the application program which then interacts with the user and with hardware devices.

NOTES

A computer system consists of the following four components:

- **Hardware:** Various hardware devices, such as processor, memory, input and output.
- **Operating System:** This coordinates the usage of the hardware by the application programs.
- **Application Programs:** They include word processor, editors, compilers, Web browsers, etc., through which the user interacts with the computer system.
- **Users:** User of the computer.

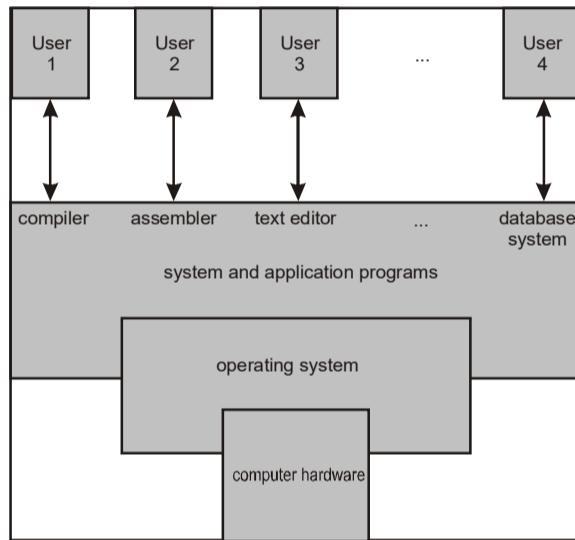


Fig. 1.1 Four Components of a Computer System

Figure 1.1 explains graphically how, when the user wants to work with the computer, he or she opens an application program. The application program cannot use the hardware directly; instead, it uses the operating system as an interface to interact with the computer hardware. It is clear that the operating system is acting as an intermediary between the application program and the computer hardware. Without the operating system, application programs cannot be executed.

1.4 EVOLUTION OF OPERATING SYSTEM

Evolution mean the gradual development of something. From sluggish and costly systems to present-day technology, operating systems have developed where computing power has achieved exponential speeds and relatively inexpensive costs. In the beginning, machines were manually loaded with programme code to monitor the functions of the machine and business logic-related process code.

Operating system evolution is directly based on the advancement of computer systems and how they are used by users. Here is a brief tour of the timeline of computer systems over the past fifty years.

NOTES

Processing Types

A processor is used to process jobs. A job consists of a set of instructions. Jobs can be processed in the following different ways:

Serial Processing

In serial processing, all the jobs are processed serially one after the other. If a job is waiting for some event, then all the other jobs have to wait till it is completed. Once the waiting job completes, the next job in the queue starts execution.

Batch Processing

This type of processing was used when there was no disk technology. There were two types of computers in the 1960s—slow processor computer and fast processor computer. Slow processors were used to read the input from card readers and store them on tape drives. A tape drive can store several programs, also called batch of jobs. Fast computers were used to execute the batch of jobs written by the slow processors and generated the output onto another tape drive. In batch processing, several programs are batched together on to a tape drive and given for execution. The CPU reads each batch of jobs and executes them. The output is written onto another tape drive. Finally, the output tape is given to the programmer.

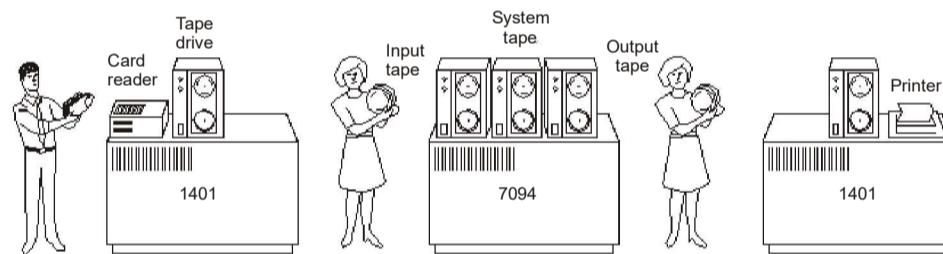


Fig. 1.2 Batch Systems

Multiprogramming

A process needs a lot of CPU time and I/O (Input/Output) time. If a process needs more CPU time, then it is a CPU-bound process. If it needs more time for I/O, then it is an I/O-bound process. When a system has more I/O-bound processes and if a process is waiting for I/O, the CPU sits idle till the I/O is complete. The main goal of an OS is to utilize the CPU to its maximum. So in multiprogramming, if one process is waiting for the I/O to complete, then the CPU can execute another process. The advantage of multiprogramming is that it does not allow the CPU to sit idle; instead, it executes another process. In a multiprogramming system, the switching of jobs is done very slowly so that the user cannot interact with the system.

Types of Operating Systems

*Introduction to
Operating Systems*

Operating systems are evolving from the early 1950s. The chronological order of their evolution is given as follows:

- 1950s** Serial processing operating systems were used. The utilization of the CPU was very low.
- 1960s** Batch operating systems evolved that executed various batches of jobs. The utilization of CPU improved. For example, mainframe systems.
- 1970s** Multiprogrammed operating systems were developed where the CPU usage was multiplexed among various jobs present in the memory. When a job was waiting for I/O, the CPU executed other jobs. Here, the CPU was not allowed to sit idle. The CPU resource was utilized to the maximum. Time-sharing systems also evolved during this decade. They were an extension of multiprogrammed systems. Here, the processor time was shared among the jobs available in the memory. This facilitated interactive computing. In a time-sharing system, the switching of jobs was done very fast and each job was given CPU time for a fraction of a second. The response time in these systems was very short. Time-sharing systems were also called multitasking systems.
- 1980s** Desktop systems or personal computers evolved, which concentrated on user convenience and responsiveness rather than improving the utilization of the CPU. Multiprocessor systems were introduced to facilitate parallel computing. These systems increased throughput (the number of jobs executed per unit time) and reliability. Distributed systems were developed. These operating systems facilitated network communication like LAN, MAN and WAN.
- 1990s** Real-time systems were introduced. This operating system put a time constraint on the operation of the processor and flow of data.

NOTES

Check Your Progress

1. Define the term operating system.
2. Give the definition of operating system.
3. What are the purpose and goals of an operating system?
4. State the two modes in which an operating system can operate.
5. Explain about the components of an operating system.
6. Why a processor is used? Define serial processing.
7. Define the terms CPU-bound process and I/O-bound process.
8. What is multiprogramming?

NOTES

1.5 ARCHITECTURE OF OPERATING SYSTEMS

The operating system provides applications with a virtual machine. This type of situation is analogous to the communication line of a telephony company, which enables separate and isolated conversations over the same wire(s). An important aspect of such a system is that the user can run an operating system of his/her choice.

The virtual machine concept can be well understood by understanding the difference between conventional multiprogramming and virtual machine multiprogramming. In conventional multiprogramming, processes are allocated a portion of the real machine resources that is a resource from the same machine is distributed among several resources. Figure 1.3 shows the process of conventional programming.

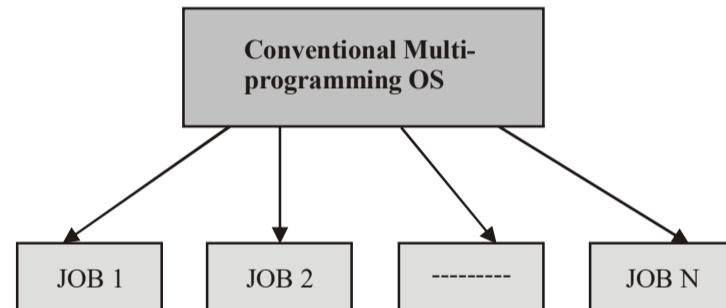


Fig. 1.3 Conventional Multiprogramming

In the virtual machine multiprogramming system, a single machine gives an illusion of many virtual machines, each of them having its own virtual processor and storage space which can be handled through process scheduling. Figure 1.4 shows the process of virtual machine programming.

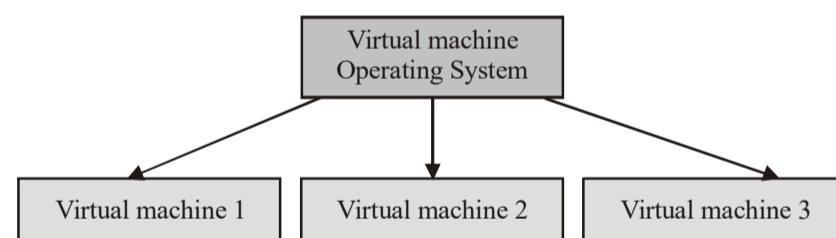


Fig. 1.4 Virtual Machine Multiprogramming

Advantages

The advantages of the virtual machine multiprogramming system are as follows:

- Each user is allocated with a machine which eliminates mutual interference between users.

- A user can select an OS of his choice for executing the virtual machine. Hence, the user can simultaneously use different operating systems on the same computer system.

Kernel Approach

In this approach:

- The kernel lies below the system call interface and above the physical hardware.
- It provides large number of functions, such as CPU scheduling, memory management, I/O management, synchronization of processes, inter-process communication and other operating system functions.

Figure 1.5 shows the structure of a computer system.

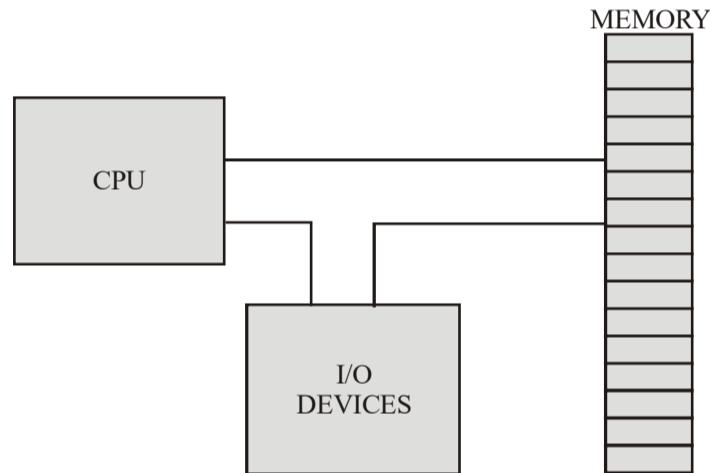


Fig. 1.5 Structure of a Computer System

Central Processing Unit (CPU)

The Arithmetic and Logical Unit (ALU) and the Control Unit (CU) of a computer system are jointly known as the Central Processing Unit or the CPU. You may call CPU as the brain of a computer system. It takes all major decisions, makes all sorts of calculations and directs different parts of the computer functions by activating and controlling the operations.

For a computer to start running, it needs to have an initial program to run. This initial program, also known as bootstrap program, tends to be simple. It is stored in CPU registers. The role of the initial program or the bootstrap program is to load the operating system for the execution of the system. The operating system starts executing the first process, such as ‘init’, and waits for some event to occur. Event is known to occur by an *interrupt* from either the hardware or the software. Hardware can interrupt through *system bus*, whereas software through *system call*.

NOTES

NOTES

When a CPU is interrupted, it immediately stops whatever it is doing and returns to a fixed location. This fixed location usually contains the starting address where the service routine for the interrupt is located.

Input/Output (I/O) Devices

There are various types of I/O devices that are used for different types of applications. They are also known as peripheral devices because they surround the CPU and enable a communication between the computer and the outer world.

Input devices: Input devices are necessary to convert information or data into a form, which can be understood by the computer. A good input device should provide timely, accurate and useful data to the main memory of the computer for processing. Keyboard, mouse and scanner are the most useful input devices.

Output devices: Visual Display Unit (VDU), terminals and printers are the most commonly used output devices.

Layered Architecture of an Operating System

Architecture is the fundamental structure of an operating system that defines interconnection between the system components. An operating system is designed using the following architectures:

1. Monolithic architecture
2. Layered architecture
3. Virtual machine architecture
4. Exokernel architecture
5. Client-server architecture

Monolithic and Layered Architecture

Monolithic architecture consists of a single layer that performs all the functions of the operating system. The concept of information hiding is altogether absent in the monolithic architecture, therefore you are able to observe and call the procedures of different users. MS-DOS (Microsoft Disk Operating System) and Novell Netware operating systems are the examples of monolithic architecture.

In monolithic architecture, operating system provides services to the users in the form of system calls. These system calls provide functions, such as positioning of procedural parameters on the stack and executing trap instruction. This instruction transfers the control to the operating system by swapping the control from user mode to kernel mode. Figure 1.6 shows the monolithic architecture of an operating system.

NOTES

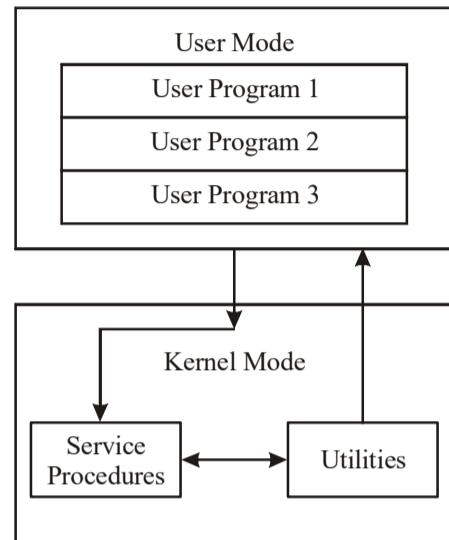


Fig. 1.6 The Monolithic Architecture

The operating system checks the parameters and calls the service procedure, which in turn calls the utilities. Utilities return the control back to the user program in user mode. The flow of arrows shows the transfer of user program from user mode to kernel mode through system calls. The limitations of an operating system using monolithic architecture are as follows:

- (i) Difficult to modify, as the whole operating system has to be redesigned.
- (ii) Failure of single program crashes the entire system.

Layered architecture categorizes the operating system into different layers that communicate using standard function calls. Each new layer is built on the top of an older layer.

The higher-level layers call the set of functions and data structures of lower layers. The various layers of operating system are as follows:

- (i) Hardware
- (ii) Kernel
- (iii) Service layer
- (iv) Applications/shell

The hardware layer consists of various I/O devices, CPU and memory. The hardware acts as a platform for an operating system. It provides computing resources for executing various application programs.

Kernel is the core of all the operating systems. Kernel directly interacts with the hardware and schedules the execution of various tasks. For example, in a multiple processor operating system, kernel decides the processor and the program to be chosen for execution.

Kernel performs low-level functions such as reading the input from keyboard and displaying the output on the monitor. In addition to these low-level functions,

NOTES

kernel, which is also called real-time executive, performs other important functions such as memory management, file management, scheduling and system accounting of various processes.

The service layer interacts with the kernel and the drivers. Drivers are a set of specialized programs used by the programs to communicate with the hardware. The service layer is responsible for maintaining the security and sanity checking of user's files and objects. For example, if a program is not currently in use, the service layer gives the permission and informs the kernel for deleting the file.

Sanity checking is used in decision-making for the execution of various jobs. It allows one job to execute and other job to wait until the execution is complete.

A user creates different applications and executes these applications through shells. A shell is a utility that is stored on the hard disk and is loaded into the memory when the kernel is invoked. It acts as an interface between the user and the operating system. Shell is also known as command line interpreter. Figure 1.7 shows the layered architecture of an operating system.

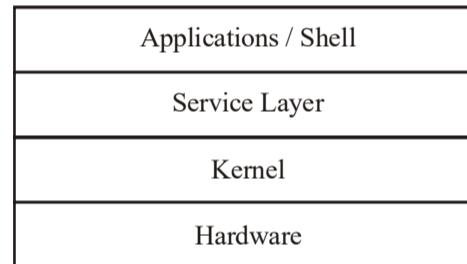


Fig. 1.7 The Layered Architecture

An operating system with the layered architecture provides the following advantages:

- Maintains the modular approach of the system by decomposing it into several layers.
- Simplifies the task of debugging and system verification. The bottom layer consists of basic hardware that is easy to debug. If an error appears in a particular layer, then only that layer is modified as lower layers are already debugged.
- Implements the concept of information hiding. The contents of lower-level layers, such as hardware, data structures and implementation of operations are hidden from the higher-level layers. Only the functioning of lower layer operations are visible to higher layers.

In addition to several advantages, the layered architecture has the following disadvantages:

- Less efficient than other architectures as each layer adds an operating cost to its higher layer and therefore, takes longer time to complete a simple task.

- Requires cautious planning for designing the layers, because a higher layer can use the functionalities of only the lower layers.

Virtual Machine and Exokernel Architecture

Virtual machine and exokernel architecture, both are based on the principle of creating duplicate copies of operating systems. These copies are concurrently executed on the same hardware platform.

Virtual machine uses the concept of CPU scheduling and virtual memory for creating a false impression of multiple processors. These multiple processors are executed concurrently in their virtual memory on the same hardware platform. Each processor consists of a set of virtual instructions. The function of a virtual machine is to map the virtual instruction set to real instruction set of the computer. Instruction set consists of commands built in an assembly language, such as JMP and MOVE. The function of instruction set is to call and execute these commands.

Virtual machine works like an actual physical computer. It is used in various concepts, such as partitioning the hardware and sharing it among different programs, creating the portable software and executing older version of software on computers with latest configuration.

Virtual machine contains a software layer called Virtual Machine Monitor (VMM) that creates the exact copies of hardware resources and exports the hardware interface. VM/370 operating system is an example of virtual machine architecture. It supports multiple virtual machines that run concurrently on IBM/370 computer. Figure 1.8 shows the virtual machine architecture.

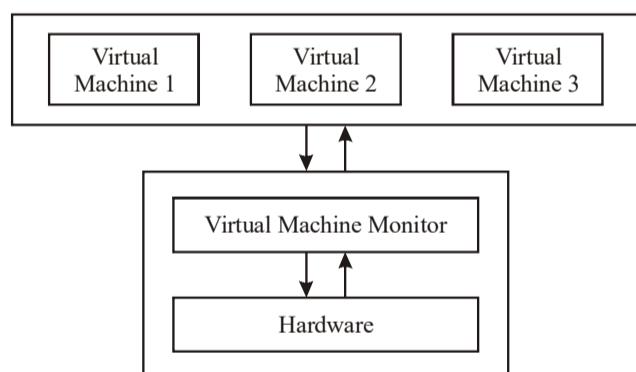


Fig. 1.8 The Virtual Machine Architecture

The advantage of using virtual architecture is that it facilitates the sharing of hardware resources. The limitation of using virtual machine is that virtual-machine software requires a lot of disk space for providing virtual memory and spooling.

Exokernel architecture facilitates the secure sharing of hardware resources by providing a unique application-specific customization from traditional operating systems abstractions, such as file system storage and exception handling. The traditional operating systems lack the flexibility, performance and functionality of

NOTES

NOTES

applications. At application level, exokernel provides software for implementing the traditional operating systems features.

The exokernel architecture contains a program called exokernel that resides in the bottom layer of the operating system. Exokernel performs the function of resource allocation to virtual machines and maintains authorised and secure use of their resources.

The advantage of using exokernel architecture is that it eliminates the requirement of the mapping layer. It not only monitors the resources allocated to virtual machines, but also separates the system code running in user space from the multiprogramming concept of exokernel.

Client–Server Architecture

Client–server architecture consists of two different computer programs—client and server. Client sends a service request to the server for accessing various resources. The server fulfils the request by providing shared access to resources such as printer, database and modem. Client–server architecture is mainly used in a network for efficient distribution of resources across different clients.

You can use Application Program Interface (API) and Remote Procedure Calls (RPC) for communicating between client and server processes. API is a set of software functions implemented as a library used by an application program as a means for providing access to an operating system's capabilities. RPC is a method of calling the procedure located at remote location by sending service request to the server. The components of client–server architecture are:

- (i) Client
- (ii) Network
- (iii) Server

Client contains components, such as Graphical User Interface (GUI), DataBase Management System (DBMS), communication package and operating system. Clients having GUI package creates a user-friendly interface with the server. DBMS communication package is used for communicating with the server database.

Network is used for communicating the clients with the server. A network can be LAN (Local Area Network), WAN (Wide Area Network), or the Internet. The TCP/IP (Transmission Control Protocol/Internet Protocol) protocol is the most commonly used protocol for communication between various users.

Server contains various components, such as authentication module, network communication package, database package and operating system. Authentication module verifies various clients using passwords. Database package contains various protocols for accessing the database that includes protecting the information from users by revoking permissions for reading and writing records. Figure 1.9 shows the client–server architecture.

NOTES

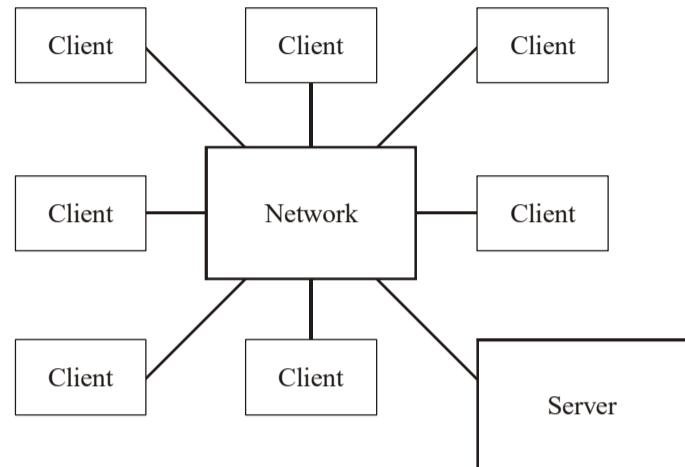


Fig. 1.9 The Client–Server Architecture

The advantage of using client–server architecture is that it provides flexibility and scalability for adding essential resources in the server computer afterwards. Since, all the computers are connected to the server and new resource does not need to be configured to the client computers.

The disadvantage of using client–server architecture is that it has complex configuration that requires greater expertise, which results in higher cost. It is less reliable because of the availability of large number of clients. This makes the system dependable on different clients.

Check Your Progress

9. Define the concept of conventional multiprogramming and virtual machine multiprogramming.
10. Elaborate on the kernel approach.
11. What is the significance of CPU?
12. Explain about the layered architecture of an Operating System (OS).
13. Define the term monolithic architecture. What are the limitations of an operating system that uses the monolithic architecture?
14. Why the layered architecture categorizes the operating system into different layers? Define the various layers of the operating system.
15. Explain the concept of virtual machine and exokernel architecture.
16. What is client-server architecture?
17. Which programs are used for communicating between client and server processes?
18. Define the components of client-server architecture.

NOTES

1.6 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. An Operating System (OS) is an interface between a computer user and computer hardware, i.e., an operating system is a specific unique program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs. Characteristically, an operating system is a system software which performs all the basic tasks like managing software resources, file management, memory management, process management, handling input and output, managing and controlling computer hardware and peripheral devices, such as disk drives and printers.
2. An operating system is a program that acts as an intermediary between the user of a computer and the computer hardware.
3. Following are the purpose and goals of an operating system:
 - Executing user programs
 - Making the computer system convenient to use
 - Using computer hardware efficiently
4. An operating system can operate in two modes to protect it from other system components. They are the kernel mode, where the OS gets privileges to control the entire system components, and the user mode, where the components are under the user program and only those privileges that do not affect the OS are granted.
5. The components of an operating system all exist in order to make the different parts of a computer work together. A computer system consists of the following four components:

Hardware: Various hardware devices, such as processor, memory, input and output.

Operating System: This coordinates the usage of the hardware by the application programs.

Application Programs: They include word processor, editors, compilers, Web browsers, etc., through which the user interacts with the computer system.

Users: User of the computer.
6. A processor is used to process jobs. A job consists of a set of instructions. In serial processing, all the jobs are processed serially one after the other. If a job is waiting for some event, then all the other jobs have to wait till it is completed. Once the waiting job completes, the next job in the queue starts execution.

NOTES

7. A process needs a lot of CPU time and I/O (Input/Output) time. If a process needs more CPU time, then it is CPU-bound process. If it needs more time for I/O, then it is an I/O-bound process. When a system has more I/O-bound processes and if a process is waiting for I/O, the CPU sits idle till the I/O is complete.
8. The main goal of an OS is to utilize the CPU to its maximum. So in multiprogramming, if one process is waiting for the I/O to complete, then the CPU can execute another process. The advantage of multiprogramming is that it does not allow the CPU to sit idle; instead, it executes another process. In a multiprogramming system, the switching of jobs is done very slowly so that the user cannot interact with the system.
9. In conventional multiprogramming, processes are allocated a portion of the real machine resources that is a resource from the same machine is distributed among several resources. In the virtual machine multiprogramming system, a single machine gives an illusion of many virtual machines, each of them having its own virtual processor and storage space which can be handled through process scheduling.
10. The kernel lies below the system call interface and above the physical hardware. It provides large number of functions, such as CPU scheduling, memory management, I/O management, synchronization of processes, inter-process communication and other operating system functions.
11. The Arithmetic and Logical Unit (ALU) and the Control Unit (CU) of a computer system are jointly known as the Central Processing Unit (CPU). The CPU is also termed as the brain of a computer system. It takes all major decisions, makes all sorts of calculations and directs different parts of the computer functions by activating and controlling the operations.
12. Architecture is the fundamental structure of an operating system that defines interconnection between the system components. An operating system is designed using the following architectures:
 - Monolithic Architecture
 - Layered Architecture
 - Virtual Machine Architecture
 - Exokernel Architecture
 - Client–Server Architecture
13. Monolithic architecture consists of a single layer that performs all the functions of the operating system. The concept of information hiding is altogether absent in the monolithic architecture, therefore you are able to observe and call the procedures of different users. In monolithic architecture, the operating system provides services to the users in the form of system calls. These system calls provide functions, such as positioning of procedural parameters

NOTES

on the stack and executing trap instruction. This instruction transfers the control to the operating system by swapping the control from user mode to kernel mode.

The limitations of an operating system, using monolithic architecture are as follows:

- Difficult to modify, as the whole operating system has to be redesigned.
 - Failure of single program crashes the entire system.
14. Layered architecture categorizes the operating system into different layers that communicate using standard function calls. Each new layer is built on the top of an older layer. The higher-level layers call the set of functions and data structures of lower layers. The various layers of operating system are as follows:
- Hardware
 - Kernel
 - Service Layer
 - Applications/Shell
15. Virtual machine and exokernel architecture, both are based on the principle of creating duplicate copies of operating systems. These copies are concurrently executed on the same hardware platform.
Virtual machine uses the concept of CPU scheduling and virtual memory for creating a false impression of multiple processors. These multiple processors are executed concurrently in their virtual memory on the same hardware platform. Each processor consists of a set of virtual instructions. The function of a virtual machine is to map the virtual instruction set to real instruction set of the computer. Virtual machine works like an actual physical computer. Virtual machine contains a software layer called Virtual Machine Monitor (VMM) that creates the exact copies of hardware resources and exports the hardware interface.
Exokernel architecture facilitates the secure sharing of hardware resources by providing a unique application-specific customization from traditional operating systems abstractions, such as file system storage and exception handling. The traditional operating systems lack the flexibility, performance and functionality of applications. At application level, exokernel provides software for implementing the traditional operating systems features. The exokernel architecture contains a program called exokernel that resides in the bottom layer of the operating system. Exokernel performs the function of resource allocation to virtual machines and maintains authorised and secure use of their resources.
16. Client–server architecture consists of two different computer programs—client and server. Client sends a service request to the server for accessing

NOTES

various resources. The server fulfils the request by providing shared access to resources such as printer, database and modem. Client–server architecture is mainly used in a network for efficient distribution of resources across different clients.

17. The programs Application Program Interface (API) and Remote Procedure Calls (RPC) are used for communicating between client and server processes. API is a set of software functions implemented as a library used by an application program as a means for providing access to an operating system's capabilities. RPC is a method of calling the procedure located at remote location by sending service request to the server.
18. Following are the components of client–server architecture:

Client: Client contains components, such as Graphical User Interface (GUI), DataBase Management System (DBMS), communication package and operating system. Clients having GUI package creates a user-friendly interface with the server. DBMS communication package is used for communicating with the server database.

Network: Network is used for communicating the clients with the server. A network can be LAN (Local Area Network), WAN (Wide Area Network), or the Internet. The TCP/IP (Transmission Control Protocol/Internet Protocol) protocol is the most commonly used protocol for communication between various users.

Server: Server contains various components, such as authentication module, network communication package, database package and operating system.

1.7 SUMMARY

- An Operating System (OS) is an interface between a computer user and computer hardware.
- Fundamentally, an operating system is a specific unique program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.
- Characteristically, an operating system is a system software which performs all the basic tasks like managing software resources, file management, memory management, process management, handling input and output, managing and controlling computer hardware and peripheral devices, such as disk drives and printers.
- An additional universal definition is that the operating system is the single unique program that runs continuously on the computer and supports the various types of compatible application programs.
- The principle task of an operating system is concerned with the allocation and management of resources and services, such as a traffic controller, a

NOTES

scheduler, the memory management module, the I/O programs, and a file system.

- There are different types of operating systems which provide common services for computer programs. Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, printing, and other resources.
- The dominant desktop operating system is Microsoft Windows, Mac OS by Apple Inc. and the varieties of Linux. In the mobile sector (including smartphones and tablets), Android's type of operating systems are used. Linux distributions are dominant in the server and supercomputing sectors. Other specialized classes of operating systems, such as embedded and real-time systems, exist for many applications.
- Some popular operating systems include UNIX Operating System, Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc. Microsoft Windows is a family of proprietary operating systems designed by Microsoft Corporation and primarily targeted to Intel architecture based computers. Server editions of Windows are widely used. The latest version is Windows 10.
- An operating system is a program that acts as an intermediary between the user of a computer and the computer hardware.
- An operating system is used to manage user and system programs, and hardware such as CPU, memory and peripherals. It performs scheduling of processes by executing the process with high priority. It also protects the resources by performing authorization, authentication and accounting. It allows a process to communicate with other processes.
- An operating system can operate in two modes to protect it from other system components. They are kernel mode, where the OS gets privileges to control the entire system components, and user mode, where the components are under the user program and only those privileges that do not affect the OS are granted.
- The components of an operating system all exist in order to make the different parts of a computer work together.
- All user software needs to go through the operating system in order to use any of the hardware, whether it be as simple as a mouse or keyboard or as complex as an Internet component.
- The operating system provides an interface between an application program and the computer hardware, so that an application program can interact with the hardware only by obeying rules and procedures programmed into the operating system.

NOTES

- The operating system is also a set of services which simplify development and execution of application programs. Executing an application program involves the creation of a process by the operating system kernel which assigns memory space and other resources, establishes a priority for the process in multi-tasking systems, loads program binary code into memory, and initiates execution of the application program which then interacts with the user and with hardware devices.
- The application program cannot use the hardware directly; instead, it uses the operating system as an interface to interact with the computer hardware. It is clear that the operating system is acting as an intermediary between the application program and the computer hardware. Without the operating system, application programs cannot be executed.
- Evolution mean the gradual development of something. From sluggish and costly systems to present-day technology, operating systems have developed where computing power has achieved exponential speeds and relatively inexpensive costs.
- In the beginning, machines were manually loaded with programme code to monitor the functions of the machine and business logic-related process code.
- Operating system evolution is directly based on the advancement of computer systems and how they are used by users. Here is a brief tour of the timeline of computer systems over the past fifty years.
- A processor is used to process jobs. A job consists of a set of instructions.
- In serial processing, all the jobs are processed serially one after the other. If a job is waiting for some event, then all the other jobs have to wait till it is completed. Once the waiting job completes, the next job in the queue starts execution.
- A process needs a lot of CPU time and I/O time. If a process needs more CPU time, then it is a CPU-bound process. If it needs more time for I/O, then it is an I/O-bound process. When a system has more I/O-bound processes and if a process is waiting for I/O, the CPU sits idle till the I/O is complete.
- The main goal of an OS is to utilize the CPU to its maximum. So in multiprogramming, if one process is waiting for the I/O to complete, then the CPU can execute another process.
- In a time-sharing system, the switching of jobs was done very fast and each job was given CPU time for a fraction of a second. The response time in these systems was very short. Time-sharing systems were also called multitasking systems.

NOTES

- The operating system provides applications with a virtual machine. This type of situation is analogous to the communication line of a telephony company, which enables separate and isolated conversations over the same wire(s). An important aspect of such a system is that the user can run an operating system of his/her choice.
- The virtual machine concept can be well understood by understanding the difference between conventional multiprogramming and virtual machine multiprogramming.
- In conventional multiprogramming, processes are allocated a portion of the real machine resources that is a resource from the same machine is distributed among several resources.
- The kernel lies below the system call interface and above the physical hardware. It provides large number of functions, such as CPU scheduling, memory management, I/O management, synchronization of processes, inter-process communication and other operating system functions.
- The Arithmetic and Logical Unit (ALU) and the Control Unit (CU) of a computer system are jointly known as the Central Processing Unit (CPU). You may call CPU as the brain of a computer system. It takes all major decisions, makes all sorts of calculations and directs different parts of the computer functions by activating and controlling the operations.
- Event is known to occur by an interrupt from either the hardware or the software. Hardware can interrupt through system bus, whereas software through system call.
- When a CPU is interrupted, it immediately stops whatever it is doing and returns to a fixed location. This fixed location usually contains the starting address where the service routine for the interrupt is located.
- There are various types of I/O devices that are used for different types of applications. They are also known as peripheral devices because they surround the CPU and enable a communication between the computer and the outer world.
- Input devices are necessary to convert information or data into a form, which can be understood by the computer. A good input device should provide timely, accurate and useful data to the main memory of the computer for processing. Keyboard, mouse and scanner are the most useful input devices.
- Visual Display Unit (VDU), terminals and printers are the most commonly used output devices.
- Input devices are necessary to convert information or data into a form, which can be understood by the computer. A good input device should provide timely, accurate and useful data to the main memory of the computer for processing. Keyboard, mouse and scanner are the most useful input devices.

NOTES

- Visual Display Unit (VDU), terminals and printers are the most commonly used output devices.
- Architecture is the fundamental structure of an operating system that defines interconnection between the system components.
- Monolithic architecture consists of a single layer that performs all the functions of the operating system. The concept of information hiding is altogether absent in the monolithic architecture, therefore you are able to observe and call the procedures of different users.
- Layered architecture categorizes the operating system into different layers that communicate using standard function calls. Each new layer is built on the top of an older layer.
- Kernel is the core of all the operating systems. Kernel directly interacts with the hardware and schedules the execution of various tasks. For example, in a multiple processor operating system, kernel decides the processor and the program to be chosen for execution.
- A user creates different applications and executes these applications through shells. A shell is a utility that is stored on the hard disk and is loaded into the memory when the kernel is invoked.
- Virtual machine and exokernel architecture, both are based on the principle of creating duplicate copies of operating systems. These copies are concurrently executed on the same hardware platform.
- Virtual machine contains a software layer called Virtual Machine Monitor (VMM) that creates the exact copies of hardware resources and exports the hardware interface.
- Client–server architecture consists of two different computer programs—client and server. Client sends a service request to the server for accessing various resources. The server fulfils the request by providing shared access to resources, such as printer, database and modem.
- Client–server architecture is mainly used in a network for efficient distribution of resources across different clients.
- Client contains components, such as Graphical User Interface (GUI), DataBase Management System (DBMS), communication package and operating system. Clients having GUI package creates a user-friendly interface with the server. DBMS communication package is used for communicating with the server database.
- Network is used for communicating the clients with the server. A network can be LAN (Local Area Network), WAN (Wide Area Network), or the Internet. The TCP/IP (Transmission Control Protocol/Internet Protocol) protocol is the most commonly used protocol for communication between various users.

NOTES

- Server contains various components, such as authentication module, network communication package, database package and operating system. Authentication module verifies various clients using passwords. Database package contains various protocols for accessing the database that includes protecting the information from users by revoking permissions for reading and writing records.

1.8 KEY WORDS

- **Operating system:** The most essential and indispensable program that is running at all times on the computer (usually called the kernel). It is a program that acts as an interface between the computer users and the computer hardware.
- **Processor:** A processor is used to process jobs. A job consists of a set of instructions.
- **Serial processing:** In serial processing, all the jobs are processed serially one after the other. If a job is waiting for some event, then all the other jobs have to wait till it is completed. Once the waiting job completes, the next job in the queue starts execution.
- **Input devices:** Input devices are necessary to convert information or data into a form, which can be understood by the computer. A good input device should provide timely, accurate and useful data to the main memory of the computer for processing. Keyboard, mouse and scanner are the most useful input devices.
- **Output devices:** Visual Display Unit (VDU), terminals and printers are the most commonly used output devices.
- **Multiprogramming:** The number of jobs competing to get the system resources in multiprogramming environment is known as degree of multiprogramming.
- **Conventional multiprogramming:** Process are allocated a portion of the real machine resources that is a resources from the same machine is distributed among several resources.
- **Virtual machine multiprogramming:** A single machine gives an illusion of many virtual machine.
- **Monolithic:** Monolithic architecture consist of a single layer that perform all the functions of operating system.
- **Layered architecture:** Layered architecture categorizes the operating system into different layers that communicate using standard function calls. Each new layer is built on the top of an older layer.
- **Kernel:** Kernel is the core of all the operating systems. Kernel directly interacts with the hardware and schedules the execution of various tasks.

1.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Why an operating system is required?
2. Elaborate on the purpose and goals of operating system.
3. State the functions of an operating system.
4. What are the two modes to protect it from other system components?
5. Explain about the components of an operating system.
6. What is serial and batch processing?
7. Why multiprogramming is required?
8. Define about the processing types of an operating system.
9. Differentiate between the conventional multiprogramming and the virtual machine multiprogramming.
10. Elucidate on the different types of layered architecture.
11. Give the limitations of an operating system that uses the monolithic architecture.
12. Explain about the client-server architecture.

Long-Answer Questions

1. Briefly discuss the concept and significance of the operating system giving its definition purposes and goals.
2. Explain about the components of operating system with the help of diagrams.
3. What are the components of a computer system? Explain the role of an operating system in relation to these components.
4. Discuss about the significance of processor with reference to operating system. Differentiate between serial processing and batch processing giving appropriate examples.
5. Briefly explain about the evolution of operating systems in the chronological order giving examples of each type.
6. In conventional multiprogramming how the processes are allocated? Explain giving example.
7. Describe in detail about the architecture of operating system with the help of examples and diagrams.
8. Discuss about the virtual machine concept architecture and advantages of an operating system giving appropriate examples and diagrams.

NOTES

NOTES

9. Elaborate briefly on the kernel approach in operating system with reference to CPU and I/O devices. Support your answer giving examples and diagrams.
10. Explain the significance and need of layered architecture in computer system giving its types and examples.
11. Briefly discuss about the monolithic and layered architecture concept giving appropriate examples and diagrams. Also explain the advantages and disadvantages of using monolithic and layered architecture.
12. Discuss about the virtual machine and exokernel architecture concept with the help of appropriate examples and diagrams. Also explain the advantages and disadvantages of using virtual machine and exokernel architecture.
13. Explain briefly about the client–server architecture and the components of client–server architecture giving appropriate examples and diagrams. Also explain the advantages and disadvantages of using client–server architecture.

1.10 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.
- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.
- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.
- Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.
- Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.
- Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

UNIT 2 PROCESSES

Structure

- 2.0 Introduction
 - 2.1 Objectives
 - 2.2 Process Management
 - 2.3 Process Concept and Threads
 - 2.3.1 OS View of Processes
 - 2.3.2 Process States
 - 2.4 Answers to Check Your Progress Questions
 - 2.5 Summary
 - 2.6 Key Words
 - 2.7 Self Assessment Questions and Exercises
 - 2.8 Further Readings
-

NOTES

2.0 INTRODUCTION

In computing, a process is the instance of a computer programme that is being executed by one or even more threads. It contains the software code and its operation. Depending on the Operating System (OS), a process can consist of multiple execution threads that execute instructions continuously. There is a hairline difference between the program and process in the sense that a program is a passive entity that does not initiate anything by itself whereas a process is an active entity that performs all the actions specified in a particular program.

Depending on the number of processors used in a system, a computer system can be broadly categorized mainly into one of the two types: single-processor system or multiprocessor system. Every operating system has its own internal structure in terms of file arrangement, memory management, storage management, etc. The performance of the entire system depends on its structure.

Previously, there was a boundation of loading only one program into the main memory for execution at a time. This program was very multifaceted and resourceful as it had access to all the hardware resources, such as memory, CPU time, I/O (Input/Output) devices, and so on. With the advancement in the computer system a variety of new and powerful features are automatically added to improve the efficiency and functionality of the system. Modern computer systems support multiprogramming, which allows a number of programs to reside in the main memory at the same time. These programs have the potential to run a number of programs simultaneously thereby requiring the system resources to be shared among them.

Process management requires different functions, such as designing, scheduling, process termination, and a dead lock. Managing all of the system's running processes is the responsibility of the OS, i.e., it manages operations by managing tasks, such as arranging processes and allocating resources.

NOTES

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. Multiple threads can exist within one process, executing concurrently and sharing resources, such as memory, while different processes do not share these resources.

In this unit, you will study about the introduction of process, process states and process management.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- Describe the significance of processes used in the operating system
- Understand the basic concept of the process management
- Analyse the various operations on processes
- Explain the concept of threads
- Know about the process states
- Elaborate on the concept of Process Control Block (PCB) and concurrent processes

2.2 PROCESS MANAGEMENT

Process is the execution of a program executing the actions that are specified in the program. It can be defined as a unit of execution where a program runs. The Operating System (OS) allows us to create, schedule, and terminate the processes that the Control Processing Unit (CPU) uses. A child process is called a process generated by the main process.

Process management requires different functions, such as designing, scheduling, process termination, and a dead lock. Process is a software that is under implementation, an essential component of modern operating systems. Resources that allow processes to share and exchange information must be allocated by the operating system. It also protects from other techniques the resources of each process and facilitates synchronisation between processes.

Managing all of the system's running processes is the responsibility of the operating system. It manages operations by executing activities, such as arranging processes and allocating resources.

Operating System Services

In order to make the task of the programmer easier, an operating system provides the following services:

- **Program execution:** An operating system loads a program from a disk onto the main memory. It allocates the CPU for executing the program. A

program either runs till all the instructions are completely executed or ends abnormally when an error occurs.

Processes

- **I/O operation:** When a running process needs I/O (Input/Output), an operating system provides mechanism to allocate input/output devices to the processes requesting the I/O.
- **Communication:** When one process wants to communicate with another process, it can do so using shared memory or messaging mechanisms provided by the operating system.
- **File system manipulation:** An operating system provides mechanisms to read and write files. It can have one or more file systems. For example, NTFS, FAT32, etc.
- **Error Detection:** Errors can occur during the execution of a process. Errors may be due to power failure, memory error, parity error, connection failure, arithmetic error, array out of bounds, illegal access to a resource, lack of paper in the printer, etc. An operating system must detect the errors and support necessary error-handling mechanisms. Most operating systems have interrupt-handling routines that are called when an interrupt occurs due to errors.
- **Resource allocation:** A computer system consists of several resources like CPU, memory, I/O devices, files, etc., which are to be allocated for user processes. The allocation of the resources to the processes is the job of the operating system. An operating system consists of several routines (programs) to effectively allocate the resources. For example, CPU scheduling routines, memory allocation and deallocation routines, allocation and deallocation of I/O devices, etc.
- **Accounting:** Accounting information includes how many processes are present in the memory, how many users are logged into the system, what resources are being accessed and by whom. Accounting information is useful to improve the services by reconfiguring the system.
- **Protection and security:** Protection ensures controlled access to the system resources. Security allows the users to be authenticated.

NOTES

Operations on Processes

The two important operations provided by an operating system on processes are process creation and process termination. Let us look at each one in detail.

Process Creation

To create a process, we use the *create process* system call. The process in which the *create* system call is called is known as parent process. *Create* system call creates a new process called child process. When a child process is created by a parent process, the parent and child processes can execute concurrently. The child process must be terminated before the parent process. In the UNIX operating

system, the child process is created using the fork system call. The newly created child process contains a copy of the address space of the parent process. The following program creates the child process.

NOTES

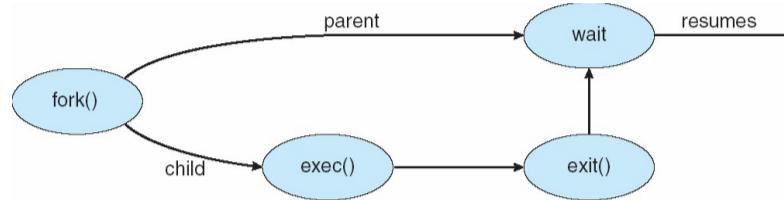


Fig. 2.1 Parent creates the child using the `fork()`.
`fork()` returns 0 for child and > 0 for parent.

```

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

Process Termination

When the last instruction of the process finishes execution, then the operating system must delete it by calling the `exit()` system call. The `abort()` system call is also used to terminate a process. The `exit()` system call is used to terminate itself, whereas the `abort()` system call is used to terminate the other process.

A parent process can be terminated only when all its child processes have been terminated. The parent process calls the `wait()` system call and waits for its children to terminate. When all the child processes have terminated, then the parent process can use the `exit()` system call.

Check Your Progress

1. Define the term process.
2. What is process management?
3. State the various operating System Services.
4. In UNIX, Which system call creates the new process?

NOTES

2.3 PROCESS CONCEPT AND THREADS

A user writes a program, say prime.c, which has a set of instructions to display prime numbers less than a given number. This program is stored on a disk. When the user wants to run the program, the following activities take place:

- It must be brought from the disk on to the primary memory called RAM (Random Access Memory). Here it is termed as process.
- CPU time must be allocated to the process to start execution. If all the instructions are executed, then the process completes and the operating system terminates the process.
- All the resources required for a process like I/O devices such as keyboard, monitor, printer, etc. are allocated to it.

Before a process is brought into the memory, it is called a program. A program is a passive entity, i.e., it needs only memory on the disk for storing instructions. Once the program is brought from the disk into the memory (RAM), it needs CPU registers for computing, bus to transfer data, I/O devices, etc. Hence, a process is an active entity. It is active (requests for resources) till it is completed. During its execution, a process requests, acquires, uses the resources and releases them once it finishes.

It is possible to divide a long process into sub-units and execute the units simultaneously. The sub-units of a process are called threads. There is a subtle difference between a process and a thread. A process is a single flow of execution called main thread. A main thread can spawn other threads each with a different flow of execution.

A thread is a lightweight process. A process by default has a single thread. Most modern operating systems support multithreading and can execute multiple threads simultaneously. For example, a word processor may contain a thread to display graphics, another thread to read the keyboard input and a third thread to check spelling and grammar.

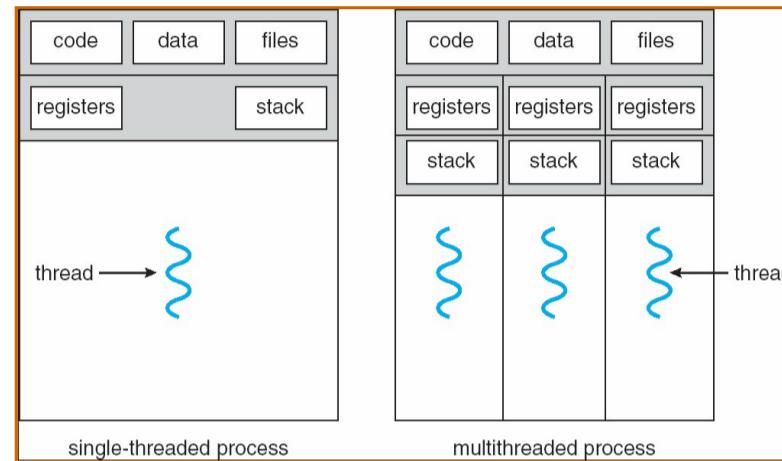
NOTES**Fig. 2.2 A Single Thread and Multiple Threads**

Figure 2.2 shows a single thread, which contains only one flow of execution and multiple threads having multiple flows of execution.

The advantages of threads are as follows:

- **Responsiveness:** If a part of a program (Thread 1) is waiting, then we may allow another part of the program (Thread 2) to continue execution. For example, a multithreaded Web browser can have a thread for user interaction and another thread for loading images. If the loading images thread is taking more time to load a high-resolution picture, a user interaction thread may accept the input from the user to abort the loading images thread.
- **Resource sharing:** All the threads of a process share the same address space. Hence, they all can share the resources effectively. Context switching is easy in threads.
- **Economy:** As the threads are simple as compared to the process, the resource sharing is economical.
- **Utilization of multiprocessors:** In multiple processor architecture, multiple processors can execute multiple threads thereby increasing the benefits. Each processor can execute a thread and all processors put together can speed up the computation.

Threads can be of the following two types:

- **User thread:** These threads run above the kernel threads. The thread library at the user level creates user threads. These are faster to create than the kernel-level thread.
- **Kernel thread:** The kernel threads are at the lower level on which user threads run; creation of kernel thread is complicated.

2.3.1 OS View of Processes

Processes

An operating system is a very huge software that is partitioned into several components. The various components of an operating system include process management, memory management, secondary storage management, I/O management, networking management and security management. Process management deals with the processes. From the operating system view, a program under execution is a process. A process needs resources like CPU, primary and secondary memory, I/O units, etc. All the resources needed for executing a process must be allocated to it during its execution by the operating system. An operating system does the following activities as part of the process management:

- Creating and deleting system processes and user processes
- Suspending and resuming processes
- Allowing process synchronization
- Allowing one process to communicate with the other
- Providing some mechanisms to handle deadlocks

A program can have two copies executing simultaneously. For example, many copies of a word processor can be running, all related to a single word processor. Multiple copies of Web browsers may be running simultaneously, each loading a different Web page.

A system process is an operating system program. Memory manager is an operating system program that allocates and deallocates the memory. A user process is an application process; for example, media player is a user process. One or more system processes execute a user process.

2.3.2 Process States

A process changes its state during its lifetime. A state defines the current activity of that process.

A process can be in any one of the following states:

- **New:** A process is in a new state if it is just created or loaded into the main memory.
- **Ready:** All the processes waiting for CPU time are in a ready state.
- **Running:** When a CPU is allocated to a ready state process, then it is in a running state.
- **Waiting:** If a process is waiting for I/O, then it is in a waiting state.
- **Terminated:** If a process has finished execution, then it is in a terminated state.

When a program is loaded from a disk into the main memory, it is considered to be in a new state. In a multiprogramming environment, several programs may be loaded into the main memory for execution. Hence, several processes can be

NOTES

NOTES

in the new state. All the processes waiting for the CPU alone are in the Ready state. All the ready state processes are put in a ready queue. A ready queue consists of processes that need CPU cycles. An operating system from a ready queue selects a process for execution. This selection can be done by using CPU scheduling algorithms. The selected process is given the CPU time and considered to be in the running state. The selected process changes its state from ready to running. A running state process has the CPU resource with it. It can release the CPU resource voluntarily when it needs I/O or has finished execution. The running process can be forced to release the CPU if a high-priority process is brought into the ready queue. In both the cases, the running state process can be converted to the ready state or the waiting state. A process waiting for I/O is put in a wait queue. If a process has finished waiting, it can be put in a ready queue. A running process is terminated if all the instructions are executed. This is called run till completion.

A process first is in a new state, then it enters into a ready state; a ready state process can be selected for running. A running process can wait or terminate.

Figure 2.3 depicts the states of a process.

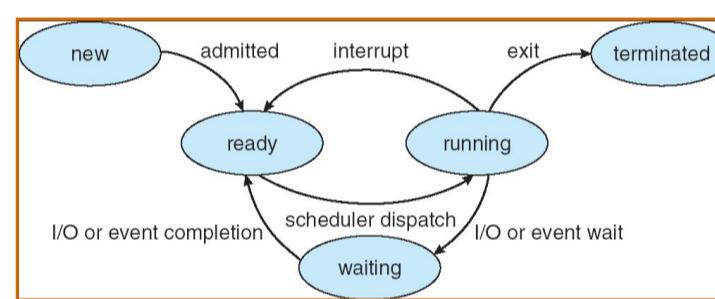
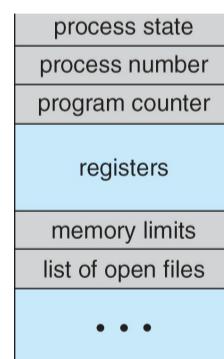


Fig. 2.3 States of a Process

Process Control Block

All the data related to a process such as process number, the state of a process, program counter, value list of registers, their contents, memory limits, list of open files, etc. are stored in a block called Process Control Block (PCB).

The following Figure below depicts a process control block.



In Figure 2.3, a process state can be ready, running, waiting or terminated. A program counter contains the address of the next instruction. Registers include the contents of the accumulator, index registers, stack pointer, general-purpose registers, etc. Memory limits include the value of the base and limit registers. The list of files opened by the process is also stored in the PCB.

NOTES

Concurrent Processes

In a multiprogramming environment where there are more than one processors, we can execute more than a single process. Each processor can execute a process and multiple processors can execute multiple processes. All the processes executing in the system simultaneously are called concurrent processes.

Concurrent processes are of two types—Independent and cooperative. If a concurrent process cannot affect or be affected by the other process executing in the system, it is called independent process.

If a concurrent process can affect or be affected by other processes in the system, it is called cooperating process.

A cooperating process must share data and information with other processes. To share information, cooperating processes must communicate with each other using inter-process communication and process synchronization techniques.

Check Your Progress

5. Give the definition of thread.
6. Explain about the user thread and kernel thread.
7. What is the ready state of a process?
8. What is a Process Control Block?
9. Explain Concurrent processes.
10. Classify the types of Concurrent processes.

2.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. A process is the instance of a computer programme that is being executed by one or even more threads. It contains the software code and its operation. Depending on the operating system (OS), a process can consist of multiple execution threads that execute instructions continuously.
2. Process management means scheduling threads of the CPU, creating and deleting the both user and system processes.

NOTES

3. Program execution, I/O operations, file system manipulation, error detection, resource allocation, accounting, protection and security.
4. In the UNIX operating system, the child process is created using the fork system call.
5. A thread is a Simple CPU use unit composed of a program clock, a stack, and a set of registers (and a thread ID.).
6. Threads can be of the following two types:
 - **User thread:** These threads run above the kernel threads. The thread library at the user level creates user threads. These are faster to create than the kernel-level thread.
 - **Kernel thread:** The kernel threads are at the lower level on which user threads run; creation of kernel thread is complicated.
7. Ready state of the process means process has all necessary resources which are required for execution of that process when CPU is allocated. Process is ready for execution but waiting for the CPU to be allocated.
8. All the data related to a process such as process number, the state of a process, program counter, value list of registers, their contents, memory limits, list of open files, etc. are stored in a block called Process Control Block (PCB).
9. All the processes executing in the system simultaneously are called concurrent process.
10. Concurrent processes are of two types—Independent and cooperative. If a concurrent process cannot affect or be affected by the other process executing in the system, it is called independent process.
If a concurrent process can affect or be affected by other processes in the system, it is called cooperating process.

2.5 SUMMARY

- An operating system loads a program from a disk onto the main memory. It allocates the CPU for executing the program.
- Process management requires different functions, such as designing, scheduling, process termination, and a dead lock.
- Process is a software that is under implementation, an essential component of modern operating systems. Resources that allow processes to share and exchange information must be allocated by the operating system.
- Managing all of the system's running processes is the responsibility of the operating system.

- An operating system loads a program from a disk onto the main memory. It allocates the CPU for executing the program. A program either runs till all the instructions are completely executed or ends abnormally when an error occurs.
- When a running process needs I/O, an operating system provides mechanism to allocate input/output devices to the processes requesting the I/O.
- When one process wants to communicate with another process, it can do so using shared memory or messaging mechanisms provided by the operating system.
- An operating system provides mechanisms to read and write files. It can have one or more file systems. e.g. NTFS, FAT32, etc.
- Errors can occur during the execution of a process. Errors may be due to power failure, memory error, parity error, connection failure, arithmetic error, array out of bounds, illegal access to a resource, lack of paper in the printer, etc. An operating system must detect the errors and support necessary error-handling mechanisms.
- A computer system consists of several resources like CPU, memory, I/O devices, files, etc., which are to be allocated for user processes. The allocation of the resources to the processes is the job of the operating system.
- Protection ensures controlled access to the system resources. Security allows the users to be authenticated.
- To create a process, we use the *create process* system call. The process in which the *create* system call is called is known as parent process. *Create* system call creates a new process called child process.
- In the UNIX operating system, the child process is created using the fork system call.
- The newly created child process contains a copy of the address space of the parent process.
- When the last instruction of the process finishes execution, then the operating system must delete it by calling the exit () system call. The abort system call is also used to terminate a process. The exit () system call is used to terminate itself, whereas the abort system call is used to terminate the other process.\
- Before a process is brought into the memory, it is called a program. A program is a passive entity, i.e. it needs only memory on the disk for storing instructions
- If a part of a program (Thread 1) is waiting, then we may allow another part of the program (Thread 2) to continue execution.
- All the threads of a process share the same address space. Hence, they all can share the resources effectively. Context switching is easy in threads.

Processes

NOTES

NOTES

- As the threads are simple as compared to the process, the resource sharing is economical.
- In multiple processor architecture, multiple processors can execute multiple threads thereby increasing the benefits. Each processor can execute a thread and all processors put together can speed up the computation.
- User threads run above the kernel threads. The thread library at the user level creates user threads. These are faster to create than the kernel-level thread.
- The kernel threads are at the lower level on which user threads run; creation of kernel thread is complicated.
- An operating system is a very huge software that is partitioned into several components. The various components of an operating system include process management, memory management, secondary storage management, I/O management, networking management and security management.
- A process is in a new state if it is just created or loaded into the main memory.
- All the processes waiting for CPU time are in a ready state.
- When a CPU is allocated to a ready state process, then it is in a running state.
- If a process is waiting for I/O, then it is in a waiting state.
- If a process has finished execution, then it is in a terminated state.
- In a multiprogramming environment where there are more than one processors, we can execute more than a single process. Each processor can execute a process and multiple processors can execute multiple processes. All the processes executing in the system simultaneously are called concurrent processes.
- All the data related to a process such as process number, the state of a process, program counter, value list of registers, their contents, memory limits, list of open files, etc. are stored in a block called Process Control Block (PCB).
- All the processes executing in the system simultaneously are called concurrent processes.
- If a concurrent process cannot affect or be affected by the other process executing in the system, it is called independent process.
- If a concurrent process can affect or be affected by other processes in the system, it is called cooperating process.

2.6 KEY WORDS

- **Process:** A program under execution is called a process. A process is an active entity. Each process has a program counter that specifies the address of the next instruction to be executed.
- **Parent process:** The process in which the create system call is called is known as parent process.
- **Child process:** A child process is a process created by a parent process in operating system using a fork () system call.
- **Thread:** The sub-units of a process are called threads. It is possible to divide a long process into sub-units and execute the units simultaneously.
- **User thread:** These threads run above the kernel threads. The thread library at the user level creates user threads. These are faster to create than the kernel-level thread.
- **Kernel thread:** The kernel threads are at the lower level on which user threads run; creation of kernel thread is complicated.
- **Process states:** A process changes its state during its lifetime. A state defines the current activity of that process. A process can be new, ready, running, waiting and terminated.
- **CPU utilization:** The amount of CPU time utilized must be maximum for a scheduling algorithm. This may be in the range from 0 to 100 per cent; for a lightly loaded system, it is about 40 per cent and for heavily loaded system, CPU utilization is 90 per cent.
- **Process control block:** All the data related to a process such as process number, the state of a process, program counter, value list of registers, their contents, memory limits, list of open files, etc. are stored in a block called Process Control Block (PCB).
- **Concurrent Processes:** All the processes executing in the system simultaneously are called concurrent processes.

NOTES

2.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What do you meant by program execution?
2. Define parent and child process in process management.
3. Which function is used for termination of process?

NOTES

4. What is program and process?
5. Explain the concept of thread.
6. Classify the types of thread.
7. Which State of a process defined “The process is being created”?
8. What is the lifecycle of process?

Long-Answer Questions

1. Briefly discuss the services of operating system.
2. Explain about the process creation and process termination in detail.
3. Discuss about the activates which are used for running a program?
4. What is multiprocessors?
5. Illustrate the advantages of thread.
6. Describe in detail about the process management activities.
7. Explain briefly about the process state in detail.
8. Describe the Process Control Block (PCB) and concurrent process.
9. Discuss about the concurrent process and its types in detail?

2.8 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.
- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.
- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.
- Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.
- Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.
- Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

UNIT 3 INTERRUPTS AND INTERPROCESS COMMUNICATION

*Interrupts and
Interprocess
Communication*

NOTES

Structure

- 3.0 Introduction
 - 3.1 Objectives
 - 3.2 Introduction to Interrupts in Operating System
 - 3.3 Interprocess Communication
 - 3.4 Answers to Check Your Progress Questions
 - 3.5 Summary
 - 3.6 Key Words
 - 3.7 Self Assessment Questions and Exercises
 - 3.8 Further Readings
-

3.0 INTRODUCTION

Interrupts are events that are used to stop the current execution and start a new urgent execution. The events may be created externally by some hardware devices, internally by software interrupt instructions, due to attempts to execute illegal instructions and access illegal memory or some hardware failures.

Interrupts are signals that external devices usually I/O devices, transmit to the Central Processing Unit (CPU). They tell the CPU to stop its current operations and execute the Operating System (OS) portion that is necessary. An interrupt is a response by the processor in digital computers to an event that requires software attention. An interrupt state warns the processor and when allowed, acts as a request for the processor to interrupt the currently executing code so that the event can be treated in a timely manner.

Most of the devices are connected or interfaced to the computer system as one or more memory locations. All I/O devices have controller with some registers or memory locations, and interfacing address, data and control lines called buses. Other operations of the device are internal to it and the operating system has no role in the internal working other than initiating the operations and reading the status of the device through the controller.

Direct Memory Access (DMA) is a computer system feature that enables access to main system memory (random access memory) by some hardware subsystems independent of the central processing unit (CPU). Without DMA, it is usually fully occupied for the entire duration of the read or write process when the CPU uses programmed input/output, and is thus unable to do other work. With DMA, the CPU initiates the transfer first, then performs other operations while the

NOTES

transfer is in progress, and when the process is completed, it eventually receives an interrupt from the DMA controller (DMAC). This feature is useful at any time when the CPU is unable to keep up with the data transfer rate, or when the CPU needs to operate while waiting for a relatively slow transfer of I/O data. DMA is used by many hardware systems.

Inter-Process Communication or InterProcess Communication (IPC) simply refers to the mechanisms provided by the operating system to allow shared data to be handled by processes. Applications may usually use IPC, categorised as clients and servers, where data is requested by the client and the server responds to client requests. The design process for microkernels and nanokernels is very important for IPC, which reduces the number of functionalities given by the kernel. Those features are then acquired by interacting via IPC with servers, leading to a significant increase in communication compared to a regular monolithic kernel.

In this unit, you will study about the interrupts in Operating System (OS) and inter-process communication in operating system.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basic concepts of an interrupts in operating system
- Discuss about the input-output data transfer techniques
- Elaborate on the concept of Direct Memory Access(DMA)
- Analysis the cause of interrupts
- Provide an overview of Inter-Process Communication(IPC)
- Explain the need for IPC synchronization

3.2 INTRODUCTION TO INTERRUPTS IN OPERATING SYSTEM

Interrupts are events that are used to stop the current execution and start a new urgent execution. The events may be created externally by some hardware devices, internally by software interrupt instructions, due to attempts to execute illegal instructions and access illegal memory or some hardware failures. An interrupt first, saves the contents of program counter in the stack. Then the processor enters a subroutine called Interrupt Service Subroutine (ISS). CPU interrupt handling enables or disables interrupts in the CPU. Each device has an Interrupt ReQuests (IRQ) number which is based on Program Controlled Interrupt (PIC). It supports PIC the 8259A chip and eight Interrupt Request (IRQ) lines. There are two available schemes as master and slave which give the priority from highest to lowest as IRQ0-1, IRQ8-15, IRQ3-7. PIC is accessed by port addresses such as 0x20

NOTES

and 0x21 for master. IRQ devices generate a signal on IRQn, where n is the number of request. Only those parts of the program and data are executed which are active in physical RAM. The other parts resides in a swap file named as Win386.swp is called in Windows 95/98 or page file. The pagefile.sys comes in Windows NT versions. When a program accesses some addresses which are not in physical RAM, an interrupt file called Page Fault (PF) is generated. When a device generates an interrupt, the CPU hardware stops running an ordinary program and jumps to an interrupt handling routine in the Device Driver. When a peripheral is ready it sends an interrupt signal the CPU then the CPU stops execution of the program. It accepts data from peripheral and returns to the interrupted program. Almost all modern processors have a mechanism to catch the interrupt events and switch the execution from the currently executing program to some other program or function called the interrupt handler. The interrupt handler does the required urgent processing. In operating systems, interrupts are mainly used for the following:

- (i) I/O device interrupts are used for data transfers between computer memory and I/O devices.
- (ii) Hardware timer interrupts are used to record time and distribute the processor time among different applications currently loaded for execution.
- (iii) Handle page or segment missing events (page or segment faults) in memory management for loading pages or segments from secondary storage.
- (iv) Exception handling: Attempts to access Illegal memory, attempts to execute illegal instructions, attempts to divide by zero causing arithmetic overflow, etc., raises exception error interrupt and transfers execution to exception handler to abort the current process.

Following an interrupt, the execution branches to the OS code, and the OS looks for the reason of the interrupt. From the interrupt number and the contents of some register indicating the cause of interrupt, the OS identifies the device or instruction that caused the interrupt and chooses to execute the appropriate handler required to do the requested processing. The OS must also remember what instruction was under execution and the memory location of the next instruction (and the current context of execution) to report error conditions, if any, and to resume the interrupted program after the execution of the interrupt handler.

Interrupts must be enabled so that they can respond to their events. There may be many levels or numbers of interrupts, for example 5, in a CPU and there may be bits in status register of the CPU corresponding to each of the interrupts. So, each of these interrupts can be enabled or disabled selectively.

Input–Output Data Transfer Techniques

Most of the devices are connected or interfaced to the computer system as one or more memory locations. All I/O devices have controller with some registers or memory locations, and interfacing address, data and control lines called buses. Other operations of the device are internal to it and the operating system has no

NOTES

role in the internal working other than initiating the operations and reading the status of the device through the controller. The interfacing lines are connected to the system buses through appropriate I/O ports present in both the CPU box and device controller part of the devices.

There are two ways of interfacing I/O devices; memory mapped I/O and I/O mapped I/O. The devices use some addresses in the memory address space in case of memory mapped I/O or separate I/O address space in case of I/O mapped I/O. Thus, the input-output operations may be done by executing **Move** instructions (if memory address space is used) or **In/Out** instructions (if separate I/O address space is used) for moving data between CPU and memory locations. However, before actually transferring the data, the CPU must first initiate the transfer when the device is ready (not busy with I/O for some other process) and check whether the device is ready to accept data (in case of writing output to devices) or the data is readily available in the registers/memory (in case of reading from devices) of device controller.

Three major techniques are in use for transferring data between memory and I/O devices. They are as follows:

- (i) Programmed data transfer or I/O
- (ii) Interrupt-driven I/O
- (iii) Direct Memory Access (DMA) I/O

(i) Programmed Data Transfer or I/O

In programmed data transfer technique, the CPU first initiates the transfer and checks continuously in a busy loop waiting for *data ready* signal (for reading) or *ready to accept* signal (for writing) from the device. This is called polling. The CPU will not be doing anything else while waiting for I/O data ready/ accept signal. When the data is available (or device is ready to accept data) as indicated by setting some status bits in the device controller, the CPU transfers the data between device controller and the system memory. This is not an efficient way of using CPU, and is a slower method of data transfer. It is likely that some of the data samples get lost in case of realtime data transfer.

(ii) Interrupt-Driven I/O

In interrupt-driven data transfer, the CPU initiates the data transfer and enables the device interrupt to interrupt CPU execution when the data is ready or the device is ready to accept data. The CPU then executes other jobs. When the device is ready for the transfer, it will inform the CPU (operating system) through the interrupt. The interrupt handler will then do the actual data transfer between device and memory. Here, the CPU does not wait in a busy loop; instead, it executes other jobs. The problem here is the overhead of interrupt handling which involves stopping the current process execution, saving the context, executing the interrupt handler, selecting a new process for execution, restoring the context of new process and transferring control to the new process.

NOTES

(iii) Direct Memory Access (DMA) I/O

In this method, the CPU just initiates the data transfer through a DMA controller, and does other jobs as in the case of interrupt driven I/O. The DMA controller issues request to the I/O device and does the transfer when ready. A DMA controller is like a processor with functions to check the data ready or accept signal and to transfer data between device controller and system memory. The DMA can directly access memory bus for data transfer asking the CPU to wait for some clock cycles (no process switching overheads). After finishing the transfer of entire block of data requested, the DMA interrupts the CPU to inform the completion of data transfer. Large block of data can be transferred without CPU intervention. The execution is slowed down as the data transfer (of each word) is done using clock cycles in between the execution of two instructions.

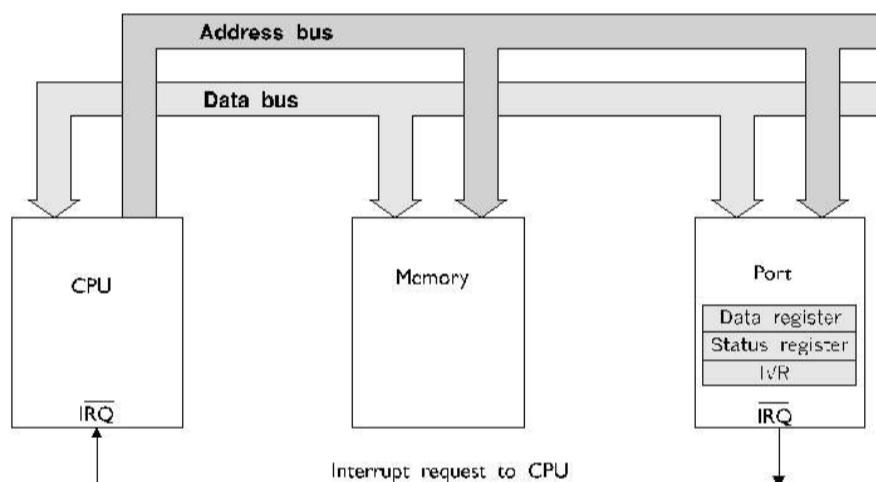


Fig. 3.1 Interrupt Organization

Figure 3.1 shows an interrupt request line that runs from the CPU to all the peripherals. Each peripheral capable of generating an interrupt is connected to the Interrupt ReQuest (IRQ) line. When a peripheral requires attention, the peripheral asserts its IRQ output. Interrupt means the device gets the attention of CPU.

Interrupts and Intel Architecture

Interrupts are not entirely associated with I/O devices. The 8086 family microprocessors provide 256 interrupts as software interrupts. It produces an interrupt vector table which is situated in 0000:0000 and extended up to 1024 bytes. It holds the address of Interrupt Service Routines (ISR). The all four bytes give to users a pool of the 256 interrupt vectors. Table 3.1 shows the interrupt numbers and their uses with reference to Intel architecture.

NOTES

Table 3.1 x86 Interrupt Vectors

INT (Hex)	IRQ	Common Uses
00 - 01	Exception Handlers	-
02	Non-Maskable IRQ	Non-Maskable IRQ (Parity Errors)
03 - 07	Exception Handlers	-
08	Hardware IRQ0	System Timer
09	Hardware IRQ1	Keyboard
0A	Hardware IRQ2	Redirected
0B	Hardware IRQ3	Serial Comms. COM2/COM4
0C	Hardware IRQ4	Serial Comms. COM1/COM3
0D	Hardware IRQ5	Reserved/Sound Card
0E	Hardware IRQ6	Floppy Disk Controller
0F	Hardware IRQ7	Parallel Comms.
10 - 6F	Software Interrupts	-
70	Hardware IRQ8	Real Time Clock
71	Hardware IRQ9	Redirected IRQ2
72	Hardware IRQ10	Reserved
73	Hardware IRQ11	Reserved
74	Hardware IRQ12	PS/2 Mouse
75	Hardware IRQ13	Math's Co-Processor
76	Hardware IRQ14	Hard Disk Drive
77	Hardware IRQ15	Reserved
78 - FF	Software Interrupts	-

Hardware Interrupts

The Programmable Interrupt Controller (PIC) handles hardware interrupts. Most PC's have two of them located at different addresses. One handles IRQs 0 to 7 and the other, IRQs 8 to 15, giving a total of 15 individual IRQ lines, as the second (Program Controlled Interrupt) PIC is cascaded into the first, using IRQ2. Table 3.2 lists the PIC1 bit format using IRQ1 and Table 3.3 lists the PIC2 bit format using IRQ2.

NOTES

Table 3.2 PIC1 Bit Format Operation using IRQ1

Bit	Disable IRQ	Function
7	IRQ7	Parallel Port
6	IRQ6	Floppy Disk Controller
5	IRQ5	Reserved/Sound Card
4	IRQ4	Serial Port
3	IRQ3	Serial Port
2	IRQ2	PIC2
1	IRQ1	Keyboard
0	IRQ0	System Timer

Table 3.3 PIC2 Bit Format Operation (0xA1) using IRQ2

Bit	Disable IRQ	Function
7	IRQ15	Reserved
6	IRQ14	Hard Disk Drive
5	IRQ13	Maths Co-Processor
4	IRQ12	PS/2 Mouse
3	IRQ11	Reserved
2	IRQ10	Reserved
1	IRQ9	Redirected IRQ2
0	IRQ8	Real Time Clock

The following are the features of Interrupt Service Routine (ISR):

- (i) It saves all used registers.
- (ii) It issues EOI (End-Of-Interrupt) command to PIC.
- (iii) It restores registers.
- (iv) It is finished with iret instruction.

There is a type of interrupt known as timers interrupt. It has the following characteristics:

- (i) It allows STI (Set Interrupts) and also CLI (Clear Intercepts) in CPU (cli).
- (ii) It sets up interrupt gate descriptor in IDT (Interrupt Descriptor Table) using irq0inthand.
- (iii) It sets up timer downcount to determine tick interval.
- (iv) It disallows timer interrupts by masking IRQ0 in the P.I.C. by making bit 0 be 1 in the mask register (port 0x21).

NOTES

Interrupt Action

Let us understand interrupt action with an example of a card reader (1442). This I/O device starts and the card moves to the read station. This is shown in Figure 3.2.

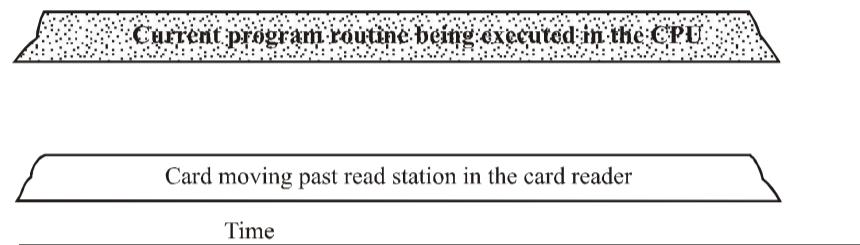


Fig. 3.2 Card reader (1442) moves to the Read Station

As soon as it moves, the card reader signals the CPU. This signal to the CPU is an interrupt request. The interrupt does not occur until execution of the current instruction is completed. At that time, a forced branch occurs to an interrupt-handling subroutine. This is shown in Figure 3.3.

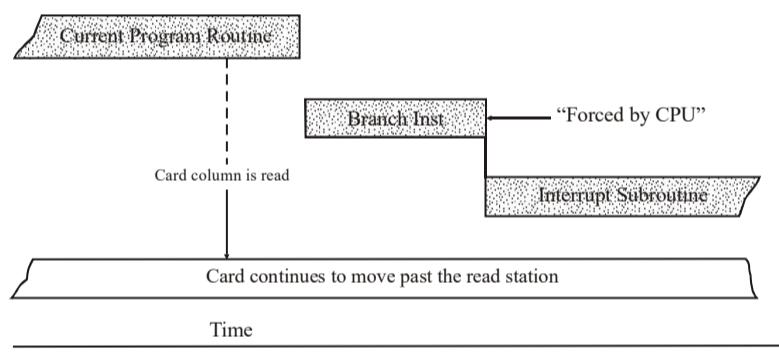


Fig. 3.3 Interrupt Action is Shown

Logical requirements of the interrupt subroutine are carried out before a return to the interrupted program occurs.

The two ways of branching into an input/output routine are as follows:

- (i) Program-controlled branching
- (ii) CPU-forced branching

Program-controlled branching

In a program-controlled branch to an I/O routine, a properly written program has address information for the branch-to location and the return address, if it is required.

CPU-forced branching

In a CPU-forced branch, it is possible to branch to the proper subroutine and return via another branch to the interrupted program at the end of the subroutine. For this purpose, the following information is available:

NOTES

- (i) The core-storage address of the beginning of the interrupt subroutine.
- (ii) The core-storage address of the next instruction to be executed in the current program, that is, the address of the place to return to after the interrupt routine is completed.

At the time of the actual interrupt after the current instruction is completely executed) the address of the next instruction is in the instruction-address register.

Table 3.4 shows the interrupt levels and their associated locations.

Table 3.4 Interrupt Levels and their Associated Locations

Interrupt Level	Interrupt Vector Location (in core storage)	Device
0	00008 (in decimal)	1442 Card Read Punch (column read, punch)
1	00009	1132 Printer, synchronous communications adapter
2	00010	Disk storage, storage access channel
3	00011	1627 Plotter, 2250 Display Unit, storage access channel, System/7
4	00012	1442 (operation complete), keyboard, console printer, 1134 Paper Tape Reader, 1055 Paper Tape Punch, 2501 Card Reader, 1403 Printer, 1231 Optical Mark Page Reader, storage access channel
5	00013	Console (program stop switch and interrupt run), storage access channel

The interrupt subroutine save the contents of certain index and machine registers by storing their contents into core storage. Such registers are used for data manipulation within the subroutine. Only the contents of those registers that are used by the subroutine need to be saved. The data integrity of such registers is maintained because the interrupted program uses the same registers. At the end of the interrupt subroutine, the contents of the affected registers are loaded back into the registers from core storage.

For example, the accumulator is used by all subroutines. Therefore, the contents of the accumulator are stored into core storage before the accumulator is used by the subroutine. At the end of the subroutine, before a return to the interrupted program occurs, the accumulator is loaded with the data that is saved.

Causes of Interrupts

Basically, two methods are used to determine what condition is causing an interrupt. These methods are as follows:

NOTES

- (i) Examination of the device status word
- (ii) Examination of the interrupt-level status word

The I/O device causing an interrupt is determined by examination via programming of the interrupt-level status word, when necessary. Then, after the device is known, the device-status word is examined to determine the condition in the device that caused the interrupt to occur.

First consider a level-0 interrupt. An interrupt-level status word is not used at level 0. Only the device status word need be examined at interrupt level 0. Either of the following two conditions can cause level-0 interrupt to occur:

- (i) The card reader (1442) has a data word ready for storing into core storage during a read operation.
- (ii) The card punch (1442) is ready to accept data for punching a column during a punch operation.

Because the 1442 can perform only a read or a punch operation at one point of time, an interrupt at level 0 is for either a read or a punch operation. The program must determine which operation has occurred; sending a data word from core storage to the card punch during a read operation would be inappropriate.

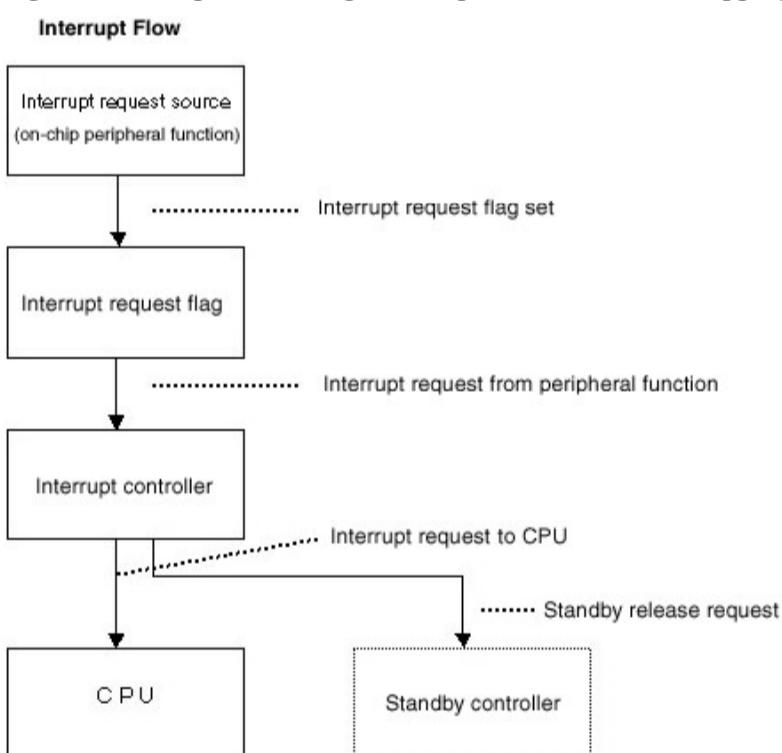


Fig. 3.4 Interrupt Flow

Figure 3.4 shows that the basic operations of interrupt service are broadly carried out by four phases. The first is the part that requests the interrupt, the second is the interrupt request flag, the third is the interrupt controller that controls the interrupt and the fourth is the CPU that actually executes the interrupt services.

NOTES

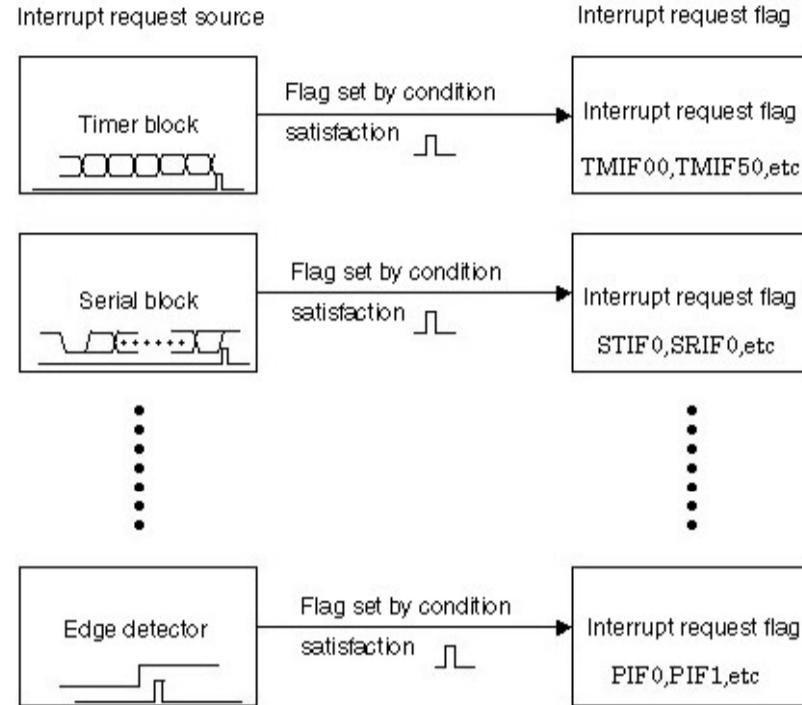


Fig. 3.5 Interrupt Acknowledgement

Figure 3.5 shows how interrupt request source parts request interrupts include the timer block, serial block, and external signal edge detector. Once an interrupt request condition, meets with the corresponding interrupt, it requests flag to set in these parts. The first phase of interrupt service involves the satisfaction of a specified condition in each peripheral function block. Some conditions such as matching with a compare register in the timer block, completion of communication in the serial block, detection of an edge via an external interrupt are some interrupt services for which the corresponding interrupt request flag is set.

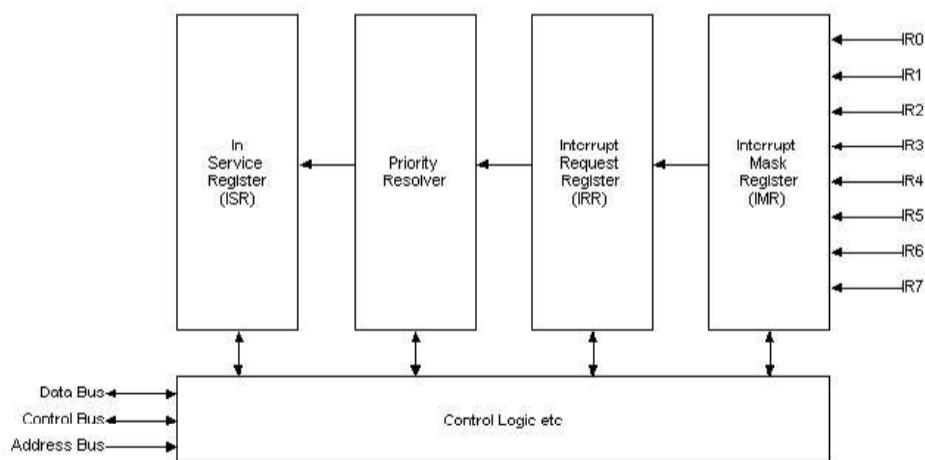


Fig. 3.5 Block Diagram of the PIC

NOTES

Figure 3.6 shows the basic block diagram of the PIC. The 8 individual interrupt request lines are first passed through the Interrupt Mask Register (IMR) to check whether they are masked or not. If they are masked, the request is not processed further. And if they are not masked, they register their request with the Interrupt Request Register (IRR).

Interrupts include system call that executes syscall program executed and a timer. The timer register starts from 1 to 0 and a non-zero timer counts down every microsecond. They both collectively complete a disk operation. Interrupts also include various types of program errors as follows:

- (i) ip=0 : undefined instruction
- (ii) ip=1 : illegal instruction in user mode
- (iii) ip=2 : logical address >= bound register

The required steps to handle an interrupt are as follows:

Step 1: psw saved in ipsw, psw set to 0

Step 2: interrupt parameter (if any) placed in ip register

Step 3: ia saved in ia

Step 4: new is taken from interrupt vector area (offset depends on which interrupt it is)

Step 5: timer and disk interrupt can be masked (recorded but delayed) by setting psw bit 0.

The timer interrupts keep the information of disk controller. The following code is written in ‘C’ as follows:

```
const int Block_Size = 4096;
enum disk_command {Load_Block=0, Store_Block=1};

//Declared Enum data type and assigns 0 to Load_Block and
1 to Store_Block

struct disk_control_register
{
    unsigned int command : 1;
    unsigned int interrupt_enabled : 1;
    unsigned int disk_block : 20;
    unsigned int padding : 10;
}; // Structure of Disk Control Register is defined

volatile disk_control_register *Disk_control =
(disk_control_register *)0xF0000000;
```

NOTES

```
void **Disk_memory_address = (void **)0xF0000004;
// Disk_memory_address is defiend by pointer

enum disk_status { Disk_Idle=0, Disk_Busy=1 };
//Declared Enum data type and assigns 0 to Disk_Idle and
1 to Disk_Busy

struct disk_status_register
{
    unsigned int busy : 1;
    unsigned int padding : 31;
}; Structure of Disk Status Register is defined

disk_status_register *Disk_status = (disk_status_register
*)0xF0000008;
// 0xF0000008 is prefetchable by 32-bit memory
```

The Microprocessor without Interlocked Pipeline Stages (MIPS) OS contains code in distribution and runs on UNIX systems. A number of changes are possible in the low-level code that interfaces to the hardware. The Java OS code in distribution runs on Java 1.1 systems and Java 1.1 browsers. A number of changes are possible in the low-level code that interfaces to the hardware. The MIPS OS and Java OS almost use same hardware interface.

Check Your Progress

1. What do you meant by interrupts?
2. In Which system Interrupts form an important part?
3. State the three input-output data transfer techniques.
4. Give the definition of direct memory access (DMA).
5. In a computer, which two methods are used for branching into input/output routine?
6. Which type of program errors are included in interrupts?
7. Define program-controlled branching.

3.3 INTERPROCESS COMMUNICATION

The processes that coexist in the memory at the same time are called concurrent processes. The concurrent processes may either be independent or cooperating. The independent (also called competitors) processes, as the name implies, do not

NOTES

share any kind of information or data with each other. They just compete with each other for the resources like Central Processing Unit (CPU), I/O(Input/Output) devices, and so on that are required to accomplish their operations. The cooperating (also called interacting) processes, on the other hand, need to exchange data or information with each other. For this, they require some mechanism that allows them to communicate with each other. One such mechanism is Interprocess Communication (IPC)—a very useful facility provided by the operating system.

Two basic communication models for providing IPC are shared memory systems and message passing systems. In the former model, a fragment of memory is shared among the cooperating processes. Hence, if processes want to exchange data or information, it can do so by writing to and reading from this shared memory. However, in the latter model, the cooperating processes communicate by sending and receiving messages from each other. The communication using message passing is very time consuming as compared to shared memory. This is because the message passing system is implemented with the help of an operating system calls and thus, it requires a major involvement of kernel. On the other hand, in shared memory systems, system calls are used only to set up the shared memory area. Once the shared area is set up, no further kernel intervention is required.

Need for IPC Synchronization

When two or more processes try to access and manipulate the same data simultaneously, then we call this concurrent access to shared data. Concurrent access and manipulation to shared data introduce the concurrency problems. To solve the concurrency problems, such as race condition, we use process synchronization.

For example, let Deposit (x, 5000) be a process that deposits 5000 amount into the account x. Let Withdraw (X, 2000) be the process that withdraws 2000 from account X and Interest (X, 200) be the third process that adds the interest to the account X. All the above three processes operate on the same data, i.e. account balance. Let the balance of the account X be 3000.

Suppose all the three processes are simultaneously executing, then all of them read the balance as 3000. The following activity takes place:

```
Deposit (X, 5000) {  
  
    T0 : Read bal; //reads balance as 3000  
    T1   Balance += 5000;  
    T2   Write balance;  
}  
  
_____| |_____
```

```
Withdraw (X, 2000) {  
  
T0:    Read bal; //reads balance as 3000  
T1     Balance -= 2000;  
T2     Write balance;  
}
```

```
Interest (X, 200) {  
  
T0:    Read bal; //reads balance as 3000  
T1     Balance += 200;  
T2     Write balance;  
}
```

All the above processes have three instructions executing at time T₀, T₁, T₂.
At T₀ all the processes read the balance, which is 3000.

At T₁ all the processes manipulate the balance.

At T₂ all the processes write the balance value.

Since the balance is a data value, it can store only single value. Whichever process finishes last updates the process balance value. If all of the processes read balance and update it, then we will get inconsistent balance. The shared data like balance must be accessed by one process after the other process. The order must be followed; otherwise, if several processes are manipulating the shared data simultaneously, the outcome depends on the order in which the processes finish execution. This is called race condition. Several processes are racing for the shared data. Race condition occurs very frequently in an operating system. To maintain the consistency, we need to synchronize the shared data access. To solve the problem of race condition, we use process synchronization. It will impose certain rules while accessing the share data.

Critical Section Problems and Solutions

Consider a system consisting of n processes {p₀, p₁, p₂...p_n} where each process has a section of code called critical section. A critical section code is the code that manipulates the shared data. For example, in the previous section, manipulating the balance is the critical section code. The critical section problem states that when one process is executing in its critical section, then no other process is allowed to enter into its critical section. This implies that the execution of critical section is mutually exclusive. Before entering into the critical section, a process must request permission. The code that implements the request is called entry section. A process first executes the entry section code and then enters into the critical section. After

NOTES

NOTES

completing the critical section, the process executes the exit section code. The general structure of the process P_i is given in the following code:

```
do {  
    entry section;  
    critical section;  
    exit section  
    remainder section;  
} while (true);
```

From the above code, it is clear that the critical section manipulating the shared data is always preceded by the entry section and followed by the exit section. Remainder section is the code that is not involved in shared data manipulation.

The solution to the critical section problem must satisfy the following requirements:

- Mutual Exclusion: If P is executing in its critical section, then no other process can execute in their critical section.
- Progress: When a process wants to enter into its critical section, then the processes that are not executing in their remainder sections make the decision.
- Bounded waiting: If a process makes a request to enter into its critical section, then it must get the chance to enter into the critical section in finite time.

Semaphores and Implementation

A semaphore is a synchronization tool. It is an integer variable. It can be initialized and two atomic operations are possible with semaphore; they are wait and signal. The syntax for the two operations is as follows:

```
Wait(s) {  
    While(s <= 0)  
        ; //Do no operation or wait;  
    s--;  
}  
  
signal(s) {  
    s++;  
}
```

where s is the semaphore. It is an integer variable.

The wait operation checks to see whether the value of semaphore s on which wait is called is less than or equal to zero; if it is $<= 0$, then no operation is done or it waits. If S is not less than or equal to 0, then the condition in it is false

S—is exempted. The signal operation is straight forward. If increments the semaphore value by one. Let us see how semaphore can be used to implement mutual exclusion (mutex).

Let mutex be the semaphore, which is initialized to 1.

```
do {mutex  
    wait(mutex)  
    CS  
    signal (mutex)  
    RS  
} which(1);  
if mutex = 1, then wait(mutex) checks; if mutex value is <= 0, then wait does no operation
```

if mutex value >0 as in this case, CS is executed.

If mutex <= 0 and if other process wants to enter into CS where p1 is executing, then it performs wait (mutex) as mutex = 0; it waits until the previous process finishes CS and executes signal (mutex), which increments mutex value to 1. No other process can enter into its critical section.

In short, if a process wants to enter into CS, it executes wait (mutex) and if other process is also ready into CS, then it waits; otherwise, enter into CS. When CS is finished, it executes signal RS as mutex = 0

IPC using Messages

Inter-process communication allows a process to communicate with other processes. It also allows synchronizing the actions. All communication and synchronization are done without sharing the same address space. An example of inter-process communication is a chat program on World Wide Web (WWW). Here two processes communicate across the network. IPC can be done using message passing system.

Message passing system: This system allows two processes to communicate without using the shared data. The communication among processes is done by using message passing system. The message passing system provides some operations like send and receive. The send operation is used to send a message to a process. The receive operation is used to receive a message from a process. Messages can be of fixed length or variable length. Dealing with fixed-length messages is simple but restricted for a complex system. Hence, variable size messages are used in the complex message passing system. For two processes to communicate there must be a communication link between them. A link can be implemented using shared memory bus, network, etc. A link can be logically implemented by using send and receive primitives. Let us look at various methods of implementing a link logically.

NOTES

NOTES

- **Direct communication:** In this scheme, if a process wants to communicate with other process, then it must explicitly use the name of the process with which it intends to communicate. For example, if P wants to communicate with Q, then it does so by using send (Q, message) or receive (Q, message).

- i) send (P, message): send a message to process P
- ii) receive (Q, message): receive a message from process Q

The first primitive is the sender initiated request. Here Q is the sender and intends to send a message to process P. The second primitive is receiver initiated request. Here the process P is the receiver and it intends to receive the message from Q.

The above primitives, send and receive, must explicitly name each other for communication. This scheme is known as symmetrical addressing. In asymmetrical addressing, the sender has to name the recipient and recipient is not required to name the sender.

In asymmetric addressing, the primitives are used in the following manner:
Send (P, message), Receive (id, message). Here, the sender is explicitly naming the recipient, whereas the receiver is receiving the message from id, which can be set in the process.

- **Indirect addressing:** Two processes communicate or exchange messages via mailboxes or ports. Mailboxes are objects, which are used to receive or send a message to a process. Every process has a mailbox and each mailbox has an id that can be uniquely identified.

The syntax for the send and receive primitives in indirect communication are as follows:

- Send (A, message): send a message to the A's mailbox
Receive (A, message): receive a message from A's mailbox

In direct communication, if a process wishes to communicate with the other then it has to name the process. In indirect communication the process communicates with the mailboxes

Mailbox can be owned by a process or the operating system. If the process P owns the mailbox then it is called the owner of the mailbox. And all other processes become the users of P's mailbox. P has privileges on its mailbox. The communication among processes is done using user mailboxes.

If Operating System owns the mailboxes then it provides mechanism to create the mailbox, send-receive messages, and delete the mailbox.

Synchronous communication: In the above section, we have seen how the processes communicate using the send and receive primitives. The communication can be implemented either in blocking or non-blocking. Blocking communication is called synchronous communication and non-blocking communication is called asynchronous communication.

The various possibilities with blocking and non-blocking communication are as follows:

Block send: Once the process P sends a message to the process Q, then P is blocked until Q receives the message. After sending the message to the process Q, the process P is blocked until Q receives the message.

Block receive: Once the process P receives a message from the process Q, P is blocked until Q's message is received.

Non-block send: The sending process sends a message and continues to operate. The sending process is not blocked.

Non-block receive: A receiver continues to operate even if the message from a process is not received. The receiver is not blocked.

Buffering: When a message is sent to a process, it resides in a temporary queue. The queue can be implemented in the following three ways:

- **Zero capacity:** The length of the Q is zero, i.e. the messages are not stored but directly given to the process.
- **Bounded capacity:** The length of the Q is finite, say n. Then it can store n messages. When a queue is full, the sender must be blocked. If the queue is empty, then messages can be sent.
- **Unbounded capacity:** The length of the queue is infinite. Here any number of messages can be sent as the queue length is infinite. The sender can send any number of messages without being blocked. The unbounded capacity is theoretically infinite but practically limited to the memory limits.

Check Your Progress

8. Which of the two operations are provided by the IPC facility?
9. Elaborate on the critical section problem.
10. Explain solution for critical section problem.
11. State the difference between symmetric and asymmetric direct communication.
12. Differentiate between the bounded and unbounded buffer.

3.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Interrupts are events that are used to stop the current execution and start a new urgent execution.

NOTES

2. This forms an important part of the Real time system because if a process arrives with greater priority then it raises an interrupt and the other process is stopped and the interrupt will be serviced.
3. (i) Programmed data transfer or I/O
(ii) Interrupt-driven I/O
(iii) Direct Memory Access (DMA) I/O
4. Direct memory access (DMA) is a feature of modern computers and microprocessors that allows certain hardware subsystems within the computer to access the system memory for reading and/or writing independently of the CPU.
5. The program-controlled branching and CPU-forced branching are the two methods used for branching into input/output routine in a computer.
6. Interrupts include the following three errors:
 - (i) $Ip=0$: undefined instruction
 - (ii) $Ip=1$: illegal instruction in user mode
 - (iii) $Ip=2$: logical address \geq bound register
7.
 - Mutual Exclusion: If P is executing in its critical section, then no other process can execute in their critical section.
 - Progress: When a process wants to enter into its critical section, then the processes that are not executing in their remainder sections make the decision.
 - Bounded waiting: If a process makes a request to enter into its critical section, then it must get the chance to enter into the critical section in finite time.
8. Two operations provided by the IPC facility are receive and send messages. Exchange of data takes place in cooperating processes.
9. The critical section problem states that when one process is executing in its critical section, then no other process is allowed to enter into its critical section. This implies that the execution of critical section is mutually exclusive. Before entering into the critical section, a process must request permission. The code that implements the request is called entry section.
10. The solution to the critical section problem must satisfy the following requirements:
 - (i) Mutual exclusion: If P is executing in its critical section, then no other process can execute in their critical section.
 - (ii) Progress: When a process wants to enter into its critical section, then the processes that are not executing in their remainder sections make the decision.
 - (iii) Bounded waiting: If a process makes a request to enter into its critical section, then it must get the chance to enter into the critical section in finite time.

NOTES

11. In symmetric direct communication, both sender and receiver process need to know each other's PID. On the other hand, in asymmetric direct communication, only the sender process needs to know PID of the receiver process but the receiver process need not know the PID of the sender process.
12. • **Bounded capacity:** The length of the Q is finite, say n. Then it can store n messages. When a queue is full, the sender must be blocked. If the queue is empty, then messages can be sent.
• **Unbounded capacity:** The length of the queue is infinite. Here any number of messages can be sent as the queue length is infinite. The sender can send any number of messages without being blocked. The unbounded capacity is theoretically infinite but practically limited to the memory limits.

3.5 SUMMARY

- Interrupts are events that are used to stop the current execution and start a new urgent execution.
- An interrupt first, saves the contents of program counter in the stack. Then the processor enters a subroutine called Interrupt Service Subroutine (ISS).
- CPU interrupt handling enables or disables interrupts in the CPU.
- Each device has an Interrupt ReQuests (IRQ) number which is based on Program Controlled Interrupt (PIC).
- There are two available schemes as master and slave which give the priority from highest to lowest as IRQ0-1, IRQ8-15, and IRQ3-7.
- PIC is accessed by port addresses such as 0x20 and 0x21 for master. IRQ devices generate a signal on IRQn, where n is the number of request.
- I/O device interrupts are used for data transfers between computer memory and I/O devices.
- Hardware timer interrupts are used to record time and distribute the processor time among different applications currently loaded for execution.
- Handle page or segment missing events (page or segment faults) in memory management for loading pages or segments from secondary storage.
- Attempts to access Illegal memory, attempts to execute illegal instructions, and attempts to divide by zero causing arithmetic overflow, etc. raises exception error interrupt and transfers execution to exception handler to abort the current process.
- Interrupts must be enabled so that they can respond to their events.
- Most of the devices are connected or interfaced to the computer system as one or more memory locations.

NOTES

- The interfacing lines are connected to the system buses through appropriate I/O ports present in both the CPU box and device controller part of the devices.
- There are two ways of interfacing I/O devices; memory mapped I/O and I/O mapped I/O.
- In programmed data transfer technique, the CPU first initiates the transfer and checks continuously in a busy loop waiting for *data ready* signal (for reading) or *ready to accept* signal (for writing) from the device. This is called polling.
- In interrupt-driven data transfer, the CPU initiates the data transfer and enables the device interrupt to interrupt CPU execution when the data is ready or the device is ready to accept data. The CPU then executes other jobs. When the device is ready for the transfer, it will inform the CPU (operating system) through the interrupt. The interrupt handler will then do the actual data transfer between device and memory.
- In a program-controlled branch to an I/O routine, a properly written program has address information for the branch-to location and the return address, if it is required.
- In a CPU-forced branch, it is possible to branch to the proper subroutine and return via another branch to the interrupted program at the end of the subroutine.
- The I/O device causing an interrupt is determined by examination via programming of the interrupt-level status word, when necessary. Then, after the device is known, the device-status word is examined to determine the condition in the device that caused the interrupt to occur.
- The microprocessor without interlocked pipeline stages (MIPS) OS contains code in distribution and runs on UNIX systems.
- A number of changes are possible in the low-level code that interfaces to the hardware. The Java OS code in distribution runs on Java 1.1 systems and Java 1.1 browsers. A number of changes are possible in the low-level code that interfaces to the hardware.
- The MIPS OS and Java OS almost use same hardware interface.
- When two or more processes try to access and manipulate the same data simultaneously, then we call this concurrent access to shared data.
- Concurrent access and manipulation to shared data introduce the concurrency problems.
- To solve the concurrency problems, such as race condition, we use process synchronization.
- A semaphore is a synchronization tool. It is an integer variable. It can be initialized and two atomic operations are possible with semaphore; they are wait and signal.

NOTES

- Inter-process communication allows a process to communicate with other processes. It also allows synchronizing the actions.
- The I/O device causing an interrupt is determined by examination via programming of the interrupt-level status word, when necessary.
- Interrupts include system call that executes syscall program executed and a timer. The timer register starts from 1 to 0 and a non-zero timer counts down every microsecond.
- When two or more processes try to access and manipulate the same data simultaneously, then we call this concurrent access to shared data.
- Concurrent access and manipulation to shared data introduce the concurrency problems. To solve the concurrency problems, such as race condition, we use process synchronization.
- A semaphore is a synchronization tool. It is an integer variable. It can be initialized and two atomic operations are possible with semaphore; they are wait and signal.
- Inter-process communication allows a process to communicate with other processes. It also allows synchronizing the actions.
- All communication and synchronization are done without sharing the same address space. An example of inter-process communication is a chat program on World Wide Web (WWW).
- Two processes communicate or exchange messages via mailboxes or ports. Mailboxes are objects, which are used to receive or send a message to a process
- The communication can be implemented either in blocking or non-blocking. Blocking communication is called synchronous communication and non-blocking communication is called asynchronous communication.
- The length of the queue is infinite. Here any number of messages can be sent as the queue length is infinite.

3.6 KEY WORDS

- **Interrupt:** It refers to an event used to stop the current execution and start a new urgent execution.
- **Direct Memory Access (DMA):** Direct memory access (DMA) is a feature of modern computers and microprocessors that allows certain hardware subsystems within the computer to access the system memory for reading and/or writing independently of the CPU.
- **Page Fault (PF):** When a program accesses some addresses which are not in physical RAM, an interrupt file called Page Fault (PF) is generated.

NOTES

- **Programmable Interrupts Controller (PIC):** The Programmable Interrupt Controller (PIC) handles hardware interrupts.
- **Race Condition:** if several processes are manipulating the shared data simultaneously, the outcome depends on the order in which the processes finish execution. This is called race condition.
- **Synchronization:** Synchronization means sharing system resources by processes in a, such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.
- **Semaphores:** A semaphore is a synchronization tool. It is an integer variable. It can be initialized and two atomic operations are possible with semaphore; they are wait and signal.

3.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Write a short note on three techniques used for transferring data between memory and I/O devices.
2. Which type of interrupts Programmable Interrupts Controller (PIC) handles?
3. State the features of Interrupt Service Routine (ISR).
4. Differentiate between program-controlled branching and CPU-forced branching.
5. Which methods are used to determine what condition is causing interrupts?
6. In which situation several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order?
7. In which segment of code the process may change common variables, update tables, write into files?
8. What is the use of interprocess communication?
9. Define race condition.

Long-Answer Questions

1. What is interrupts? How it is handled by an OS?
2. In Operating System (OS) where interrupts are mainly used?
3. Elaborate the basic operations of interrupts service.
4. Explain the required steps to handle an interrupts.

NOTES

5. Illustrate the features of Interrupts Service Routine (ISR).
6. Classify the ways of process communication in message passing systems.
7. What is critical section problem? What are the requirements that a solution to the critical section must satisfy?
8. What will be the sequence of execution of send () and receive () calls in both cases?
9. Describe in detail various possibilities with blocking and non-blocking communication.
10. Write short notes on:
 - (i) Hardware interrupts
 - (ii) Causes of interrupts
 - (iii) Program-controlled branching

3.8 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.
- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.
- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.
- Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.
- Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.
- Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

BLOCK - II
ASYNCHRONOUS CONCURRENT EXECUTION

UNIT 4 MUTUAL EXCLUSION

Structure

- 4.0 Introduction
 - 4.1 Objectives
 - 4.2 Mutual Exclusion
 - 4.3 Implementing Mutual Exclusion Primitives
 - 4.4 Answers to Check Your Progress Questions
 - 4.5 Summary
 - 4.6 Key Words
 - 4.7 Self Assessment Questions and Exercises
 - 4.8 Further Readings
-

4.0 INTRODUCTION

In computer science, mutual exclusion is a property of competition control developed for the purpose of avoiding the conditions of race. The requirement is that one execution thread never enters its critical section at the same time that another execution concurrent thread enters its own critical section, which refers to an interval of time during which an execution thread, such as shared memory, accesses a shared resource.

This problem (called a race condition) can be avoided by using the mutual exclusion rule to ensure that no simultaneous changes can occur to the same section of the list.

The issue raised by mutual exclusion is a resource sharing problem: how can a software system control the access of multiple processes to a common resource, when each process needs exclusive control of that resource while doing its job. The mutual-exclusion approach to this only makes the shared resource accessible when the process is in a particular segment of code called the critical section. By managing each collective execution of the portion of its programme where the resource can be used, it manages access to the shared resource.

Most mutual exclusion algorithms are constructed with the assumption that there is no failure while a process is running within the critical segment. In fact, however, such failures can be common place. A sudden loss of power or defective interconnection, for example, may cause a process to encounter an unrecoverable error in a critical section or otherwise be unable to proceed. If such a failure occurs, traditional mutual exclusion algorithms that are non-failure-tolerant can

deadlock key liveness properties or otherwise fail. Several solutions using crash-recovery mechanisms have been proposed to deal with this problem.

A variety of mutual exclusion algorithms, with different performance parameters and different techniques, are available in the literature. A key point is the selection of a 'nice' mutual exclusion algorithm. It is possible to broadly divide these mutual exclusion algorithms into token-based and non-token-based algorithms.

In this unit, you will study about the basic concepts of Mutual Exclusion and implementation of mutual exclusion primitives.

4.1 OBJECTIVES

After going through this unit, you will be able to:

- Introduce the basic concepts of mutual exclusion
- Elaborate on the concepts of mutual exclusion process in detail
- Discuss about the requirements for mutual exclusion
- Provide an overview of problems occurred in mutual exclusion
- Explain the implementation of mutual exclusion

4.2 MUTUAL EXCLUSION

Mutual exclusion is a function of competition regulation, which is set up for the purpose of eliminating race conditions. It is the requirement that one execution thread never enters its critical section at the same time as another execution concurrent thread enters its own critical section, referring to an interval of time during which an execution thread, such as shared memory, accesses a shared resource.

Processes waiting to enter the critical section use the CPU to keep checking if they can enter their critical section. The act of using the CPU to repeatedly check for entry to the critical section is called busy waiting. To avoid busy waiting, resources are created to enforce mutual exclusion and make them accessible by the operating system. Processes waiting to enter a critical section will be put in the blocked state, such as they waited for I/O so that other processes can continue doing useful work. Mutual exclusion makes sure that concurrent process access shared resources or data in a serialized way. If a process, say P_i , is executed in its critical section, then no other processes can be executing in their critical sections, for example updating a Deadlock Buffer (DB) or sending control signals to an Input/Output (I/O) device. Following are the requirements for mutual exclusion:

- A process must not be delayed access to a critical section when there is no other process using it.

Mutual Exclusion

NOTES

NOTES

- No assumptions are made about relative process speeds or number of processes.
- A process remains inside its critical section for a finite time only.
- Only one process at a time is allowed in the critical section for a resource.
- A process that halts in its noncritical section must do so without interfering with other processes.
- No deadlock or starvation is allowed.
- A process must not be delayed access to a critical section when there is no other process using it.
- No assumptions are made about relative process speeds or number of processes.
- A process remains inside its critical section for a finite time only.

The most powerful method available for mutual exclusion is the disabling of interrupts. Such a lock guarantees exclusive access to the CPU is coded in C language as follows:

```
funcA ()  
{  
    int lock = intLock();  
    ....  
    /* critical region of code that cannot be interrupted */  
    ....  
    intUnlock (lock);  
}
```

Following code is required for hardware support including test and set instruction for mutual exclusion:

```
/* Program Mutual Exclusion for test and set instruction */  
  
const int n = /* Number of processes starts*/;  
int bolt;  
void P (int i)  
{  
    while (true)  
    {  
        while (!testset (bolt))  
            /* Do not perform operation */  
        /* Critical Section */  
        bolt = 0;
```

| |

```
/* If remainder returns*/
```

Mutual Exclusion

```
}
```

```
}
```

```
void main()
```

```
{
```

```
    bolt =0;
```

```
    parbegin (p(1), p(2), ..., p(n));
```

```
}
```

Following C code is required for hardware support including exchange instruction for mutual exclusion:

```
/* Program Mutual Exclusion for exchange instruction */
```

```
int const n = /* Number of processes starts*/;
```

```
int bolt;
```

```
void P (int i)
```

```
{
```

```
    int keyi;
```

```
    while (true)
```

```
{
```

```
    keyi =1;
```

```
    while (keyi !=0)
```

```
        exchange(keyi, bolt);
```

```
        /* Critical Section Begins*/
```

```
        exchange(keyi, bolt)
```

```
        /* If remainder returns*/
```

```
}
```

```
}
```

```
void main()
```

```
{
```

```
    bolt =0;
```

```
    parbegin (p(1), p(2), ..., p(n));
```

```
}
```

The mutual exclusion problem involves an analytic mode to solve the problem which usually means a solution technique. This process allows writing a functional relation between system parameters and a chosen performance measure in terms of processes that are analytically or numerically solvable. It supports queuing theory for various resources, such as CPU, memory, I/O channels and devices. A major

NOTES

function of operating system is to manage the use of resources among many programs.

NOTES

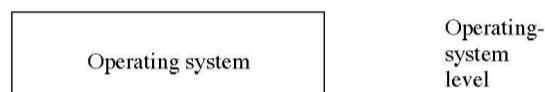
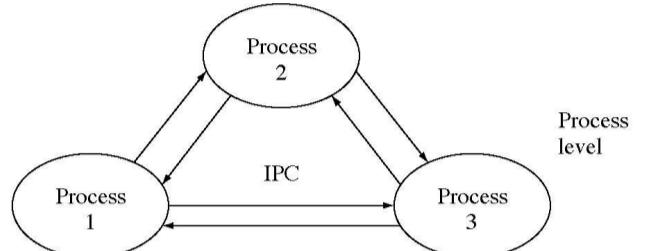


Fig. 4.1 IPC at Process Level

In Figure 4.1, it is shown that the three processes are controlled in assembled operating system through IPC. These operating system processes are set at operating system level whereas all the processes are set at process level. Figure 4.2 represents the concept how two process mutual exclusion can take place in the operating system. Here, Process A and Process B maintain a message queue through three methods namely, `ReceiveMessage()`; , `UseFile()`; and `SendMessage()`. The processes are decided after getting the received messages. These received messages are collected in the process which is used in method `UseFile`. The last message is sent to the clients. This concept can be understood with the help of example.

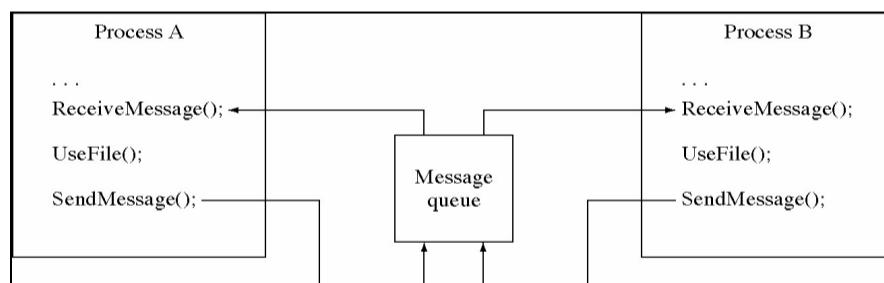


Fig. 4.2 Mutual Exclusion Process

Let us say the messages are solved at the user-level but the mutual exclusion is needed to be implemented at the operating system level. So, this step is considered to reduce a problem in a special case. For example, all the commands and events which used at command prompt are not easy to remember but this problem can be solved if use Help menu or Help command for specific command of the running application. The reentrant programs are the two threads that execute the same code simultaneously. The programs are not thread safe if programs with embedded

data are not reentrant. The programs can be made as reentrant by removing the embedded data and allocated them for each execution. Recent versions of Microsoft Disk Operating System (MS DOS) are not reentrant but in the object oriented programs the data area of each object is allocated as heap store therefore they are automatically thread safe which means reentrant. A model is required to represent the system. The model could be an equation, a simulation, etc. It is used to generate the idea whether the system is useful to validate the data or not. The operating system receives two queues for non-blocking heap which uses threads or processes by using scheduler thread facility. If mutual exclusion is added, either the disable interrupts function is implemented or exchanged word instruction. Adding monitors are taken as extra option. Reusable patterns are considered as a typical problem with a confirm solution. A model is required to represent the system. The model could be an equation, a simulation, etc. It is used to generate the idea whether the system is useful to validate the data or not. The OS receives two queues for non-blocking heap that uses threads or processes by using ‘scheduler thread’ facility. A general problem defined in OS can be taken as more than once whereas a solution is given as to work well with the system. For example, IPC provides a typical way to use IPC and design pattern which arranges the objects to solve common problems and in frameworks skeleton code is used to solve common problems. The all IPC patterns are set in OS in that way that processes do not fail at critical times but sometimes processes do fail in networks. The solutions are hard but the portability is reduced if a single process failure that can cause the entire system to fail. To overcome this problem, fault tolerant server system is used. This mechanism can be referred to as adding a new facility to the system. Figure 4.3 displays the mutual exclusion example in which a subroutine is broken down into the four parts known as remainder, entry, critical and exit.

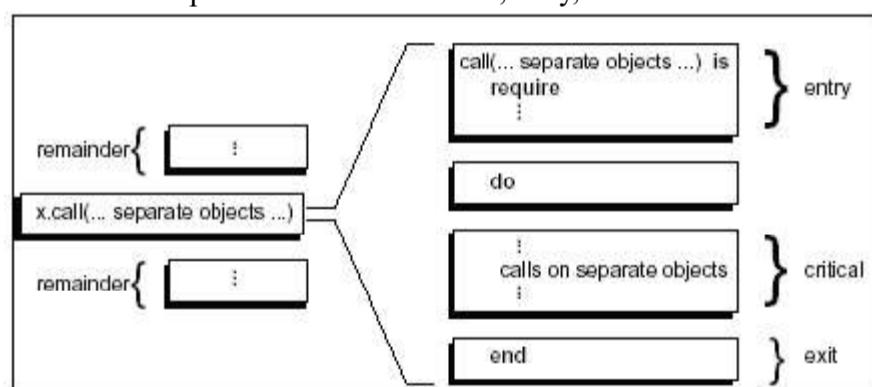


Fig. 4.3 Problem Occurred in Mutual Exclusion

The mutual exclusion problem was first described and solved by Dijkstra algorithm. The four parts in the Figure 4.3 represents the various stages which return to remainder. In the all four stages, the execution cycle takes place in the processor. The remainder basically represents the non-critical section in which code does not call separate objects. The entry stage refers to the mutual exclusion solution if, problem occurs. The `call (... separate objects ...)` function provides

NOTES

the algorithm that uses actual arguments to call. This stage is known as critical section which contains some code to protect the mechanism of mutual exclusion. For this, separate calls are attempted for separate objects. The exit stage is executed by writing the code which helps to free the resources. These resources are obtained exclusively and hence obtained in almost each section including one-level and two-level implementation. The role of system calls is here to solve the problem of mutual exclusion. These are special machine instructions that cause an interrupt by `syscall`, `trap` and `svc` commands. These are assembly language functions and not generated by High Level Languages (HLLs). If process makes a system call, the operating system calls a scheduler thread that is implemented in kernel. The following code shows how system calls happen in various OS:

```
int load_hdr(char *name, char buf)
{
    int fd;
    fd = open(name, O_RDONLY);
    if(fd<0)
    {
        return(fd);
    }
    n = read(fd,buf,HDR_SIZE);
    close(fd);
    return(n);
}
```

In the above code, the file is opened for reading. The `open()` system call returns the integer file descriptor that is used for calling the other file related to it. In the `read()` system call, the OS is requested to copy data from file to memory space at the location `buf` whereas parameter `fd` is the file name that has to be read and the third parameter `HDR_SIZE` is denoted by number of bytes of file.

Check Your Progress

1. Define the term mutual exclusion.
2. What is busy waiting?
3. Write the method available for mutual exclusion.
4. Which mode involves in solving mutual exclusion?
5. How to controlled IPC process level for mutual exclusion?
6. Program is thread safe or not?

4.3 IMPLEMENTING MUTUAL EXCLUSION PRIMITIVES

It must implement *mutual exclusion*: only one process can be in the critical section at a time.

Peterson's algorithm is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.

Problem: Given 2 processes I and j, you need to write a programme that can guarantee mutual exclusion between the two without any additional hardware support.

Solution: There may be many ways to solve this problem, but most of them need additional hardware support. Using Peterson's Algorithm for mutual Exclusion is the easiest and most common way to do this. It was invented by Peterson in 1981, while Theodorus Jozef Dekker, who created Dekker's algorithm in 1960, which was later improved by Peterson and became known as Peterson's Algorithm, did the initial work in this direction.

Peterson's algorithm (or Peterson's solution) is a simultaneous mutual exclusion programming algorithm that allows two or more conflict-free processes to share a single-use resource, using only shared communication memory.

Peterson's Algorithm

The algorithm uses two variables, flag and turn. A flag[n] value of true indicates that the process n wants to enter the critical section. Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting turn to 0.

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

```
P0: flag[0] = true;
P0_gate: turn = 1;
while (flag[1] == true && turn ==
1)
{
// busy wait
}
// critical section
...
// end of critical section
flag[0] = false;
```

```
P1: flag[1] = true;
P1_gate: turn = 0;
while (flag[0] == true &&
== 0)
{
// busy wait
}
// critical section
...
// end of critical section
flag[1] = false;
```

NOTES

NOTES

The algorithm satisfies the three essential criteria to solve the critical section problem, provided that changes to the variables turn, flag [0], and flag [1] propagate immediately and atomically. The while condition works even with pre-emption.

The three criteria are mutual exclusion, progress, and bounded waiting.

Since turn can take on one of two values, it can be replaced by a single bit, meaning that the algorithm requires only three bits of memory.

Check Your Progress

7. Who was described mutual exclusion problem first?
8. Why Peterson's algorithm implement?

4.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Mutual exclusion is a function of competition regulation, which is set up for the purpose of eliminating race conditions.
2. Processes waiting to enter the critical section use the CPU to keep checking if they can enter their critical section. The act of using the CPU to repeatedly check for entry to the critical section is called busy waiting.
3. The most powerful method available for mutual exclusion is the disabling of interrupts.
4. The mutual exclusion problem involves an analytic mode to solve the problem which usually means a solution technique.
5. Three processes are controlled in assembled operating system through IPC. These operating system processes are set at operating system level whereas all the processes are set at process level.
6. The program is not thread safe.
7. The mutual exclusion problem was first described and solved by Dijkstra algorithm.
8. The Peterson algorithm is a simultaneous mutual exclusion programming algorithm that allows two or more processes to share a single-use resource without conflict, using only shared communication memory.

4.5 SUMMARY

- Processes waiting to enter the critical section use the CPU to keep checking if they can enter their critical section. The act of using the CPU to repeatedly check for entry to the critical section is called busy waiting.

- To avoid busy waiting, resources are created to enforce mutual exclusion and make them accessible by the operating system.
- **Mutual exclusion** is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process cannot enter its critical section while another concurrent process is currently present or executing in its critical section, i.e., only one process is allowed to execute the critical section at any given instance of time.
- A process must not be delayed access to a critical section when there is no other process using it.
- No assumptions are made about relative process speeds or number of processes.
- A process remains inside its critical section for a finite time only.
- Only one process at a time is allowed in the critical section for a resource.
- A process that halts in its noncritical section must do so without interfering with other processes.
- No deadlock or starvation is allowed.
- A process must not be delayed access to a critical section when there is no other process using it.
- No assumptions are made about relative process speeds or number of processes.
- The mutual exclusion problem involves an analytic mode to solve the problem which usually means a solution technique. This process allows writing a functional relation between system parameters and a chosen performance measure in terms of processes that are analytically or numerically solvable.
- Three processes are controlled in assembled operating system through IPC. These operating system processes are set at operating system level whereas all the processes are set at process level.
- Processes waiting to enter the critical section use the CPU to keep checking if they can enter their critical section. The act of using the CPU to repeatedly check for entry to the critical section is called busy waiting.
- The programs are not thread safe if programs with embedded data are not re-entrant.
- If mutual exclusion is added, either the disable interrupts function is implemented or exchanged word instruction. Adding monitors are taken as extra option.
- To avoid busy waiting, resources are created to enforce mutual exclusion and make them accessible by the operating system. Processes waiting to enter a critical section will be put in the blocked state, such as they waited for I/O so that other processes can continue doing useful work.

Mutual Exclusion

NOTES

NOTES

- The most powerful method available for mutual exclusion is the disabling of interrupts.
- The OS receives two queues for non-blocking heap that uses threads or processes by using ‘scheduler thread’ facility. A general problem defined in OS can be taken as more than once whereas a solution is given as to work well with the system.
- The mutual exclusion problem was first described and solved by Dijkstra algorithm.
- Only one process can be in the critical section at a time.

4.6 KEY WORDS

- **New state:** A process is said to be in a ‘new’ state if it is being created for the first time.
- **Process:** A process is the instance of a computer program that is being executed by one or many threads. It contains the program code and its activity.
- **Ready state:** A process is said to be in a ‘ready’ state if it is ready for the execution and waiting for the CPU to be allocated to it.
- **Running state:** A process is said to be in a ‘running’ state if CPU has been allocated to it and it is being executed.
- **Waiting state:** A process is said to be in a ‘waiting’ state if it has been blocked by some event.
- **Bounded-buffer problem:** In case of bounded buffer, the producer-consumer problem is also known as bounded-buffer problem.

4.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Elaborate the term of mutual exclusion with example.
2. Why mutual exclusion is required?
3. State the definitions of mutual exclusion problem.
4. Write the C code for hardware support including exchange instruction for mutual exclusion.
5. Which method is available for mutual exclusion?

Long-Answer Questions

Mutual Exclusion

1. Describe the requirements for mutual exclusion with the help of illustration and example.
2. Briefly illustrate the mutual exclusion process.
3. How to occurred problem for mutual exclusion with the help of a diagram?
4. Write a code which shows how system calls happen in various OS.
5. Briefly describe the Peterson's Algorithm.
6. Elaborate briefly an analytic mode to solve the mutual exclusion problem.

NOTES

4.8 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.
- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.
- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.
- Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.
- Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.
- Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

UNIT 5 MUTUAL EXCLUSION PROBLEM AND SEMAPHORES

Structure

- 5.0 Introduction
 - 5.1 Objectives
 - 5.2 Critical Section Problem
 - 5.2.1 Software Solutions for Critical Section Problem
 - 5.2.2 Hardware Solutions for Critical Section Problem
 - 5.2.3 Mutual Exclusion Algorithms
 - 5.3 Semaphores: Definition
 - 5.4 Answers to Check Your Progress Questions
 - 5.5 Summary
 - 5.6 Key Words
 - 5.7 Self Assessment Questions and Exercises
 - 5.8 Further Readings
-

5.0 INTRODUCTION

In a multiprocessing (a job executed as multiple processes) and a multithreaded environment, processes or threads have to cooperate in their execution while using shared resources like shared memory buffers. Cooperating processes or thread execution can affect or be affected by other processes/threads. So there should be some way to synchronize their execution to use shared resources; otherwise, unexpected errors may occur.

Concurrent access to shared resources can result in unintended or incorrect actions in concurrent programming, so areas of the program where the shared resource is accessed need to be secured in ways that prevent concurrent access. The critical segment or critical area is this safe portion. It cannot be implemented at one time by more than one operation. Usually, a shared resource, such as a data structure, a peripheral computer, or a network link, is accessed by the critical portion, which does not function properly in the case of multiple concurrent accesses.

The segment is known as a critical section if more than one method accesses the same code segment. The critical section includes shared variables or resources that are expected to be synchronised in order to preserve data variable consistency.

In simple terms, a critical section is a collection of instructions/statements or code region that must be executed atomically (read this post for atomicity), such as accessing a resource (file, input or output port, global data, etc.).

It is important to understand the significance of race condition when writing kernel mode programming (a device driver, kernel thread, etc.). Since the programmer can access and change the structures of kernel data directly.

A semaphore is a vector or abstract data type used in a concurrent system such as a multitasking operating system to manage access to a shared resource by multiple processes and avoid critical section issues. A trivial semaphore is a simple variable that, depending on programmer-defined conditions, is modified (for example, incremented or decremented or toggled).

In this unit, you will study about the software solution to the mutual exclusion problem, hardware solution to mutual exclusion problem and semaphore.

*Mutual Exclusion
Problem and
Semaphores*

NOTES

5.1 OBJECTIVES

After going through this unit, you will be able to:

- Describe the critical section problem
- Know about the producer-consumer problem
- Explain the requirements for solving critical section problem
- Describe the hardware and software solutions for critical section problem
- Understand the mutual exclusion algorithm
- Discuss about the semaphore

5.2 CRITICAL SECTION PROBLEM

A critical section is typically used when a multi-threaded program must update multiple related variables without a separate thread making conflicting changes to that data. In a related situation, a critical section may be used to ensure that a shared resource, for example, a printer, can only be accessed by one process at a time.

In a multiprocessing (a job executed as multiple processes) and a multithreaded environment, processes or threads have to cooperate in their execution while using shared resources like shared memory buffers. Cooperating processes or thread execution can affect or be affected by other processes/ threads. So there should be some way to synchronize their execution to use shared resources; otherwise, unexpected errors may occur. We will illustrate the problem using the classical producer-consumer synchronization problem for sharing buffers. The following discussion applies equally to threads and processes.

NOTES

Producer-Consumer Problem

The classical producer-consumer synchronization problem is shown in Figure 5.1.

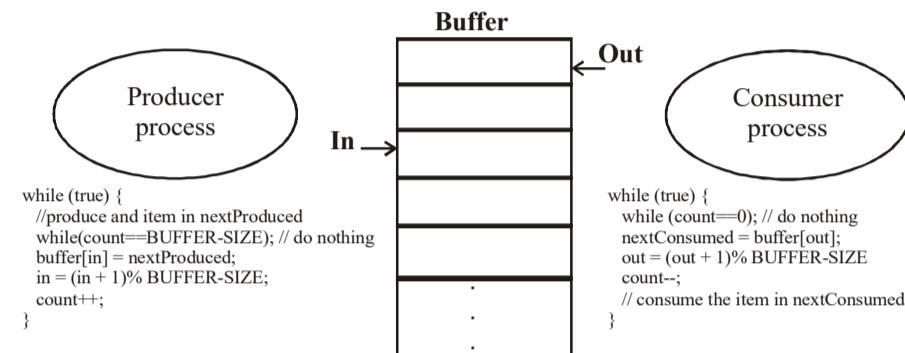


Fig. 5.1 Producer-Consumer Problem

There are two processes, one called producer process and the other called consumer process. Buffer-size is the number of buffers available for placing items (data). The producer process produces items (data) and places them into empty buffers. The consumer process takes the item from the occupied buffers and does some processing. The next empty buffer is indicated by an **In** pointer and the next item (buffer) to be consumed is indicated by the **Out** pointer. There is a variable **count** to indicate the number of full buffers at any time.

The codes executed by the Producer and Consumer processes for the synchronization of buffer accesses are given below:

Producer Process

```
while (1) {
    /* produce an item and place in nextItem */
    while (count == BUFFER-SIZE) ; /*wait for one
empty buffer*/
    buffer [In] = nextItem;
    In = (In + 1) % BUFFER-SIZE;
    count++;
}
```

Consumer Process

```
while (1) {
    while (count == 0) ; /*wait for at least one full
buffer*/
    NextItemConsumed = buffer[Out];
    Out = (Out + 1) % BUFFER_SIZE;
    count--;
    /* consume the item in nextItemConsumed */
}
```

NOTES

The producer should check for finding an empty buffer before placing an item into it. It places the item to the buffer indicated by `In` pointer. The producer then makes the `In` variable point to the next empty buffer and increments the number of full buffers (`count`).

The consumer should check for the availability of any item not yet consumed before transferring the item pointed to by the `Out` pointer. Zero value of `count` indicates that there is no new item in the buffer. Consumer then increments the `Out` pointer and decrements the number of full buffers (`count`).

This `count` variable is updated by the producer and the consumer processes. What will happen if both the processes try to update the `Count` at the same time?

The increment operation, `count++` of the producer is implemented by the following machine codes:

```
register1 = count
register1 = register1 + 1
count = register1
```

The decrement operation, `count--` of the consumer is implemented by the following machine codes:

```
register2 = count
register2 = register2 - 1
count = register2
```

In a multithreaded/processing system, the increment and decrement operations may get interleaved due to time-out interruptions. Consider the interleaving of the above execution, initially with a value of `count = 8`. The producer increments once and the consumer decrements once, but the execution is interleaved as given below:

```
S0: producer execute register1 = count; // {register1 = 8}
S1: producer execute register1 = register1 + 1; // {register1
     = 9}
S2: consumer execute register2 = count; // {register2 =
     8}
S3: consumer execute register2 = register2 -1; // {register2
     = 7}
S4: producer execute count = register1; // {count = 9 }
S5: consumer execute count = register2; // {count = 7}
```

The result we got is 7 instead of the actual result 8. So, the result of execution (the value of `count`) depends upon the way they are interleaved.

This is called **race condition**. The wrong value obtained is due to interleaved execution of increment and decrement operations on a shared variable. You would have got the correct value, if all the three instructions required for increment (or decrement) were executed without any interruption.

Java language provides a keyword `synchronized` to solve this race condition. Such language-based support is not available in the lower-level languages.

NOTES

These are to be implemented mostly using basic programming primitives, even though some systems support hardware instructions for the same to build the solutions easily.

The above-described problem is known as a Critical Section (CS) problem in computer science. Here, critical section means the code that is used to access the shared data structures by more than one process or thread normally in cooperated multitasking or multiprocessing or multithreaded computation.

Requirements of Solutions to CS Problem

A good acceptable solution to critical section problem must satisfy the following requirements:

Mutual exclusion: At a time only one process/thread is allowed to execute the critical section code. In other words, processes/threads must execute critical sections in a mutually exclusive fashion.

Progress requirement: When no process/thread is executing in the critical section, only processes/threads waiting for entry to the critical section should decide which one should enter the critical section next. This decision should be taken in such a way as to ensure progress in execution.

Bounded waiting: Processes/threads must be allowed to enter the critical section in bounded time or without any starvation.

Following is the basic scheme proposed for the solutions of a critical section problem:

The code of each of the processes must be structured or sectioned as in this basic scheme.

```
Repeat
{
    /* place entry section code */
    entry section
    /* place critical section code */
    critical section
    /* place exit section code */
    exit section
    /* place remainder section code */
    remainder section
}
until false;
```

The purposes of each of the sections are explained as below.

Entry Section: This code is the same for all processes wishing to enter CS for sharing the same data structure. This is the code that any process/thread must execute before entry to the critical section code. This code must verify that there is no process currently executing in critical section code and must also ensure the **progress** and **bounded waiting** requirements in conjunction with the **exit section** code.

Critical Section: This code may be the same or different for different processes. This stands for the code that accesses the shared data structures: for example, the code for incrementing or decrementing count variable in producer-consumer problem. At a time only one process can execute this code. That is, when a process is executing this code no other process will be allowed to execute this code.

Exit Section: This code is the same for all processes wishing to use CS for sharing the same data structure. The code is executed when processes leave CS. This may be to change the flag to indicate that the process that entered CS has finished executing CS and is not in CS.

Remainder Section: This is the code specific to the application's processing. That is, it is different for each of the processes.

There are two classes of solutions for CS problem, one using software and the other using special hardware instructions. In the next two subsections, we will examine some solutions to critical section problem in both of these classes.

5.2.1 Software Solutions for Critical Section Problem

Two Process Solutions

Solution 1: Following is the solution.

```
int turn = 0;          /* shared control variable */
/* Code structure for Process Pi, i is 0 or 1
/* start entry section
while (turn != i); /* busy wait: while loop goes on
executing as long as turn is not equal to i. This is the
entry section code */
Critical Section (CS) code for Pi;
turn = 1 - i; /* exit section code*/
Remainder Section code for Pi
```

Pros and Cons

Mutual exclusion requirement is guaranteed in the above solution, but progress and bounded wait requirements are not satisfied. For the solution to work properly, the processes should enter the critical section alternately. That is, if the first process that exits from CS sets the turn to that of the other process, the other process should then enter CS before the first process can again enter it. Suppose the second process does not need entry again for a long time, then the first process must wait on the Entry section. That is, a process that is not waiting for entry to CS is deciding when the other process should enter CS, thus violating progress requirement. Also, if one of the processes terminates execution, the other cannot enter CS, thus violating bounded waiting requirement.

Solution 2: In this solution, the strict alternation requirement of Solution 1 is removed.

```
int flag[2] = { FALSE, FALSE }; /* true value of flag[i]
tells that Pi is executing in its CS */
```

NOTES

NOTES

```
// Structure of Pi
//enter entry section
while (flag[1 - i]) ; /* entry section code*/
flag[i] = TRUE; /* entry section code*/
// enter CS
CriticalSection code for Pi;
flag[i] = FALSE; /* Exit section code */
Remainder Section code for Pi
```

Pros and Cons

1. When both the flags are false, two processes trying for entry to CS simultaneously can win entry to CS at the same time. So, mutual exclusion is violated.
2. Progress is satisfied.
3. Bounded wait is also satisfied.

Solution 3: This solution tries to enforce mutual exclusion.

```
int flag[2] = { FALSE, FALSE }; /* true value of flag[i]
tells that Pi is executing in its CS */
// Structure of Pi
//enter entry section
flag[i] = TRUE; /* entry section code*/
while (flag[1 - i]) ; /* entry section code*/
// enter CS
CriticalSection code for Pi;
flag[i] = FALSE; /* Exit section code */
Remainder Section code for Pi
```

Pros and Cons

1. The above solution satisfies mutual exclusion, but
2. The progress requirement is violated. If both the flags are set, both the processes will be executing in entry section forever without winning entry to CS. This is deadlock situation.
3. Also, the solution does not satisfy bounded wait requirement.

Solution 4: This solution tries to eliminate the deadlock situation in Solution 3.

```
int flag[2] = { FALSE, FALSE }; /* true value of flag[i]
tells that Pi wishes to enter its CS */

/* Structure of Pi */
/*enter entry section */
flag[i] = TRUE; /* entry section*/
while (flag[1 - i]) { /*entry section*/
```

```

        flag[i]=FALSE;
        delay(); /*makes process pi sleep for a fixed
duration*/
        flag[i]=TRUE;
    }
/*enter CS */
CriticalSection code for Pi;
flag[i] = FALSE; /* Exit section code*/
Remainder Section code for Pi

```

NOTES

Pros and Cons

1. Guarantees mutual exclusion requirement.
2. Violates progress requirement.
3. Violates bounded wait.

Peterson's Algorithm

```

int flag[2] = { FALSE, FALSE }; /* true value of flag[i]
tells that Pi wishes entry to its CS */
int turn =0; /* turn= I tells that process I has priority
to enter CS. Initially priority is given to process 0 */
/*Structure of Pi */
/* enter entry section */
flag[i] = TRUE; /* entry section*/
Turn =1 - i; /* Pi gives priority to the other process
Pj, j=1-i */
while ((flag[1 - i]) && (turn== 1-i)); /* busy wait*/
/* enter CS */
CriticalSection code for Pi;
flag[i] = FALSE; /* Exit section code*/
Remainder Section code for Pi

```

Pros and Cons: This solution satisfies all the requirements for the critical section problem.

Multiple Process Solution

A popular multiple process solution is the Bakery algorithm for n process solution. The solution is given below:

```

int choosing[NPROCS] = { FALSE, FALSE, FALSE,... } ;
int number[NPROCS] = { 0,0,0,... };
/* Structure of Pi */
/*enter entry section */
choosing[i] = TRUE;
number[i] = max(number) + 1;
choosing[i] = FALSE;
for (j = 0; j < NPROCS; j++) {

```

NOTES

```
while (choosing[j]) ; /* busy wait */  
while ((number[j] != 0 && (number[j] < number[i] ||  
    number[j] == number[i] && j < i)) ; /*  
busy wait */  
}  
/* enter CS */  
CriticalSection code for Pi;  
number[i] = 0; /* Exit section code*/  
Remainder Section code for Pi
```

5.2.2 Hardware Solutions for Critical Section Problem

There are two common methods for hardware solutions:

- Disabling interrupts so that mutual exclusion can be achieved easily as there will not be any interrupt to pre-empt the currently executing process from CS.
- By using special hardware instructions, if provided by the hardware.

These two hardware solutions are described below.

Disabling Interrupts

The two primitive functions required are as follows:

```
mutexbegin()  
{  
    disable_ints;  
}  
mutexend()  
{  
    enable_ints;  
}
```

Before entry to critical section, disable interrupts by calling `mutexbegin()`. After exiting from critical section code call `mutexend()`.

Drawback: Disabling interrupts for long periods of time leads to under utilization of resources.

Special Machine Instructions

Many machines provide at least one of the following special hardware instructions to help in the solution of CS problem:

- Test-and-set instruction
- Swap instruction
- Fetch-and-increment instruction

Any of these may be used for the solution of CS problems. The following will describe the solutions using test-and-set and swap instructions.

Test-and-Set Instruction

Text-and-Set or TS instruction provides a direct hardware support to mutual exclusion. It allows only one concurrent process to enter the critical section. It is able to make the programming task easier. It also improves the efficiency of the system.

The critical-section problem can be solved simply in a uniprocessor environment if you are able to prevent the occurrence of interrupts during the modification of a shared variable. Hence, the current sequence can be allowed to be executed without pre-emption. This is the approach taken by non-pre-emptive kernels.

Unfortunately, this solution is not feasible in a multiprocessor environment. Message passing to all the processors could be time-consuming in case of disabling the interrupts in a multiprocessor environment. As entry into the critical section is delayed due to such a message passing, the efficiency of the system decreases.

Therefore, many modern systems provide a special hardware support which helps in testing and modifying the content of a word or in swapping the contents of two words automatically.

This mechanism sets the global variable to 0, which indicates that the shared resource is available for being accessed. Each process has to execute the TS instruction in order to use the resource available with a control variable as an operand. As a principle, TS takes one operand, the address of the control variable or a register, which may act as a semaphore.

You can implement the wait operation on the semaphore variable S using TS instruction set. However, it is possible only in case of the availability of the set with the supporting hardware.

Test-and-Set instruction:

```
boolean TestAndSet (int S) {  
    if (S == 0) {  
        S = 1;  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

The importance of these instructions is that it executes automatically. Thus, if two TestAndSet () instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

*Mutual Exclusion
Problem and
Semaphores*

NOTES

NOTES

Advantages

The advantages of the TS instruction are as follows:

- It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.
- It is simple and, hence, easily verifiable.
- It can be used to support multiple critical sections.

Disadvantages

The disadvantages of the TS instruction are as follows:

- Busy-waiting consumes a lot of the processor's time.
- Starvation may occur if a process leaves a critical section and there are more than one processes in waiting.
- It results in deadlocks.

If a low-priority process has the critical region and a high-priority process needs the higher priority process then it will obtain the processor to wait for the critical region.

CS Problem solution with test-and-set instruction:

```
boolean lock= false; /* lock is a shared
variable*/
/*Process Pi:*/
while (true) {
    while ( TestAndSet (&lock ) ); /* waits if
lock is 1*/
    /* enter CS */
    Critical Section code for Pi;
    lock = FALSE; /* Exit section code*/
    Remainder Section code for Pi
}
```

Swap Instruction

Swap instruction exchanges the contents of two memory variables atomically. The algorithm for swap is given below.

```
void Swap (boolean *a, boolean *b) /* exchange contents of
a and b atomically*/
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

CS Problem solution with swap Instruction

```
boolean lock= false; /* lock is a shared variable*/\n\n//Process Pi:\n/* each process defines a local boolean variable,\nkey*/\nboolean key=FALSE;\nwhile (true) {\n    key =TRUE;\n    while ( key==TRUE) swap(&lock, &key) ; /* busy\nwait*/\n    // enter CS\n    Critical Section code for Pi;\n    lock = FALSE; /* Exit section code*/\n    Remainder Section code for Pi\n}
```

Fetch-and-Increment Instruction

Fetch-and-increment is another hardware instruction that is executed atomically. The algorithm of the above instruction is as given below:

```
int FAI(int& val)\n{\n    return val++; // Performed atomically.\n}
```

5.2.3 Mutual Exclusion Algorithms

In a multiprogramming environment, while one process is executing the shared variable, all other processes wishing to do so at the same time moment should be kept waiting; while that process is done executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this manner, each process that is executing the shared variables prohibits all others from doing so concurrently. This is called **mutual exclusion**. But, mutual exclusion needs to be enforced only when processes access shared modifiable data, i.e., when processes are performing operations that do not conflict with one another they should be allowed to proceed simultaneously.

Mutual exclusion or mutex algorithms are used in concurrent programming to avoid the concurrent use of a common resource, such as a global variable, by pieces of computer code called critical sections. There are both software and hardware solutions for enforcing mutual exclusion. The software solutions for mutual exclusion are Dekker's algorithm, Peterson's algorithm, Lamport's algorithm, and Eisenberg and McGuire's algorithm.

NOTES

NOTES

Dekker's Algorithm

This algorithm is the first correct solution proposed for the two-process case. It was initially developed by Dekker. Dijkstra applied it to the critical section problem. In this algorithm, both the turn variable and the status flags are combined in a careful way. The entry protocol begins as follows:

The requesting process sets the flag and then checks the neighbor's flag.

If that flag is also set, the turn variable is used. There is no progress problem now because we know that the other process is either in its critical section or entry protocol.

If it is the turn of the requesting process, it waits for the flag of the other process to clear. No process will wait indefinitely with its flag set.

If the turn belongs to the other process the requesting process waits, but clears its flag before waiting to avoid blocking the other process. When the turn is given to the requesting process, it resets the flag and proceeds.

Initialization

```
typedef char boolean;  
...  
shared boolean flags[n-1];  
shared int turn;  
...  
turn = i ;  
...  
flags[i] = FREE;  
...  
flags[j] = FREE;  
...
```

Entry Protocol

```
/* claim the resource */  
flags[i] = BUSY;  
  
/* wait if the other process is using the resource */  
while (flags[j] == BUSY) {  
  
    /* if waiting for the resource, also wait our turn */  
    if (turn != i ) {  
  
        /* but release the resource while waiting */  
        flags[i] = FREE;
```

```
    while (turn != i) {  
    }  
    flags[i] = BUSY;  
}  
}
```

NOTES

Exit Protocol

```
/* pass the turn on and release the resource */  
turn = j;  
flags[i] = FREE;
```

Analysis: The mutual exclusion requirement is certain to be met. No process will go into its CS without setting its flag. Every process checks the other's flag after setting its own. If both are set, the turn variable is used to allow only one process to continue. The progress requirement is also certain. The turn variable is only considered when both processes are either using or trying to use the resource.

Deadlock is not possible. No process waits with its flag continuously set and one process always has the turn. The process with the turn will (ultimately) discover the other flag free and will proceed.

Finally, bounded waiting is assured. Suppose Process j exits its CS and re-enters immediately while Process i is waiting. Then the turn has been given to Process i , and the flag of Process i is set. Process j will clear its flag and wait, and Process i will proceed.

Peterson's Algorithm

This algorithm was developed by Peterson. Peterson developed and proved versions of his algorithm for both the 2-process case and the N -process case. In this algorithm, both the turn variable and the status flags are used, as in the case of Dekker's algorithm.

Initialization

```
typedef char boolean;  
...  
shared boolean flags[n-1];  
shared int turn;  
...  
turn = i;  
...  
flags[i] = FREE;  
...  
flags[j] = FREE;  
...
```

NOTES

Entry Protocol

```
/* claim the resource */
flags[i] = BUSY;

/* give away the turn */
turn = j ;
/* wait while the other process is using the resource
and has the turn */
while ((flags[j] == BUSY) && (turn != i )) {
}
```

Exit Protocol

```
/* release the resource */
flags[i] = FREE;
```

Analysis: The mutual exclusion requirement is assured. Suppose instead that both processes are in their critical section. Since only one can have the turn, the other must have reached the while test before the process with the turn set its flag. But after setting its flag, the other process had to give away the turn. The process at the while test has already changed the turn and will not change it again, contradicting our assumption. The progress requirement is assured. Again, the turn variable is only considered when both processes are using or trying to use the resource. Deadlock is not possible. If both processes are testing the while condition, one of them must have the turn. That process will proceed. Finally, bounded waiting is assured. When a process that has exited the CS reenters, it will give away the turn. If the other process is already waiting, it will be the next to proceed.

Eisenberg and McGuire's Algorithm

This is a correct solution for the N-process case developed by Eisenberg and McGuire. The turn variable and status flags are used as in Dekker's algorithm for the 2-process case. The flags now have three possible values: WAITING for a process in the entry protocol, waiting for the resource ACTIVE for a process in the critical section, using the resource; and IDLE for other cases.

Process priority is maintained in circular order beginning with the one holding the turn. Each process begins the entry protocol by scanning all processes from the one with the turn up to itself. These are the only processes that might have to go first if there is competition.

If the scan finds all processes idle, the process advances tentatively to the ACTIVE state. However, it is still possible that another process which started scanning later but belongs before us will also reach this state. We check one more time to be sure there are no active processes.

Initialization

```
shared enum states {IDLE, WAITING, ACTIVE} flags[n-1];
shared int turn;
int index; /* not shared! */
...
turn = 0;
...
for (index=0; index<n; index++) {
    flags[index] = IDLE;
}
```

Entry Protocol

```
repeat {

    /* announce that we need the resource */
    flags[i] = WAITING;

    /* scan processes from the one with the turn up to
     ourselves. */
    /* repeat if necessary until the scan finds all
     processes idle */
    index = turn;
    while (index != i) {
        if (flag[index] != IDLE) index = turn;
        else index = index+1 mod n;
    }

    /* now tentatively claim the resource */
    flags[i] = ACTIVE;

    /* find the first active process besides ourselves,
     if any */
    index = 0;
    while ((index < n) && ((index == i) || (flags[index]
    != ACTIVE))) {
        index = index+1;
    }

    /* if there were no other active processes, AND if we
     have the turn or else whoever has it is idle, then proceed.
     Otherwise, repeat the whole sequence. */
}
```

NOTES

NOTES

```
        } until ((index >= n) && ((turn == i) || (flags[turn]  
== IDLE)));
```

```
/* claim the turn and proceed */  
turn = i;
```

Exit Protocol

```
/* find a process which is not IDLE */  
/* (if there are no others, we will find ourselves) */  
index = turn+1 mod n;  
while (flags[index] = IDLE) {  
    index = index+1 mod n;  
}  
  
/* give the turn to someone that needs it or keep it */  
turn = index;  
  
/* finish the process */  
flag[i] = IDLE;
```

Peterson's N-Process Algorithm

This is an extension of Peterson's algorithm to the N -process case. Both status values and turn values are used. The status array is expanded to an integer value for each process which is used to track that process' progress in scanning the status of other processes. The turn value is also expanded to an integer array. Its values represent the relative ordering for each pair of processes. Peterson's 2-process solution is used repeatedly in $N-1$ levels to eliminate at least one process per level until only one remains.

Initialization

```
shared int flags[NUMPROCS];  
shared int turn[NUMPROCS - 1];  
int index;  
  
for (index = 0; index < (NUMPROCS); index++) {  
  
    flags[index] = -1  
}
```

```
for (index = 0; index < (NUMPROCS-1); index++) {  
    turn[index] = 0;  
}
```

*Mutual Exclusion
Problem and
Semaphores*

Entry Protocol

```
/* repeat for all partners */  
for (count = 0; count < (NUMPROCS-1); count++) {  
  
    flags[i] = count;  
    turn[count] = i;  
  
    "wait until (for all k != i, temp[k]<count) or  
(turn[count] != i)"  
  
}
```

Exit Protocol

```
/* finish the process */  
flags[i] = -1;
```

A Novel N-Process Solution: Lamport's Bakery Algorithm

The Bakery algorithm is a very different approach proposed by Leslie Lamport. It is based on the ‘take-a-number’ system used in bakeries and delicatessens. A process waiting to enter its critical section chooses a number. This number must be greater than all other numbers currently in use. There is a global shared array of current numbers for each process. The entering process checks all other processes sequentially and waits for each one which has a lower number. Ties are possible; these are resolved using process IDs.

Initialization

```
typedef char boolean;  
...  
shared boolean choosing[n];  
shared int num[n];  
...  
for (j=0; j < n; j++) {  
    num[j] = 0;  
}  
...
```

NOTES

NOTES

Entry Protocol

```
/* choose a number */
choosing[i] = TRUE;
num[i] = max(num[0], ..., num[n-1]) + 1;
choosing[i] = FALSE;

/* for all other processes */
for (j=0; j < n; j++) {

    /* wait if the process is currently choosing */
    while (choosing[j]) {}

    /* wait if the process has a number and comes ahead
    of us */
    if ((num[j] > 0) &&
        ((num[j] < num[i]) ||
        (num[j] == num[i]) && (j < i))) {
        while (num[j] > 0) {}
    }
}
```

Exit Protocol

```
/* clear our number */
num[i] = 0;
```

Check Your Progress

1. How a job is executed in a multiprocessing and multithreaded environment?
2. Explain about the producer-consumer problem.
3. State the requirements for solving the critical section problem.
4. What is the significance of exit section?
5. Why we use bakery algorithm?
6. What are the two important hardware solution used by critical section problem?
7. State the special hardware instruction to help in the solution of critical section problem.
8. Explain about the swap instruction.

NOTES

5.3 SEMAPHORES: DEFINITION

A semaphore is a variable or abstract data type used to control access to a common resource by multiple processes and avoid critical section problems in a concurrent system such as a multitasking operating system.

You have already learnt about critical section problems and to simplify the entry section code by using test-and-set and swap instructions. However, these solutions do not satisfy the bounded waiting requirement of the critical section problem solutions. Moreover, threads/processes are checking the conditions of lock in the busy wait state. This consumes costly processor cycle for doing no useful computation. To eliminate this busy wait problem, we use a lock variable to associate processes or threads wishing to execute a critical section code without the busy wait. Such a lock variable or data structure is called a semaphore. That is, semaphore variable is a synchronization tool that does not require busy waiting. It is an integer variable with two standard atomic operations, namely `wait()` and `signal()` (also called P and V operations) to modify it. These atomic operations are defined as follows:

Wait

```
wait( Semaphore S ) {  
    while (S<= 0) ; // no=op  
    S--;  
}
```

Signal

```
signal( Semaphore S ) {  
    S++;  
}
```

Depending up on the use for various purposes, we may use it as:

- Counting semaphore, where it can take any integer value.
- Binary semaphore, where it takes only two values.

Solution of CS problem with semaphores:

```
Semaphore S; // initialized to 1  
wait (S); // entry section  
Critical Section  
signal (S); // exit section  
Remainder Section
```

Busy Wait Implementation

In semaphore implementation, more than one process must not execute `wait()` and `signal()` operations on the same semaphore. That is, `wait()` and

NOTES

`signal()` codes are also critical section codes which should be executed as per the rules for executing a CS code.

We may use the busy waiting to enter the short CS code of `wait` and `signal` operations as these codes are small. Hence, there is no noticeable loss of efficiency due to a busy wait. Whereas, the applications, CS code is normally long and applications may spend a lot of time in their CS. Therefore, busy wait is not a good solution for entry to the application CS.

We associate a queue with each semaphore for suspending the processes which have failed to gain an entry to the CS code guarded by the semaphore. So, each entry in the queue can have two data items:

- Value (of type integer)
- Pointer to next record in the list of waiting processes

Each semaphore needs two operations to handle the processes as given below:

- Block: This operation places the process invoking the operation on the waiting queue of the semaphore.
- Wakeup: This operation takes out one process from the waiting queue and moves it to the ready queue.

The `wait()` and `signal()` primitives may be redefined as given below:

Implementation of `wait()` Function

```
wait (S) {
    value--;
    if (value < 0) {
        //place the process in the waiting queue;
        block();
    }
}
```

Implementation of `signal()` Function

```
Signal (S) {
    value++;
    if (value <= 0) {
        /* take a process P from the waiting queue of semaphore
        and move it to ready process queue */;
        wakeup (P);
    }
}
```

As we have discussed earlier, the trick to these implementations of `wait()` and `signal()` is that they themselves are critical sections. If they are both run at the same time in different threads, havoc may ensue.

To guard against this problem, as the `wait()` and `signal()` code is small, we can use a busy wait. Some operating systems provide atomic implementations for these functions. Interrupts can be turned off during execution to achieve atomic operations for operations involving short critical sections. This is possible because, when interrupts are disabled there will not be any task switching to enable multitasking. This technique is practicable because the code inside `wait()` and `signal()` is very short, and therefore will not tie the processor to one thread/process for too long.

*Mutual Exclusion
Problem and
Semaphores*

NOTES

Check Your Progress

9. Define the concept of mutual exclusion algorithm.
10. Write the code of entry and exit protocol for Dekker's algorithm.
11. Write the definition of semaphores.
12. State the two important operations to handle the process for semaphore.
13. Write the code of implementation of `wait()` and `signal()` function.

5.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. In a multiprocessing (a job executed as multiple processes) and a multithreaded environment, processes or threads have to cooperate in their execution while using shared resources like shared memory buffers. Cooperating processes or thread execution can affect or be affected by other processes/ threads. So there should be some way to synchronize their execution to use shared resources; otherwise, unexpected errors may occur.
2. There are two processes, one called producer process and the other called consumer process. Buffer-size is the number of buffers available for placing items (data). The producer process produces items (data) and places them into empty buffers. The consumer process takes the item from the occupied buffers and does some processing. The next empty buffer is indicated by an **In** pointer and the next item (buffer) to be consumed is indicated by the **Out** pointer. There is a variable **count** to indicate the number of full buffers at any time.
3. At a time only one process/thread is allowed to execute the critical section code. In other words, processes/threads must execute critical sections in a mutually exclusive fashion.

When no process/thread is executing in the critical section, only processes/ threads waiting for entry to the critical section should decide which one should enter the critical section next. This decision should be taken in such a way as to ensure progress in execution.

NOTES

Processes/threads must be allowed to enter the critical section in bounded time or without any starvation.

4. This code is the same for all processes wishing to use CS for sharing the same data structure. The code is executed when processes leave CS. This may be to change the flag to indicate that the process that entered CS has finished executing CS and is not in CS.

5. Bakery algorithm is used for multiple solution problem.

6. There are two common methods for hardware solutions:

- Disabling interrupts so that mutual exclusion can be achieved easily as there will not be any interrupt to pre-empt the currently executing process from CS.
- By using special hardware instructions, if provided by the hardware.

7. • Test-and-set instruction

• Swap instruction

• Fetch-and-increment instruction

8. Swap instruction exchanges the contents of two memory variables atomically. The algorithm for swap is given below.

```
void Swap (boolean *a, boolean *b) /* exchange contents of  
a and b atomically*/  
{  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

9. In a multiprogramming environment, while one process is executing the shared variable, all other processes wishing to do so at the same time moment should be kept waiting; while that process is done executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this manner, each process that is executing the shared variables prohibits all others from doing so concurrently. This is called mutual exclusion.

10. Entry Protocol

```
/* claim the resource */  
flags[i] = BUSY;  
  
/* wait if the other process is using the resource */  
while (flags[j] == BUSY) {  
  
    /* if waiting for the resource, also wait our turn */  
    if (turn != i) {
```

NOTES

```
/* but release the resource while waiting */
flags[i] = FREE;
while (turn != i) {
}
flags[i] = BUSY;
```

Exit Protocol

```
/* pass the turn on and release the resource */
turn = j;
flags[i] = FREE;
```

11. A semaphore is a variable or abstract data type used to control access to a common resource by multiple processes and avoid critical section problems in a concurrent system such as a multitasking operating system.
12. • Block: This operation places the process invoking the operation on the waiting queue of the semaphore.
 - Wakeup: This operation takes out one process from the waiting queue and moves it to the ready queue.
13. Implementation of wait() Function

```
wait (S) {
    value--;
    if (value < 0) {
        //place the process in the waiting queue;
        block();
    }
}
```

Implementation of signal() Function

```
Signal (S) {
    value++;
    if (value <= 0) {
        /* take a process P from the waiting queue of semaphore
        and move it to ready process queue */;
        wakeup(P);
    }
}
```

NOTES

5.5 SUMMARY

- Concurrent access to shared resources can result in unintended or incorrect actions in concurrent programming, so areas of the programme where the shared resource is accessed need to be secured in ways that prevent concurrent access. The critical segment or critical area is this safe portion.
- In a multiprocessing (a job executed as multiple processes) and a multithreaded environment, processes or threads have to cooperate in their execution while using shared resources like shared memory buffers.
- Cooperating processes or thread execution can affect or be affected by other processes/ threads.
- There are two processes, one called producer process and the other called consumer process.
- The producer process produces items (data) and places them into empty buffers. The consumer process takes the item from the occupied buffers and does some processing.
- Buffer-size is the number of buffers available for placing items (data).
- The producer process produces items (data) and places them into empty buffers. The consumer process takes the item from the occupied buffers and does some processing.
- The wrong value obtained is due to interleaved execution of increment and decrement operations on a shared variable. You would have got the correct value, if all the three instructions required for increment (or decrement) were executed without any interruption.
- Java language provides a keyword synchronized to solve this race condition. Such language-based support is not available in the lower-level languages.
- These are to be implemented mostly using basic programming primitives, even though some systems support hardware instructions for the same to build the solutions easily.
- At a time only one process/thread is allowed to execute the critical section code. In other words, processes/threads must execute critical sections in a mutually exclusive fashion.
- When no process/thread is executing in the critical section, only processes/ threads waiting for entry to the critical section should decide which one should enter the critical section next. This decision should be taken in such a way as to ensure progress in execution.
- Processes/threads must be allowed to enter the critical section in bounded time or without any starvation.
- There are two classes of solutions for CS problem, one using software and the other using special hardware instructions.

NOTES

- The progress requirement is violated. If both the flags are set, both the processes will be executing in entry section forever without winning entry to CS. This is deadlock situation.
- Disabling interrupts so that mutual exclusion can be achieved easily as there will not be any interrupt to pre-empt the currently executing process from CS.
- By using special hardware instructions, if provided by the hardware.
- Disabling interrupts for long periods of time leads to under utilization of resources.
- Text-and-Set or TS instruction provides a direct hardware support to mutual exclusion. It allows only one concurrent process to enter the critical section. It is able to make the programming task easier. It also improves the efficiency of the system.
- The critical-section problem can be solved simply in a uniprocessor environment if you are able to prevent the occurrence of interrupts during the modification of a shared variable.
- Many modern systems provide a special hardware support which helps in testing and modifying the content of a word or in swapping the contents of two words automatically.
- Each process has to execute the TS instruction in order to use the resource available with a control variable as an operand.
- This mechanism sets the global variable to 0, which indicates that the shared resource is available for being accessed. Each process has to execute the TS instruction in order to use the resource available with a control variable as an operand.
- Busy-waiting consumes a lot of the processor's time.
- Starvation may occur if a process leaves a critical section and there are more than one processes in waiting.
- Fetch-and-increment is another hardware instruction that is executed atomically.
- If a low-priority process has the critical region and a high-priority process needs the higher priority process then it will obtain the processor to wait for the critical region.
- In a multiprogramming environment, while one process is executing the shared variable, all other processes wishing to do so at the same time moment should be kept waiting; while that process is done executing the shared variable, one of the processes waiting to do so should be allowed to proceed.
- In this manner, each process that is executing the shared variables prohibits all others from doing so concurrently. This is called mutual exclusion.

NOTES

- Mutual exclusion or mutex algorithms are used in concurrent programming to avoid the concurrent use of a common resource, such as a global variable, by pieces of computer code called critical sections.
- The mutual exclusion requirement is certain to be met. No process will go into its CS without setting its flag. Every process checks the other's flag after setting its own.
- Deadlock is not possible. No process waits with its flag continuously set and one process always has the turn. The process with the turn will (ultimately) discover the other flag free and will proceed.
- Peterson developed and proved versions of his algorithm for both the 2 process case and the N-process case. In this algorithm, both the turn variable and the status flags are used, as in the case of Dekker's algorithm.
- The Bakery algorithm is a very different approach proposed by Leslie Lamport. It is based on the 'take-a-number' system used in bakeries and delicatessens. A process waiting to enter its critical section chooses a number. This number must be greater than all other numbers currently in use.
- CS code is normally long and applications may spend a lot of time in their CS. Therefore, busy wait is not a good solution for entry to the application CS.
- In semaphore implementation, more than one process must not execute wait() and signal() operations on the same semaphore. That is, wait() and signal() codes are also critical section codes which should be executed as per the rules for executing a CS code.

5.6 KEY WORDS

- **Critical Section (CS):** Critical section means the code that is used to access the shared data structures by more than one process or thread normally in cooperated multitasking or multiprocessing or multithreaded computation.
- **Interrupt:** An interrupt is a response by the processor to an event that needs attention from the software. ... Interrupts are also commonly used to implement computer multitasking, especially in real-time computing. Systems that use interrupts in these ways are said to be interrupt-driven.
- **Progress:** When no process/thread is executing in the critical section, only processes/threads waiting for entry to the critical section should decide which one should enter the critical section next. This decision should be taken in such a way as to ensure progress in execution.
- **Mutual Exclusion:** Mutual exclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions. This problem

(called a race condition) can be avoided by using the requirement of mutual exclusion to ensure that simultaneous updates to the same part of the list cannot occur.

- **Test-and-Set Instruction:** Test-and-Set or TS instruction provides a direct hardware support to mutual exclusion. It allows only one concurrent process to enter the critical section. It is able to make the programming task easier. It also improves the efficiency of the system.
- **Fetch-and-Increment Instruction:** Fetch-and-increment is another hardware instruction that is executed atomically.
- **Semaphores:** A semaphore is a vector or abstract data type used in computer science to monitor several processes of access to a common resource and prevent critical section problems in a concurrent system such as a multitasking operating system.

*Mutual Exclusion
Problem and
Semaphores*

NOTES

5.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What do you meant by critical section problem?
2. What are the solution of critical section problem?
3. Why producer-consumer process is used?
4. Write down the other name of producer-consumer classical problem.
5. Which algorithm is used in multiple process solution?
6. What is the drawback of disabling interrupts?
7. Explain advantage and disadvantage for special machine instruction.
8. Elucidate on the different type of mutual exclusion algorithm.
9. State the algorithms which are used in software solution for mutual exclusion.
10. Write the syntax for wait () and signal () operations in semaphores.
11. What is busy waiting? How is semaphore used to overcome the busy waiting problem?

Long-Answer Questions

1. How we define the critical-section problem? Explain all the requirements that a solution to a critical-section problem must meet.
2. Write down the producer-consumer process in detail.
3. Give the pros and cons of the two process solution for solving critical section problem.

NOTES

4. Write a code to eliminate the deadlock situation.
5. State a Test-and-Set instruction. Also write the algorithm that uses the Test-and-Set instruction to solve the critical-section problem and meets all the requirements of the solution for such a problem.
6. What are the advantages and disadvantages of test-set instruction?
7. Elaborate briefly on the Dekker's algorithm.
8. Describe the Peterson's algorithm.
9. Briefly illustrate the Eisenberg and McGuire's algorithm.
10. Explain the bakery algorithm to solve a critical-section problem.
11. Discuss the use of semaphores in developing a solution to a bounded-buffer problem.
12. Write short notes on the following:
 - (a) Entry and exit section
 - (b) Swap instruction
 - (c) Semaphore

5.8 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.
- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.
- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.
- Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.
- Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.
- Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

UNIT 6 CONCURRENT PROGRAMMING

Concurrent Programming

NOTES

Structure

- 6.0 Introduction
 - 6.1 Objectives
 - 6.2 Sequential and Concurrent Process
 - 6.3 Precedence Graph
 - 6.4 Bernstein's Condition
 - 6.5 Time Dependency
 - 6.6 Monitors
 - 6.7 Answers to Check Your Progress Questions
 - 6.8 Summary
 - 6.9 Key Words
 - 6.10 Self Assessment Questions and Exercises
 - 6.11 Further Readings
-

6.0 INTRODUCTION

Concurrent processing is a computing method where several processors execute instructions simultaneously for better performance. Concurrent means, which occurs when something else happens. The tasks are broken down into sub-types, which are then allocated to different processors to execute concurrently, instead of sequentially, since one processor will have to perform them.

Concurrent computing is a type of computing in which multiple calculations are conducted simultaneously, rather than sequentially, over overlapping time periods, with one completed before the next begins.

The concept of sequential process was proposed in 1978 as the basis of interaction in concurrent programming. It synchronizes and communicates by means of input and output processes. It establishes rendezvous. A rendezvous is established if one process is ready to execute an input statement and the second process is ready to execute the corresponding output statement. If either process is not ready then the other process is forced to wait. The specification of process synchronization condition is straight-forward in which input statements are predefined with synchronization conditions.

A number of different methods can be used to implement concurrent programs, such as implementing each computational process as an operating system process, or implementing the computational processes as a set of threads within a single operating system process.

NOTES

A monitor is a programming language construct which is also used to provide mutually exclusive access to critical sections. The programmer defines monitor type which consists of declaration of shared data (or variables), procedures or functions that access these variables and initialization code.

The Dining Philosopher Problem-The Dining Philosopher Problem notes that K philosophers sit between any pair of philosophers around a circular table with one chopstick. Each philosopher has one chopstick between them. A philosopher can eat if the two chopsticks adjacent to him can be picked up.

This unit will discuss concurrent programming in detail, highlighting the concepts of precedence graphs, Bernstein's condition, time-dependency and monitors.

6.1 OBJECTIVES

After going through this unit, you will be able to:

- Describe the significance of sequential and concurrent processes
- Discuss the various features of precedence graphs
- Describe the Bernstein's condition
- Analyse the factors that determine time dependency
- Understand the role and function of monitors
- Know about the dining philosophers problem

6.2 SEQUENTIAL AND CONCURRENT PROCESS

Process is not the same as program; instead, it is an active entity for the program. A program is an algorithm expressed in some suitable notation. A process includes the current value of the Program Counter (PC), contents of the processor registers and value of the variables. The Process Stacks (SP) contains temporary data, such as subroutine parameter, return address and temporary variables. It includes a data section that contains global variables. A process is the unit of work in a system. It goes through a series of discrete process states.

Sequential process: The concept of sequential process was proposed in 1978 as the basis of interaction in concurrent programming. It synchronizes and communicates by means of input and output processes. It establishes rendezvous. A rendezvous is established if one process is ready to execute an input statement and the second process is ready to execute the corresponding output statement. If either process is not ready then the other process is forced to wait. The specification of process synchronization condition is straight-forward in which input statements are predefined with synchronization conditions.

NOTES

Concurrent process: The concurrent process refers to an instance of the program written for operating system. These processes are processed by the kernel. In this technique, scheduler assigns time slices. The slice could be 1/100 or 1/1000 seconds, concurrently working at process level. The various attributes, such as unique Process ID (PID), Process ID of Parent (PPID), User ID of process owner (UID), execution priority (priority), state of the process, such as sleeping, running are involved in concurrent processes. The concurrent processes are called by the system calls that perform some tasks, such as low-level I/O, interrupted control I/O to terminals.

Since UNIX OS is assembled with C language, therefore, various functions are used to implement the concurrent process. A system call is used to associate the header files. Two processes are said to be concurrent, if their execution overlaps in time and the execution of the second process starts before the first process is completed. The concurrent processes generally interact through the mechanisms known as shared variables and message passing. If the application is set with various concurrent processes then the sequential program is determined.

main program	
let x = ref 1;;	
process P	process Q
x := !x + 1;;	x := !x * 2;;

Fig. 6.1 Program with two processes P and Q

Figure 6.1 shows two processes, process P and process Q. At the end of the execution of P and Q, the reference x can point to 2,3 or 4, depending on the order of execution of each process. This concept can be explained with the help of an example (Figure 6.2). The keyword ‘send’ sends a value to the destination and receives a value from a process.

process P	process Q
let x = ref 1;;	let y = ref 1;;
send(Q,!x);	y := !y + 3;
x := !x * 2;	y := !y + receive(P);
send(Q,!x);	send(P,!y);
x := !x + receive(Q);	y := !y + receive(P);

Fig. 6.2 Use of Send Keyword in Concurrent Processes P and Q

A PID variable is declared as the form class in which various routines are involved. When the memory state depends on the execution of each parallel process, an application may fail to terminate on a particular execution and terminate on another. To provide some control over the execution, the processes must be synchronized. The interaction of processes using distinct memory areas, but communicating with each other depends on the type of communication. The

NOTES

following example uses two communication primitives; ‘Send’ which sends a value, showing the destination and ‘Receive’ which receives a value from a process. Let P and Q be two communicating processes. In case of a transient communication, process Q can miss the messages of P. For an asynchronous communication, the communication channel stores the different values that have been transmitted. Only reception is blocked. Process P can be waiting for Q, even if the latter has not yet read the two messages from P.

Figure 6.3 shows how concurrent programming maintains the sequence between programs and parallel programs.

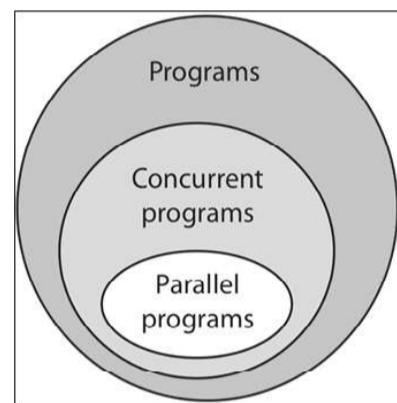


Fig. 6.3 Domain of Concurrent Programs

Let us take an example, in which the two units of execution are started at the same time. If they run in parallel, then over any time interval during their execution, they may be executing at the same time. If they run concurrently, they may execute in parallel, or they may be sequentialized where only one makes progress at any given point of time and their execution is interleaved. Multitasking operating systems implement this concept. Figure 6.4 shows how three programs execute concurrently. It shows that the programs can be concurrent with non-parallel as well as parallel conditions.

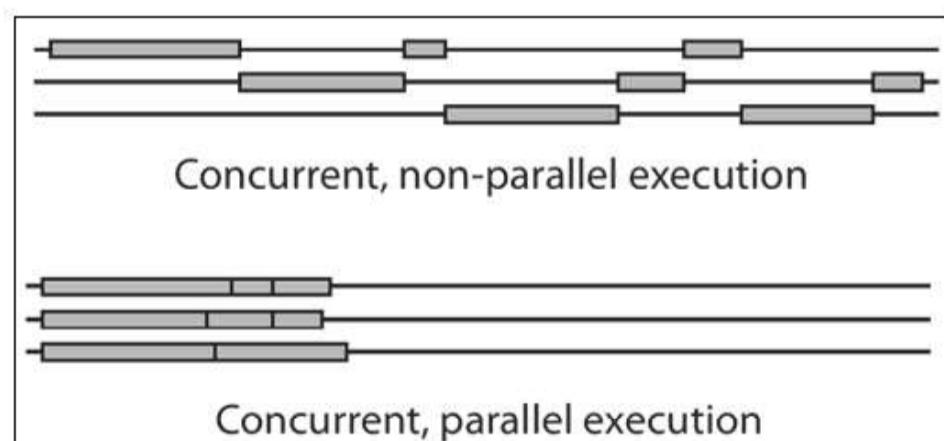


Fig. 6.4 Concurrent Execution of Programs

NOTES

Both conditions can be analysed in the following way so that:

- (i) You can formalize this via Bernstein's conditions.
- (ii) Assign a subprogram P.
- (iii) Let IN (P) represent the set of memory locations that includes the registers or variables that P uses as input by reading from them.
- (iv) Let OUT (P) represent a similar set of locations that P uses as output by writing to them.
- (v) You can use these sets to determine if two subprograms P1 and P2 are dependent, and therefore whether or not they can execute in parallel.
- (vi) Given two subprograms P1 and P2, their execution in parallel is equivalent to their execution in sequence if the following conditions hold.

6.3 PRECEDENCE GRAPH

A precedence graph refers to a acyclic and directed graph that carries nodes. These nodes are, in fact, software tasks, recognized by the operating system. The statements, which are written in a set of statements, can be executed concurrently. In order to observe the process functionality, the processes are restricted to determine the interaction based shared resources and on sending and receiving the messages. The precedence graph expresses only the precedence relations among the sequential processes. The synchronization process is expressed by the precedence graph. Let us take an example of a sequential program:

```
a := x + y; /* Statement S1 */  
b := z + 1; /* Statement S2 */  
c := a - b; /* Statement S3 */  
w := c + 1; /* Statement S4 */  
d := a + e; /* Statement S5 */  
w := w * d; /* Statement S6 */
```

The Bernstein conditions are used to derive the precedence graph from the above sequential program. The nodes are represented by S1, S2, S3, S4, S5 and S6. The successors of S1 are S3 and S5. The S6 is the successor of S5. Two processors are used in the precedence graph. One processor executes the statement in the order of S1, S5, S6, whereas other processor executes S2, S3, S4. The characteristics of a precedence graph are as follows:

- (i) It is that is acyclic and can not have loops.
- (ii) It has only one dependency and hence cannot have cyclic computations.

NOTES

Figure 6.5 shows a precedence graph,

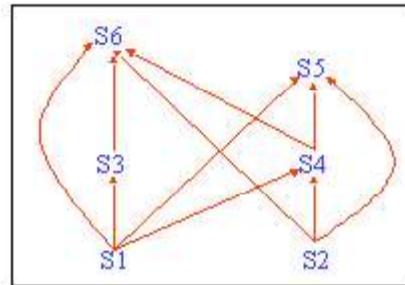


Fig. 6.5 A Precedence Graph

In the above Figure 6.5, S1 represents $a = 0$, S2 represents $b = 1$, $c := a + 1$, $d = b + a$, $e := d + 1$ and $e := c + d$ shows in the precedence graph. S3 depends on S1 but it does not depend on S2. Independent statements are executed in parallel to speed up the program.

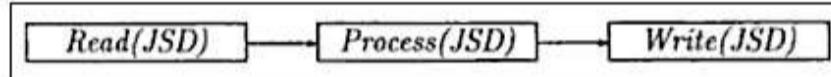


Fig. 6.6 Precedence Graph of BOS Processing a JDS

Figure 6.6 illustrates that a Batch Operating System (BOS) is used to operate a job file where all Job Data Structures (JDS) are processed. Using the precedence graph of a BOS, the physical processors execute the sequential operations Read, Process and Write. The sequential processes are described as the parallel processes. The parallel process $PP = (P_1, P_2, \dots, P_n)$ is functional if while executing PP with the same input data X , the process is able to generate the output data Y . The functional processes are enforced by mutual exclusion and variables are shared.

The precedence graph of P is a directed acyclic graph. It is denoted by $G(PP)_1$, whose nodes are denoted by the processes P_1, P_2, \dots, P_n . The precedence graph illustrates the parallel version of the (BOS).

Various processors are used to execute the precedence graphs in the following ways:

1. If the precedence graph is executed using a single processor, it is to be checked that the graph is not exploited and parallelism is inherent.
2. Many processors can be used according to the nodes defined in the graph. A processor completes the activity to represent the node. It later becomes idle and cannot be used efficiently.
3. The maximum number of processors is used efficiently if the cardinality of the set of nodes do not depend any two nodes in the set.

NOTES

Fork and Join in precedence graph: Dennis and VanHorne introduced the Fork and Join concepts in the year 1966. If the statement ‘Fork (label)’ is executed by the thread of control, concurrently the very next thread of control starts and in this way, the two threads are executed concurrently. If the statement ‘Join (count)’ is executed the value of count is decremented. The count parameter is taken as in data type. If the returned value is positive, the executing thread is terminated. The Fork and Join are suitable for precedence graphs because they can be expressed using Fork and Join the without loss of concurrency. Any of the precedence graphs could be executed as a sequential program, but at a loss in concurrency.

The CoBegin Coend construct is also known as the ParBegin ParEnd construct. This concept was introduced by Dijkstra in 1968. It is much easier to use than Fork and Join. Let S1, S2, .. Sn statements use the construct. The construct then starts producing the following statements:

```
cobegin
S1;
S2;
..
Sn;
coend;
```

When CoBegin is executed, S1, S2, ..., Sn are executed concurrently each in a separate thread. When all these statements are terminated, only then the CoEnd is executed and the construct is terminated. There are precedence graphs that can be represented using CoBegin CoEnd only at the cost of some loss in concurrency. Figure 6.7 shows the Fork-Join model that is collectively produced in parallel region.

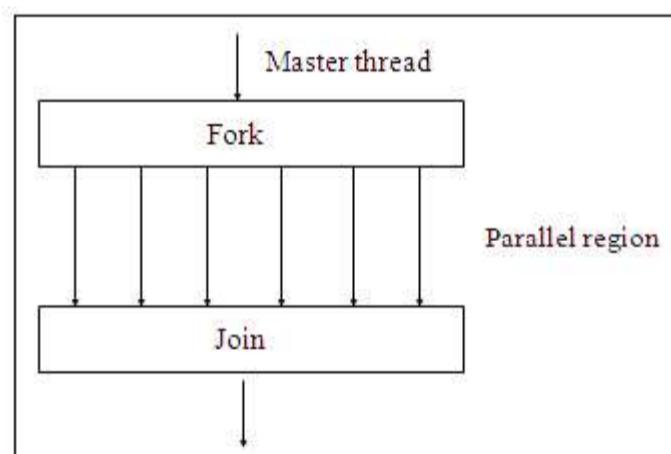


Fig. 6.7 Fork – Join Model

NOTES

Check Your Progress

1. Explain about the concept of sequential process.
2. Define the term of concurrent process.
3. How concurrent programming maintains the sequence between programs and parallel programs?
4. What are the purpose of precedence graph?
5. Which condition is used for precedence graph?
6. How processors are used to execute precedence graph?
7. Write the definition of fork and join for precedence graph.

6.4 BERNSTEIN'S CONDITION

The Bernstein's condition is applied if the input sets of two tasks, which are independent of each other's output sets execute in parallel. The following three conditions are taken as constraints if shared memory is used to enforce precedence among the processes:

1. If process P_i writes to a memory cell M_i , then no process P_j can read the cell M_i .
2. If process P_i reads from a memory cell M_i , then no process P_j can write to the cell M_i .
3. If process P_i writes to a memory cell M_i , then no process P_j can write to the cell M_i .

The application of semaphores, for example, can provide a means of forcing processes to take turns in reading and writing to shared memory. Subtasks in a parallel program are called threads. Some parallel computer architectures use smaller, lightweight versions of threads known as fibers, while others use bigger versions known as processes. However, 'threads' is generally accepted as a generic term for subtasks. They are needed to update some variable that is shared between them. The instructions between the two programs may be interleaved in any order. Let take a condition, if sequential process P is determinate a given input X moves through identical instruction sequences to arrive at its result; for example:

```
read x
y = 2*x;
z = y * y;
print z;
```

For example, consider the following program:

Thread A Thread B

```
1A: Read variable V 1B: Read variable V  
2A: Add 1 to variable V 2B: Add 1 to variable V  
3A: Write back to variable V 3B: Write back to variable V
```

If instruction 1B is executed between 1A and 3A, or if instruction 1A is executed between 1B and 3B, the program will produce incorrect data. This is known as a race condition. The programmer must use a lock to provide mutual exclusion. A lock is a programming language construct that allows one thread to take control of a variable and prevent other threads from reading or writing it, until that variable is unlocked. The thread holding the lock is free to execute its critical section and unlock the data when it is finished. Therefore, in order to guarantee correct program execution, the above program can be rewritten to use locks as follows:

Thread A Thread B

```
1A: Lock variable V 1B: Lock variable V  
2A: Read variable V 2B: Read variable V  
3A: Add 1 to variable V 3B: Add 1 to variable V  
4A: Write back to variable V 4B: Write back to variable V  
5A: Unlock variable V 5B: Unlock variable V
```

One thread is used to lock variable V, while the other thread will be locked out unable to proceed until V is unlocked again. This guarantees correct execution of the program. Locks, while necessary to ensure correct program execution can slow down a program. Locking multiple variables using non-atomic locks introduces the possibility of program deadlock. An atomic lock is a lock that locks multiple variables at a time. If it cannot lock all of them, it does not lock any of them. If two threads each need to lock the same two variables using non-atomic locks, it is possible that one thread will lock one of them and the second thread will lock the second variable. In this case, neither thread can complete, and deadlock results. Many parallel programs require their subtasks to act in synchrony. This requires the use of a barrier. Barriers are typically implemented using a software lock. One class of algorithms, known as lock-free and wait-free algorithms, avoid altogether the use of locks and barriers. However, this approach is generally difficult to implement and requires correctly designed data structures. A task is split up into more and more threads, and those threads spend an ever-increasing portion of their time communicating with each other. Eventually, the overhead from communication dominates the time spent solving the problem and causes further parallelization, that is, splitting the workload over even more threads. This increases the amount of time required to finish the task. This is known as parallel slowdown. Applications are frequently classified according to how often their subtasks need to synchronize or communicate with each other. An application exhibits fine-grained

NOTES

NOTES

parallelism if its subtasks must communicate many times per second; it exhibits coarse-grained parallelism if they do not communicate many times per second. It is parallel if they rarely or never have to communicate. The embarrassingly parallel applications are considered the easiest to parallelize.

6.5 TIME DEPENDENCY

If each queue has absolute priority over lower-priority queues then no process in the queue could run unless the queue for the highest-priority processes is all empty. For example, no process in the batch queue could run unless the queues for system processes, interactive processes and interactive editing processes are all empty. If there is a time slice between the queues then each queue gets a certain amount of CPU times, which it can then schedule among the processes in its queue, for example, the following two conditions determine the time dependency:

- (i) 80 per cent of the CPU time to foreground queue using RR.
- (ii) 20 per cent of the CPU time to background queue using FCFS.

Since processes do not move between queues so, this policy has the advantage of low scheduling overhead, but it is inflexible. The compile-time dependency exists if a component of program has to be recompiled if one of the following parts in another component changes:

- (i) Base class that includes private inheritance
- (ii) Inline functions and inline member functions
- (iii) Private and protected members
- (iv) Compiler generated member functions, such as the assignment operator
- (v) Include directives and default arguments
- (vi) Enumerations

The availability of resources, for example number of CPUs and the resource sharing policy of the run-time system are also supported by the process scheduling algorithm. The time-dependency method involves two steps; the first step determines if a timing property can be proved from the system specification and identify implementation-dependent assumptions, and the second step, determines if the system specifications and other required implementation dependencies can be enforced by control structures that meet the required timing constraints. The advantage of this method is that implementation dependencies are explicitly identified. It is important to isolate them in a proof of correctness so that if the execution environment changes, the impact of the change can be readily identified and understood. Both steps are accomplished by making use of the logic Real-Time Logic (RTL). The time dependency basically depends on the following three factors:

- (i) **Resource Model:** It describes the system resources available to the applications.

NOTES

- (ii) **Workload Model:** It describes the applications supported by the system.
- (iii) **Scheduling algorithm:** It defines how the application system uses the resources at all times. This is where real-time operating systems have a key role to play.

6.6 MONITORS

A monitor is a programming language construct which is also used to provide mutually exclusive access to critical sections. The programmer defines monitor type which consists of declaration of shared data (or variables), procedures or functions that access these variables and initialization code. The general syntax of declaring a monitor type is as follows:

```
monitor <monitor-name>
{
    //shared data (or variable) declarations
    data type <variable-name>;
    ...
    //function (or procedure) declarations

    return_type <function-name>(parameters)
    {
        //body of function
    }
    .
    .
    .
    monitor-name()
    {
        //initialization code
    }
}
```

The variables defined inside a monitor can only be accessed by the functions defined within the monitor, and it is not feasible for any process to access these variables. Thus, if any process has to access these variables, it is only possible through the execution of the functions defined inside the monitor. Further, the monitor construct checks that only one process may be executing within the monitor at a given moment. But if a process is executing within the monitor, then other requesting processes are blocked and placed on an entry queue.

NOTES

Though, monitor construct ensures mutual exclusion for processes, but sometimes programmer may find them insufficient to represent some synchronization schemes. For such situations, programmer needs to define his own synchronization mechanisms. He can define his own mechanisms by defining variables of condition type on which only two operations can be invoked: wait and signal. Suppose, programmer defines a variable C of condition type, then execution of the operation C.wait() by a process, say P_i, suspends the execution of P_i, and places it on a queue associated with the condition variable C. On the other hand, the execution of the operation C.signal() by a process, say P_i, resumes the execution of exactly one suspended process P_j, if any. It means that the execution of the signal operation by P_i allows other suspended process P_j to execute within the monitor. However, only one process is allowed to execute within the monitor at one time. Thus, monitor construct prevents P_j from resuming until P_i is executing in the monitor. There are following possibilities to handle this situation.

- The process P_i must be suspended to allow P_j to resume and wait until P_j leaves the monitor.
- The process P_j must remain suspended until P_i leaves the monitor.
- The process P_i must execute the signal operation as its last statement in the monitor so that P_j can resume immediately.

Now, write a solution to the diningPh problem which avoids deadlock, we are in a situation to use the monitor to develop a deadlock-free solution to dining philosophers problem. The following monitor controls the distribution of chopsticks to philosophers.

```
monitor diningPhilosophers
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];

    void getChopsticks(int i)
    {
        int left, right;
        state[i] = hungry;
        left = (i+4)%5;
        right = (i+1)%5;
        if((state[left]==eating) || (state[right]==eating))
            self[i].wait();
        else
            state[i] = eating;
    }

    void putDownChopsticks(int i)
    {
        int left, right;
```

NOTES

```
state[i] = thinking;
left = (i+4)%5;
right = (i+1)%5;
    verifyAndAllow(left);
    verifyAndAllow(right);
}

void verifyAndAllow(int i)
{
int left, right;
left = (i+4)%5;
right = (i+1)%5;
if(state[i]==hungry)
{
if((state[left]!=eating) && (state[right]!=eating))
{
    state[i] = eating;
self[i].signal();
}
}

void initial()
{
int i;
for(i=0; i<5; i++)
    state[i] = thinking;
}
```

Each philosopher that feels hungry must invoke the `getChopsticks()` operation before start eating and after eating is finished, he must invoke `putDownChopsticks()` operations and then may start thinking. Thus, the general structure for the code segment philosopher `i` is as follows:

```
...
diningPhilosophers.getChopsticks(i);
eating
diningPhilosophers.putDownChopsticks(i);
...
```

NOTES

The `getChopsticks()` operation changes the state of philosopher process from thinking to hungry and then verifies whether philosopher on his left or right is in eating state. If either philosopher is in eating state, then the philosopher process is suspended and its state remains hungry. Otherwise, the state of philosopher process is changed to eating.

After eating is finished, each philosopher invokes `putDownChopsticks()` operation before start thinking. This operation changes the state of philosopher process to thinking and then invoke `verifyAndAllow()` operation for philosophers on his left and right side (one by one). The `verifyAndAllow()` operation verifies whether the philosopher feels hungry, and if so then allows him to eat in case philosophers on his left and right side are not eating.

Check Your Progress

8. What do you understand by Bernstein's conditions?
9. State the two conditions determine the time dependency.
10. Give the definition of monitor.
11. Monitor construct ensures that _____ may be executing within the monitor at a time.
12. Programmer can define his own synchronization mechanisms. (True or False)
13. The dining philosophers problem will occur in which case?

6.7 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. The concept of sequential process was proposed in 1978 as the basis of interaction in concurrent programming. It synchronizes and communicates by means of input and output processes. It establishes rendezvous. A rendezvous is established if one process is ready to execute an input statement and the second process is ready to execute the corresponding output statement. If either process is not ready then the other process is forced to wait. The specification of process synchronization condition is straightforward in which input statements are predefined with synchronization conditions.
2. The concurrent process refers to an instance of the program written for operating system. These processes are processed by the kernel. In this technique, scheduler assigns time slices. The slice could be 1/100 or 1/1000 seconds, concurrently working at process level. The various attributes, such as unique Process ID (PID), Process ID of Parent (PPID), User ID of process owner (UID), execution priority (priority), state of the process,

such as sleeping, running are involved in concurrent processes. The concurrent processes are called by the system calls that perform some tasks, such as low-level I/O, interrupted control I/O to terminals.

Concurrent Programming

3. If they run in parallel, then over any time interval during their execution, they may be executing at the same time. If they run concurrently, they may execute in parallel, or they may be sequentialized where only one makes progress at any given point of time and their execution is interleaved. Multitasking operating systems implement this concept.
4. A precedence graph refers to a acyclic and directed graph that carries nodes. These nodes are, in fact, software tasks, recognized by the operating system. The statements, which are written in a set of statements, can be executed concurrently. In order to observe the process functionality, the processes are restricted to determine the interaction based shared resources and on sending and receiving the messages. The precedence graph expresses only the precedence relations among the sequential processes. The synchronization process is expressed by the precedence graph.
5. The Bernstein conditions are used to derive the precedence graph from the above sequential program.
6.
 1. If the precedence graph is executed using a single processor, it is to be checked that the graph is not exploited and parallelism is inherent.
 2. Many processors can be used according to the nodes defined in the graph. A processor completes the activity to represent the node. It later becomes idle and cannot be used efficiently.
 3. The maximum number of processors is used efficiently if the cardinality of the set of nodes do not depend any two nodes in the set.
7. Dennis and VanHorne introduced the Fork and Join concepts in the year 1966. If the statement ‘Fork (label)’ is executed by the thread of control, concurrently the very next thread of control starts and in this way, the two threads are executed concurrently. If the statement ‘Join (count)’ is executed the value of count is decremented. The count parameter is taken as in data type. If the returned value is positive, the executing thread is terminated. The Fork and Join are suitable for precedence graphs because they can be expressed using Fork and Join the without loss of concurrency. Any of the precedence graphs could be executed as a sequential program, but at a loss in concurrency.
8. The Bernstein’s condition is applied if the input sets of two tasks, which are independent of each other’s output sets execute in parallel. The following three conditions are taken as constraints if shared memory is used to enforce precedence among the processes:
 1. If process P_i writes to a memory cell M_i , then no process P_j can read the cell M_i .

NOTES

NOTES

2. If process P_i reads from a memory cell M_p , then no process P_j can write to the cell M_i .
3. If process P_i writes to a memory cell M_p , then no process P_j can write to the cell M_i .
9. (i) 80 per cent of the CPU time to foreground queue using RR.
(ii) 20 per cent of the CPU time to background queue using FCFS.
10. A monitor is a programming language construct which is also used to provide mutually exclusive access to critical sections. The programmer defines monitor type which consists of declaration of shared data (or variables), procedures or functions that access these variables and initialization code.
11. Only one process.
12. True
13. In case of 5 philosophers and 5 chopsticks.

6.8 SUMMARY

- A process is the unit of work in a system. Processes could be sequential or concurrent.
- Process is not the same as program; instead, it is an active entity for the program. Program is an algorithm expressed in some suitable notation.
- Process includes the current value of the program counter (PC), contents of the processor registers and value of the variables.
- The process stacks (SP) contains temporary data, such as subroutine parameter, return address and temporary variables. It includes a data section that contains global variables.
- The concept of sequential process was proposed in 1978 as the basis of interaction in concurrent programming. It synchronizes and communicates by means of input and output processes. It establishes rendezvous.
- The concurrent process refers to an instance of the program written for operating system. These processes are processed by the kernel. In this technique, scheduler assigns time slices.
- UNIX OS is assembled with C language, therefore, various functions are used to implement the concurrent process
- A PID variable is declared as the form class in which various routines are involved. When the memory state depends on the execution of each parallel process, an application may fail to terminate on a particular execution and terminate on another.
- To provide some control over the execution, the processes must be synchronized.

NOTES

- The process stacks (SP) contains temporary data, such as subroutine parameter, return address and temporary variables. It also includes a data section that contains global variables.
- The interaction of processes using distinct memory areas, but communicating with each other depends on the type of communication.
- Precedence graphs are acyclic and directed graphs that carry nodes. These nodes are, in fact, software tasks, which are recognized by the operating system.
- In order to observe the process functionality, the processes are restricted to determine the interaction based shared resources and on sending and receiving the messages.
- The precedence graph expresses only the precedence relations among the sequential processes. The synchronization process is expressed by the precedence graph.
- The Bernstein conditions are used to derive the precedence graph from the above sequential program.
- Bernstein's conditions do not allow memory to be shared between different processes. For that, some means of enforcing an ordering between accesses is carried out by the OS.
- Batch Operating System (BOS) is used to operate a job file where all Job Data Structures (JDS) are processed. Using the precedence graph of a BOS, the physical processors execute the sequential operations Read, Process and Write.
- The sequential processes are described as the parallel processes.
- If the precedence graph is executed using a single processor, it is to be checked that the graph is not exploited and parallelism is inherent.
- Many processors can be used according to the nodes defined in the graph. A processor completes the activity to represent the node. It later becomes idle and cannot be used efficiently.
- The maximum number of processors is used efficiently if the cardinality of the set of nodes do not depend any two nodes in the set.
- Dennis and VanHorne introduced the Fork and Join concepts in the year 1966.
- If the statement ‘Fork (label)’ is executed by the thread of control, concurrently the very next thread of control starts and in this way, the two threads are executed concurrently.
- If the statement ‘Join (count)’ is executed the value of count is decremented. The count parameter is taken as in data type. If the returned value is positive, the executing thread is terminated.

NOTES

- The Fork and Join are suitable for precedence graphs because they can be expressed using Fork and Join without loss of concurrency.
- The CoBebin Coend construct is also known as the ParBegin ParEnd construct. This concept was introduced by Dijkstra in 1968.
- The Bernstein's condition is applied if the input sets of two tasks, which are independent of each other's output sets execute in parallel.
- If each queue has absolute priority over lower-priority queues then no process in the queue could run unless the queue for the highest-priority processes is all empty.
- A monitor is a programming language construct which is also used to provide mutually exclusive access to critical sections.
- Monitor construct ensures that only one process may be executing within the monitor at a time.
- Programmer can define his own synchronization mechanisms by defining variables of condition type on which only two operations can be invoked: wait and signal.
- Scheduling algorithm defines how the application system uses the resources at all times. This is where real-time operating systems have a key role to play.
- Monitor is used to develop a deadlock-free solution to dining philosopher's problem.
- Monitor construct ensures mutual exclusion for processes, but sometimes programmer may find them insufficient to represent some synchronization schemes.
- The getChopsticks() operation changes the state of philosopher process from thinking to hungry and then verifies whether philosopher on his left or right is in eating state.
- Monitor construct ensures mutual exclusion for processes, but sometimes programmer may find them insufficient to represent some synchronization schemes.

6.9 KEY WORDS

- **Sequential process:** Can only appear inside of a Process Block.
- **Concurrent process:** Can appear outside of a Process Block.
- **Precedence graph:** A precedence graph refers to an acyclic and directed graph that carries nodes. These nodes are, in fact, software tasks, recognized by the operating system. The statements, which are written in a set of statements, can be executed concurrently.

NOTES

- **Bernstein condition:** The Bernstein's condition is applied if the input sets of two tasks, which are independent of each other's output sets execute in parallel.
- **Fork:** If the statement 'Fork (label)' is executed by the thread of control, concurrently the very next thread of control starts and in this way, the two threads are executed concurrently.
- **Join:** If the statement 'Join (count)' is executed the value of count is decremented.
- **Monitors:** A monitor is a programming language construct which is also used to provide mutually exclusive access to critical sections. The programmer defines monitor type which consists of declaration of shared data (or variables), procedures or functions that access these variables and initialization code.

6.10 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. How does concurrent programming maintains the sequence between programs and parallel programs?
2. Explain the term of precedence graph with the help of example.
3. Give the characteristics of precedence graph.
4. What do you understand by the 'Fork and Join in precedence'?
5. Define the concept of Bernstein's condition in the context of concurrent Programming.
6. Write the definition of time dependency.
7. What is monitor for concurrent programming?
8. Write a solution to the dining philosophers problem which avoid deadlock.

Long-Answer Questions

1. What are sequential and concurrent programming process and its type with the help of a diagram?
2. State the ways to analyse the concurrent, on-parallel execution and concurrent, parallel execution.
3. Explain precedence graph with the help of example and diagrams.
4. Discuss briefly about the three conditions required for the Bernstein's condition.

NOTES

5. Elaborate on the term of time-dependency and also describe the three factors of time dependency.
6. Illustrate the general syntax for monitor.
7. What are the possibilities of handling the situation occurred by monitors?
8. Consider the dining philosophers problem. If we add a 6th chopstick to the centre of the table, have we cured the deadlock problem? If yes, what condition have we removed? If no, explain why not.
9. Consider the dining philosophers problem. If we place all five chopsticks in the centre of the table, how could we use semaphores to implement the philosophers eat () method? Be sure to declare and initialize all variables you use. Your solution should be as efficient as possible. You should define two methods begin () and end ().

6.11 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.
- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.
- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.
- Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.
- Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.
- Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

BLOCK - III
DEADLOCK AND INFINITE POSTPONEMENT

*Introduction to
Deadlocks*

**UNIT 7 INTRODUCTION TO
DEADLOCKS**

NOTES

Structure

- 7.0 Introduction
 - 7.1 Objectives
 - 7.2 What is Deadlock?
 - 7.3 Deadlock Characterization
 - 7.3.1 Example of Deadlock Conditions
 - 7.4 Indefinite Postponement
 - 7.5 Resource Concept
 - 7.6 Answers to Check Your Progress Questions
 - 7.7 Summary
 - 7.8 Key Words
 - 7.9 Self Assessment Questions and Exercises
 - 7.10 Further Readings
-

7.0 INTRODUCTION

A deadlock is a situation in which some processes wait for each other's actions indefinitely.

A deadlock is a situation in which each member of a community waits for another member, including self, to take action in concurrent computing, such as sending a message or releasing a lock more commonly. In multiprocessor systems, parallel computing, and distributed systems, deadlock is a common problem where software and hardware locks are used to arbitrate shared resources and implement synchronization of processes.

A deadlock occurs in an operating system when a process or thread enters a waiting state because another waiting process is keeping a requested system resource, which in turn is waiting for another resource held by another waiting process. If a process is unable to change its state indefinitely since another waiting process uses the resources requested by it, then the system is considered to be in a deadlock.

Deadlock detection mechanism detects a deadlock by checking whether all conditions necessary for a deadlock hold simultaneously. The deadlock prevention and deadlock avoidance ensure that deadlocks cannot occur, by not allowing the conditions for deadlocks to hold simultaneously.

NOTES

The occurrence of deadlocks is common in the multiprocessing system. The reason for this is that in the multiprocessing system, several processes have to share a specific type of mutually exclusive resource known as a software, or soft, lock.

The pictorial representation of the state of a system is the resource allocation graph. The resource allocation graph, as its name suggests, is the full information about all the processes that keep certain resources or wait for certain resources. Vertices are primarily of two kinds, resource and process.

In OS, Deadlock is a condition in which two or more operations are interrupted. Deadlock Conditions- Mutual Exclusion, Hold and Wait, No Pre-emption, Wait Circular. For the occurrence of deadlock, these 4 conditions must simultaneously hold.

In this unit, you will study about the basic concept of deadlock, related problem indefinite postponement and resource concept.

7.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basic concept of deadlock in operating system
- Know about the Resource Allocation Graph (RAG)
- Discuss about the deadlock conditions
- Overview of the indefinite postponement and resource concepts

7.2 WHAT IS DEADLOCK?

Definition: In concurrent computing, a deadlock is a state in which each member of a group waits for another member, including itself, to take action, such as sending a message or more commonly releasing a lock.

A set of processes is said to be in a state of deadlock if each process present in the system is waiting for an *event* to occur, which is caused by another process in the set. The *event* is the release of a resource held by the other process.

When a deadlock occurs in a system, then none of the processes can run or release resources.

The following are the examples of a deadlock:

Example 1: In a system that consists of two tape drives and two processes P1, P2, P1 is holding a tape drive and waiting for another one. Similarly, process P2 is also holding a tape drive and waiting for another one. A process can be finished if all the resources requested by it are allocated. In the above processes, each needs two tape drives but only one is allocated to each of them. Each process is waiting for the second tape drive.

Example 2: Consider the semaphores P and Q, initialized to 1

<i>Process1</i>	<i>Process2</i>
wait (P);	wait (Q)
wait (Q);	wait (P)

*Introduction to
Deadlocks*

NOTES

Example 3: Let the space available for allocation to processes be 200 Kilo bytes. Consider the following sequence of events that occurs with two processes:

P0	P1
....	
Request 80KB;	
...	Request 70KB;
Request 60KB;	
...	Request 80KB;

A deadlock may occur if both the processes try to execute their second request after executing their first instruction.

7.3 DEADLOCK CHARACTERIZATION

A deadlock is a situation in which some processes wait for each other's actions indefinitely. In real life deadlocks can arise when two processes wait for phone calls from one another or when persons crossing a narrow bridge in opposite directions meet in the middle of the bridge. Deadlock is more serious than indefinite postponement or starvation because it affects more than one job. Because resources are tied up in deadlocks, the entire system is affected.

Processes involved in a deadlock remain blocked permanently which affects the throughput, resource efficiency and the performance of the operating system. A deadlock can bring the system to standstill.

Operating system handles only deadlocks caused by sharing of resources in the system. Such deadlocks arise when some conditions concerning resource requests and resource allocations are held simultaneously.

Deadlock detection mechanism detects a deadlock by checking whether all conditions necessary for a deadlock hold simultaneously. The deadlock prevention and deadlock avoidance ensure that deadlocks cannot occur, by not allowing the conditions for deadlocks to hold simultaneously.

The most common example for deadlock is a traffic jam. In the example (refer Figure 7.1) shown below, there is no proper solution to a deadlock; no one can move forward until someone moves out of the way, but no one can move out of the way until either someone advances or a rear of a line moves back. Only then can the deadlock be resolved.

NOTES

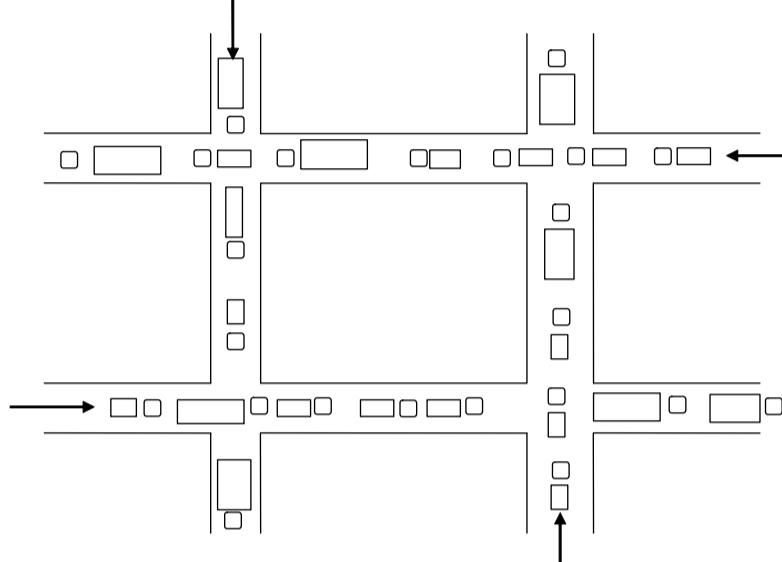


Fig. 7.1 A Classic Case of Traffic Deadlock

Thus, we can say that a deadlock refers to a situation in which two or more competing actions are waiting for the other to finish and thus neither ever does this. For example, consider about the two trains approaching each other at a crossing. In this situation, both the trains stop and none of them can restart until the other has gone.

In computer science, deadlock refers to a specific condition when two or more processes are each waiting for another to release a resource or more than two processes are waiting for resources in a circular chain.

Following are the examples of deadlock:

- The occurrence of deadlocks is common in the multiprocessing system. The reason for this is that in the multiprocessing system, several processes have to share a specific type of mutually exclusive resource known as a *software*, or *soft*, lock. There often exists a *hardware lock* (or *hard lock*) in computers that intend for the *time sharing* and/or *real time* markets. This lock provides an *exclusive access* to processes. This leads to a forced serialization. Deadlocks create troubles as we lack a general solution to this problem.
- Think of two people drawing diagrams with only one pencil and one ruler between them. If one person possesses the pencil and the other possesses the ruler, this would lead to a deadlock if the person having the pencil needs the ruler and vice versa. As it is not possible to satisfy both the requests, a deadlock is inevitable.
- In case of telecommunications, deadlock is a little more complex. Here, deadlock occurs when neither of the processes meets the condition for

NOTES

moving to another state and each communication channel is empty. The second condition is ignored in case of other systems but is very important in the context of telecommunications.

- We may consider an example of a deadlock in database products. Client applications using the database may need an exclusive access to a table. To acquire such an access, a *lock* may be demanded by the applications. Think of a client application holding a lock on a table and attempting to obtain the lock on a second table which is already held by a second client application. This may lead to deadlock if the second application tries to obtain the lock possessed by the first application. However, this particular type of deadlock is easily prevented, for example by using an *all-or-none* resource allocation algorithm.

When a Deadlock Occurs?

A deadlock occurs when the following four conditions are met:

- **Mutual Exclusion:** Each resource is allocated to only one process at any given point of time.
- **Hold and Wait:** The previously granted resources are not released by processes.
- **No Preemption:** The previously granted resources are not taken away from the processes which hold them.
- **Circular Wait:** There exists a chain of two or more processes. These processes should exist in such a way that each process in the chain holds a resource requested by the next process in the chain; there must exist a set ($P_0, P_1, P_2, P_3, \dots, P_n$) of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , \dots , P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 (refer Figure 7.2).

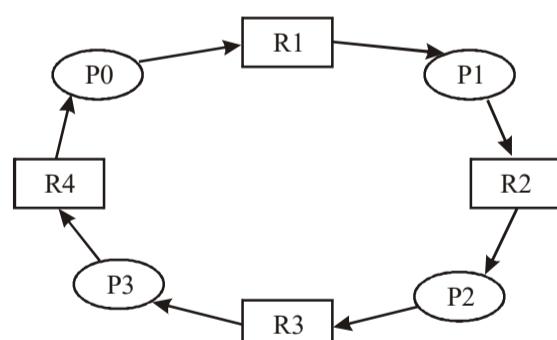


Fig. 7.2 Circular Wait

NOTES

Basics of Resource Allocation Graph or RAG

The pictorial representation of the state of a system is the resource allocation graph. The resource allocation graph, as its name suggests, is the full information about all the processes that keep certain resources or wait for certain resources. Vertices are primarily of two kinds, resource and process.

Three events concerning resource allocation can occur in a system: request for a resource, allocation of a resource and release of a resource.

A request can occur when some process P_i makes a request for a resource R_i . If R_i is currently allocated to some process P_k , process P_i gets blocked on an allocation event r_i . In effect, P_i waits for process P_k so that R_i is released. A release event does the task to free R_i by P_k . Table 7.1 shows the function of request allocation and release of resources.

Table 7.1 Request Allocation and Release of Resources

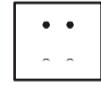
Request	A process requests a resource through a system call. If the resource is free, the kernel allocates it to the process immediately; otherwise, it changes the state of the process to block.
Allocation	The process becomes the holder of the resource allocated to it. The resource state information gets updated and the process state changes to ready.
Release	A process releases a resource through a system call. If some processes are blocked on the allocation event for the resource, the kernel uses some tie-breaking rule, for example FCFS allocation, to decide which process should be allocated the resource

Symbols used in RAG

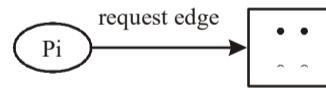
1. Process



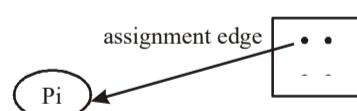
2. Resource type with four instances



3. P_i requests instances of R_j



4. Process P_i is holding an instance of R_j



Basic Facts

What are the basic facts of RAG.

1. If a graph contains no cycles, there will be no deadlock.
2. If a graph contains a cycle
 - If only one instance per resource type, then a deadlock will occur.
 - If several instances per resource type, possibility of a deadlock.

Possibility 1 for a Deadlock

The possibility 1 for a deadlock is shown in Figure 7.3.

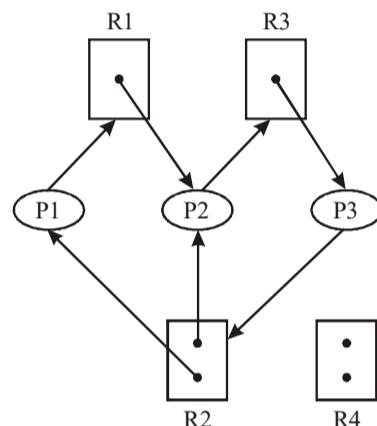


Fig. 7.3 Possibility 1 for a Deadlock

$$E = \{P1 \rightarrow R1, P2 \rightarrow R3, R1 \rightarrow P2, R2 \rightarrow P2, R2 \rightarrow P1, R3 \rightarrow P3\}$$

Two possibility for cycles are as follows:

- $P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$
- $P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$

Possibility 2 Cycles but No Deadlock

Two possibility cycles but for no deadlock is shown in Figure 7.4.

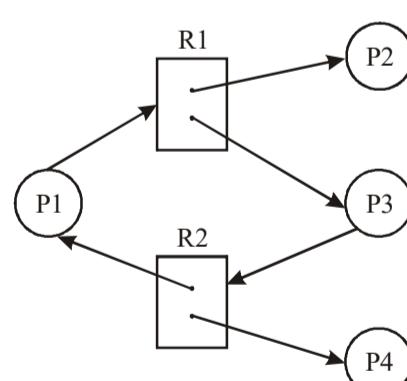


Fig. 7.4 Possibility 2 for a Deadlock

NOTES

NOTES

Cycle is P1 → R1 → P3 → R2 → P1.

No deadlock, observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3 breaking this cycle.

7.3.1 Example of Deadlock Conditions

Two processes, A and B (applications) need the same file resources F1 and F2 to complete their task. Initially, file F1 is allocated to process A, and F2 is allocated to process B. Suppose each of the processes starts execution in this state of resource allocation. Now, during the execution, suppose process A requests file F2 which is in use by B, and process B requests file F1 which is in use by A. That is, now process A is waiting for file F2 that is held by process B, and process B is also waiting for file F1 that is held by process A. The processes are in a circular wait state. The chances that such a circular wait will occur can be negated by adopting the strategy of allocating all or none of the resources to each of the processes. That is, if all of the resources needed for a process are available, they are all allocated before the start of execution; otherwise, no resources are allocated and the process has to wait. This negates the *hold* and *wait* condition and hence prevents circular wait.

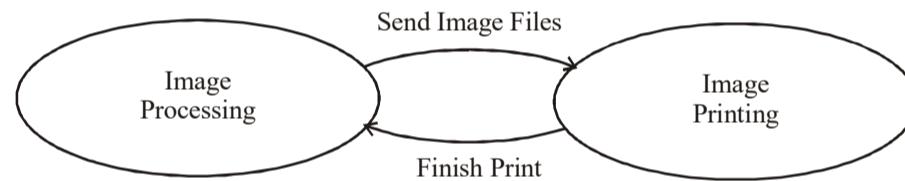


Fig. 7.5 Circular Wait May Occur between Image Processing and Printing System

For example:

Consider the photo processing and printing system given in Figure 7.5. The image processing system sends images in lots of ten for printing to the image printing system and waits for *Finish-Print* message from the image printer. The image printing system accepts images sent by the image processing system one by one, and when it receives ten images, it prints them all in a single sheet of paper to minimize the waste of printer stationery. The image printing system sends the *Finish-Print* message back to the image processing system after printing every ten images. Suppose the image processing system had only nine images to be printed in a single sheet and it sends them all and waits for the *Finish-Print* message. The image printer receives the nine images and then waits for the tenth one before starting printing. The image processing system also waits for the *Finish-Print* message for it to send the next lot of images for printing. This is an example of a circular wait (deadlocked state) in which the image printer is waiting for the event (of sending the tenth image) from the image processor and the image processor is waiting for the *Finish-Print* message from the image printer to send the next lot of images.

Suppose the image processor sends the last image indicator of a lot. Then the image printer can start printing with whatever number of images it has received. In this way the cause of circular wait is eliminated.

Introduction to Deadlocks

Check Your Progress

1. What do you mean by Deadlock in operating System?
2. How operating system handle deadlock condition?
3. Give an example of deadlock situation.
4. If system has 3 processes sharing 4 resources. If each process needs a maximum of 2 units then, deadlock occur or not?
5. In which type of system mainly deadlock situation occurs?
6. Describe the four necessary conditions for deadlock.
7. How circular wait condition can be prevented?

NOTES

7.4 INDEFINITE POSTPONEMENT

Indefinite Postponement - to delay the scheduling of a process indefinitely while the attention of the system is obtained by other processes.

This condition may be as catastrophic as deadlock, variously called indefinite postponement, indefinite blocking, or famine. Indefinite postponement may occur due to prejudices in the resource scheduling policies of a system. By increasing the priority of a process as it waits for a resource, certain systems avoid permanent postponement. This strategy is called ageing.

It is possible to indefinitely delay the scheduling of a process in any system that keeps processes waiting when making resource-allocation and process scheduling decisions, while other processes receive the attention of the system. This condition may be as catastrophic as deadlock, variously called indefinite postponement, indefinite blocking, or famine.

7.5 RESOURCE CONCEPT

Operating System (OS), a program that manages the resources of a computer, especially the allocation, among other programs, of those resources. The Central Processing Unit (CPU), computer memory, file storage, Input/Output (I/O) devices and network connections are common tools. To avoid conflicts and interference between programmes, management tasks involve scheduling resource use. An operating system runs indefinitely and terminates only when the machine is switched off, unlike most programmes, which complete a task and terminate.

NOTES

In computing, any physical or virtual part with limited availability within a computer system is a system resource, or simply a resource. A resource is every device linked to a computer system. A resource is any internal system variable. Files (specifically file handles), network connections (specifically network sockets), and memory areas provide virtual machine resources. Resource management is known as resource management which involves both the prevention of resource leakage (not the release of a resource when a process has completed its use) and the management of resource contention (when multiple processes wish to access a limited resource). In cloud computing, computing resources are used to provide services via networks.

Check Your Progress

8. What is indefinite postponement?
9. Explain the term is aging.
10. Discuss about the resource concepts.

7.6 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. In concurrent computing, a deadlock is a state in which each member of a group waits for another member, including itself, to take action, such as sending a message or more commonly releasing a lock.
2. Operating system handles only deadlocks caused by sharing of resources in the system. Such deadlocks arise when some conditions concerning resource requests and resource allocations are held simultaneously.
3. The most common example for deadlock is a traffic jam. In the example (refer Figure 7.1) shown below, there is no proper solution to a deadlock; no one can move forward until someone moves out of the way, but no one can move out of the way until either someone advances or a rear of a line moves back. Only then can the deadlock be resolved.
4. If system has 3 processes sharing 4 resources. If each process needs a maximum of 2 units then, deadlock can never occur.
5. The occurrence of deadlocks is common in the multiprocessing system.
6. A deadlock occurs when the following four conditions are met:
 - **Mutual Exclusion:** Each resource is allocated to only one process at any given point of time.
 - **Hold and Wait:** The previously granted resources are not released by processes.

- **No Preemption:** The previously granted resources are not taken away from the processes which hold them.

Introduction to Deadlocks

- **Circular Wait:** There exists a chain of two or more processes.
7. The circular wait condition can be prevented by defining a linear ordering of resource types.
 8. Indefinite postponement happens when waiting for a method is blocked and even that can, but will not occur in a potential execution sequence.
 9. Certain systems resist the strategy of permanent postponement, which is called **ageing**.
 10. Resource concept means management of resources.

NOTES

7.7 SUMMARY

- A deadlock is a situation in which some processes wait for each other's actions indefinitely.
- A deadlock is a situation in which some processes wait for each other's actions indefinitely.
- Processes involved in a deadlock remain blocked permanently which affects the throughput, resource efficiency and the performance of the operating system. A deadlock can bring the system to standstill.
- Deadlock is more serious than indefinite postponement or starvation because it affects more than one job. Because resources are tied up in deadlocks, the entire system is affected.
- Deadlock detection mechanism detects a deadlock by checking whether all conditions necessary for a deadlock hold simultaneously. The deadlock prevention and deadlock avoidance ensure that deadlocks cannot occur, by not allowing the conditions for deadlocks to hold simultaneously.
- Deadlock refers to a specific condition when two or more processes are each waiting for another to release a resource or more than two processes are waiting for resources in a circular chain.
- The occurrence of deadlocks is common in the multiprocessing system. The reason for this is that in the multiprocessing system, several processes have to share a specific type of mutually exclusive resource known as a software, or soft, lock.
- In case of telecommunications, deadlock is a little more complex. Here, deadlock occurs when neither of the processes meets the condition for moving to another state and each communication channel is empty. The second condition is ignored in case of other systems but is very important in the context of telecommunications.

NOTES

- In Mutual Exclusion each resource is allocated to only one process at any given point of time.
- Three events concerning resource allocation can occur in a system: request for a resource, allocation of a resource and release of a resource.
- A process requests a resource through a system call. If the resource is free, the kernel allocates it to the process immediately; otherwise, it changes the state of the process to block.
- If a graph contains no cycles, there will be no deadlock.
- The process becomes the holder of the resource allocated to it. The resource state information gets updated and the process state changes to ready.
- A process releases a resource through a system call. If some processes are blocked on the allocation event for the resource, the kernel uses some tie-breaking rule, for example FCFS allocation, to decide which process should be allocated the resource.
- If a graph contains no cycles, there will be no deadlock.
- The pictorial representation of the state of a system is the resource allocation graph. The resource allocation graph, as its name suggests, is the full information about all the processes that keep certain resources or wait for certain resources. Vertices are primarily of two kinds, resource and process.
- To delay the scheduling of a process indefinitely while the attention of the system is obtained by other processes.
- It is possible to indefinitely delay the scheduling of a process in any system that keeps processes waiting when making resource-allocation and process scheduling decisions, while other processes receive the attention of the system.
- Operating System (OS), a program that manages the resources of a computer, especially the allocation, among other programs, of those resources.
- Aging is increasing the priority of jobs to ensure termination in a finite time.
- A resource manager is essentially an operating system. It is responsible for the distribution of different types of a large array of resources.

7.8 KEY WORDS

- **Deadlock:** Dead-Lock is a situation where two or more processors are waiting for an event to occur, but it is a deadlock condition for certain events that do not occur, and the processors are considered to be in a deadlock state.

- **Mutual Exclusion:** Each resource is allotted to only one process at any given point of time.
- **Hold and wait:** Process holding a resource allocated to it and waiting to acquire another resource held by other process.
- **No pre-emption:** Resource allocated to a process cannot be forcibly revoked by the system, it can only be released voluntarily by the process holding it.
- **Resource Allocation Graph (RAG):** Three events concerning resource allocation can occur in a system: request for a resource, allocation of a resource and release of a resource.
- **Aging:** By increasing the priority of a process as it waits for a resource, certain systems avoid permanent postponement.

Introduction to Deadlocks

NOTES

7.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What do you mean by deadlock?
2. Explain deadlock condition with example.
3. When deadlock occur in operating system?
4. State the different condition in deadlock.
5. Explain Circular Wait condition with example.
6. What do you mean by RAG?
7. What are the basic facts of resource allocation graph?

Long-Answer Questions

1. Describe in detail when deadlock condition occur in operating system with example.
2. Describe four ways one might deal with deadlock with a brief description of each possibility and how it is achieved.
3. Which type of symbols used in Resource Allocation Graph (RAG)?
4. Briefly illustrate the deadlock conditions giving suitable examples.
5. Elaborate on the indications required to explain any resource allocation graph for multiple instances of a resource type.
6. Discuss the importance of deadlock prevention and briefly describe the deadlock avoidance techniques.
7. Explain indefinite postponement in detail.
8. Briefly discuss the resource concept.

NOTES

7.10 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.
- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.
- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.
- Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.
- Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.
- Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

UNIT 8 CONDITIONS FOR DEADLOCK

NOTES**Structure**

- 8.0 Introduction
 - 8.1 Objectives
 - 8.2 Handling Deadlocks
 - 8.3 Deadlock Prevention
 - 8.4 Deadlock Avoidance
 - 8.5 Deadlock Detection
 - 8.6 Deadlock Recovery
 - 8.7 Answers to Check Your Progress Questions
 - 8.8 Summary
 - 8.9 Key Words
 - 8.10 Self Assessment Questions and Exercises
 - 8.11 Further Readings
-

8.0 INTRODUCTION

A deadlock is a situation in which each member of a community waits for another member, including self, to take action in concurrent computing, such as sending a message or releasing a lock. In multiprocessor systems, parallel computing, and distributed systems, deadlock is a common problem where software and hardware locks are used to arbitrate shared resources and implement synchronization of processes.

A deadlock occurs when every process in a set of processes is in a simultaneous wait state and each of them is waiting for the release of a resource held exclusively by one of the waiting processes in the set. None of the processes can proceed until at least one of the waiting processes releases the acquired resource. Deadlocks may occur on a single system or across several machines.

Deadlock prevention works by preventing one of the four Coffman conditions from occurring. Removing the mutual exclusion condition means that no process will have exclusive access to a resource. This proves impossible for resources that cannot be spooled. But even with spooled resources, the deadlock could still occur.

A deadlock can be prevented by eliminating any one of the four necessary conditions of the deadlock. Preventing deadlock using this method results in the inefficient use of resources.

In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. In order to

NOTES

avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution. We use Dijkstra's algorithm for deadlock avoidance.

Dijkstra's algorithm can be used to calculate the shortest path within the same graph data structure from one node in a graph to some other node, provided the nodes can be reached from the starting node. To find the shortest path, Dijkstra's algorithm can be used.

Deadlock recovery performs when a deadlock is detected. When deadlock detected, then our system stops working, and after the recovery of the deadlock, our system start working again. Therefore, after the detection of deadlock, a method/way must require to recover that deadlock to run the system again.

In this unit, you will study about the basic concept of deadlock solution, prevention, avoidance, Dijkstra's banker algorithm, deadlock detection and deadlock recovery.

8.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain how a deadlock can be prevented by eliminating the conditions of a deadlock
- Understand the concept of safe and unsafe state
- Discuss about the various deadlock avoidance algorithms
- Analyse the different deadlock detection methodologies
- Describe the ways to recover from a deadlock

8.2 HANDLING DEADLOCKS

A deadlock is a situation in which each member of a community waits for another member, including self, to take action in concurrent computing, such as sending a message or releasing a lock.

A deadlock can be handled in four different ways which are as follows:

- Prevent the deadlock from occurring.
- Adopt methods for avoiding the deadlock.
- Allow the deadlock to occur, detect it and recover from it.
- Ignore the deadlock.

Deadlock prevention or deadlock avoidance techniques can be used to ensure that deadlocks never occur in a system. If any of these two techniques is not used, a deadlock may occur. In this case, an algorithm can

be provided for detecting the deadlock and then using the algorithm to recover the system from the deadlock.

Conditions for Deadlock

One or the other method must be provided to either prevent the deadlock from occurrence or detect the deadlock and take an appropriate action if a deadlock has occurred. However, if in a system, deadlock occurs less frequently, say, once in two years, then it is better to ignore the deadlocks instead of adopting expensive techniques for deadlock prevention, deadlock avoidance, or deadlock detection and recovery.

NOTES

8.3 DEADLOCK PREVENTION

As stated earlier, a deadlock occurs when all of the four conditions are satisfied at any point of time. The deadlock can be prevented by not allowing all four conditions to be satisfied simultaneously; that is, by making sure that at least one of the four conditions does not hold. Now, let us analyse all four conditions one by one and see how their occurrence can be prevented.

Eliminating Mutual Exclusion

The mutual exclusion property does not hold for the resources that are sharable. For example, a file opened in read-only mode can be shared among various processes. Hence, processes will never have to wait for the sharable resources. However, there are certain resources which can never be shared, like printer can work for only one process at a time. It cannot print data being sent as output from more than one process simultaneously. Hence, the condition of mutual exclusion cannot be eliminated in case of all the resources.

Eliminating Hold and Wait Condition

This condition can be eliminated by not allowing any process to request for a resource until it releases the resources held by it, which is impractical as process may require the resources simultaneously. Another way to prevent hold and wait condition is by allocating all the required resources to the process before starting the execution of that process. The disadvantage associated with it is that a process may not know in advance about the resources that will be required during its execution. Even if it knows in advance, it may unnecessarily hold the resources which may be required at the end of its execution. Thus, the resources are not utilized optimally.

Eliminating No Preemption

The elimination of this condition means a process can release the resource held by it. If a process requests for a resource held by some other process then instead of making it wait, all the resources currently held by this process can be preempted. The process will be restarted only when it is allocated the requested as well as the preempted resources. Note that only those resources can be preempted whose

NOTES

current working state can be saved and can be later restored. For example, the resources like printer and disk drives cannot be preempted.

Eliminating Circular-Wait Condition

The circular-wait condition can be eliminated by assigning a priority number to each available resource and a process can request resources only in increasing order. Whenever, a process requests for a resource, the priority number of the required resource is compared with the priority numbers of the resources already held by it. If the priority number of a requested resource is greater than that of all the currently held resources, the request is granted. If the priority number of a requested resource is less than that of the currently held resources, all the resources with greater priority number must be released first, before acquiring the new resource.

8.4 DEADLOCK AVOIDANCE

A deadlock can be prevented by eliminating any one of the four necessary conditions of the deadlock. Preventing deadlock using this method results in the inefficient use of resources. Thus, instead of preventing deadlock, it can be avoided by never allowing allocation of a resource to a process if it leads to a deadlock. This can be achieved when some additional information is available about how the processes are going to request for resources in future. Information can be in the form of the number of resources of each type that will be requested by a process and in which order. On the basis of the amount of information available, different algorithms can be used for deadlock avoidance.

One of the simplest algorithms requires each process to declare the maximum number of resources (of each type) required by it during its course of execution. This information is used to construct an algorithm that will prevent the system from entering a state of deadlock. This deadlock avoidance algorithm continuously examines the state of resource allocation ensuring that circular-wait condition never exists in a system. The state of resource allocation can be either safe or unsafe.

Safe and Unsafe State of Resource Allocation

A state is said to be **safe** if allocation of resources to processes does not lead to the deadlock. More precisely, a system is in safe state only if there is a safe sequence. A **safe sequence** is a sequence of process execution such that each and every process executes till its completion. For example, consider a sequence of processes ($P_1, P_2, P_3, \dots, P_n$) forming a safe sequence. In this sequence, first the process P_1 will be executed till its completion, and then P_2 will be executed till its completion, and so on. The number of resources required by any process can be allocated either from the available resources or from the resources held by previously executing process. When a process completes its execution, it releases all the resources held by it which then can be utilized by the next process in a sequence.

That is, the request for the resources by the process P_n can be satisfied either from the available resources or from the resources held by the process P_m , where $m < n$. Since this sequence of process execution is safe, the system following it is in the safe state. If no such sequence of process execution exists then the state of the system is said to be **unsafe** (refer Figure 8.1).

Conditions for Deadlock

NOTES

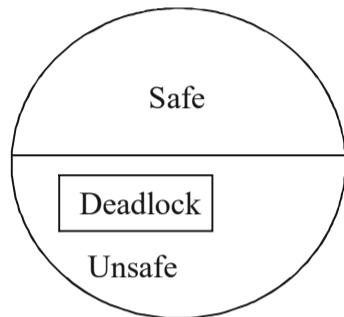


Fig. 8.1 Relationship between Safe State, Unsafe State and Deadlock

Consider a system in which three processes P_1 , P_2 and P_3 are executing and there are 10 instances of a resource type. The maximum number of resources required by each process, the number of resources already allocated and the total number of available resources are shown in Figure 8.2.

	Maximum	Currently allocated		Maximum	Currently allocated
P_1	9	3		9	3
P_2	4	2		0	0
P_3	7	2		7	2
<i>Available resources = 3</i>					
(a) Initial State					
	Maximum	Currently allocated		Maximum	Currently allocated
P_1	9	3		9	3
P_2	4	4		0	0
P_3	7	2		7	2
<i>Available resources = 1</i>			<i>Available resources = 5</i>		
(b) Resource Allocation to Process P_2			(c) State after Completion of Process P_2		
	Maximum required	Currently allocated		Maximum required	Currently allocated
P_1	9	3		9	3
P_2	0	0		0	0
P_3	7	7		0	0
<i>Available resources = 0</i>			<i>Available resources = 7</i>		
(d) Resource Allocation to Process P_3			(e) State after Completion of Process P_3		
	Maximum required	Currently allocated		Maximum required	Currently allocated
P_1	9	3		9	3
P_2	0	0		0	0
P_3	7	0		0	0

<i>Conditions for Deadlock</i>	<i>Maximum required</i>	<i>Currently allocated</i>	<i>Maximum required</i>	<i>Currently allocated</i>
P ₁	9	9	P ₁	0
P ₂	0	0	P ₂	0
P ₃	0	0	P ₃	0
NOTES	Available resources = 1			Available resources = 10
	(f) Resource Allocation to Process P₁		(g) State after Completion of Process P₁	

Fig. 8.2 Safe Sequence of Execution of Processes

On the basis of available information, it can be easily observed that the resource requirements of the process P_2 can be easily satisfied. Therefore, resources are allocated to the process P_2 and it is allowed to execute till its completion. After the execution of the process P_2 , all the resources held by it are released. The number of the resources now available are not enough to be allocated to the process P_1 , whereas, they are sufficient for the process P_3 . Therefore, resources are allocated to the process P_3 and it is allowed to execute till its completion. The number of resources available after the execution of process P_3 can now easily be allocated to the process P_1 . Hence, the execution of the processes in the sequence P_2, P_3, P_1 is safe.

Now consider a sequence P_2, P_1, P_3 . In this sequence, after the execution of the process P_2 , the number of available resources is 5, and is allocated to the process P_1 . Even after the allocation of all the available resources, the process P_1 is still short of one resource for its complete execution. As a result, the process P_1 enters a waiting state and waits for the process P_3 to release the resource held by it, which in turn is waiting for the remaining resources to be allocated for its complete execution. Now the processes P_1 and P_3 are waiting for each other to release the resources, leading to a deadlock. Hence, this sequence of process execution is unsafe.

Note that a safe state is a deadlock-free state, whereas all unsafe states may or may not result in a deadlock. That is, an unsafe state may lead to a deadlock but not always.

Resource Allocation Graph Algorithm

As discussed earlier, resource allocation graph consists of two types of edges: request edge and assignment edge. In addition to these edges, another edge known as **claim edge** can also be introduced in this graph, which helps in avoiding the deadlock. A claim edge from a process to the resource indicates that the process will request for that resource in near future. This edge is represented to be same as that of request edge but with dotted line. Whenever the process actually requests for that resource, the claim edge is converted to the request edge. Also, whenever a resource is released by any process, corresponding assignment edge is converted back to the claim edge. The pre-requisite of this representation is that all the claim

edges related to a process must be depicted in the graph before the process starts executing. However, a claim edge can be added at the later stage only if all the edges related to that process are claim edges.

Whenever, the process requests for a resource, the claim edge is converted to request edge only if converting the corresponding request edge to assignment edge does not lead to the formation of a cycle in a graph, as cycle in a graph indicates the deadlock. For example, consider the resource allocation graph shown in Figure 8.3, the claim edge from process P_1 to the resource R_1 cannot be converted to the request edge as it will lead to the formation of cycle in the graph.

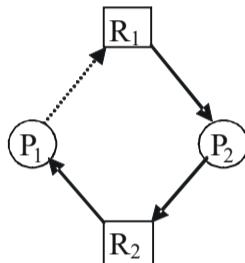


Fig. 8.3 Resource Allocation Graph with Claim Edges

Dijkstra's Banker's Algorithm

In case there are multiple instances of a resource type in a system, the deadlock cannot be avoided using resource allocation graph algorithm. This is because the presence of cycle in the resource allocation graph for multiple resources does not always imply the deadlock. In such cases, an algorithm known as **Banker's algorithm** is used. In this algorithm, any process entering the system must inform the maximum number of resources (less than the total number of available resources) required during its execution. If allocating this much number of resources to the process leaves the system in a safe state only then the resources are allocated. On the other hand, if allocation of resources leaves the system in unsafe state then the resources are not allocated and the process is made to wait for some other processes to release enough resources. To implement the banker's algorithm certain data structures are required, which help in determining whether the system is in safe state or not. These data structures are as follows:

1. **Available Resources, A:** A vector of size q stores information about the number of resources available of each type.
2. **Maximum, M:** A matrix of order $p \times q$ stores information about the maximum number of resources of each type required by each process (p number of processes). That is, $M[i][j]$ indicates the maximum number of resources of type j required by the process i .
3. **Current Allocation, C:** A matrix of order $p \times q$ stores information about the number of resources of each type allocated to each process. That is, $C[i][j]$ indicates the number of resources of type j currently held by the process i .

Conditions for Deadlock

NOTES

NOTES

4. **Required, R:** A matrix of order $p \times q$ stores information about the remaining number of resources of each type required by each process. That is, $R[i][j]$ indicates the remaining number of resources of type j required by the process i . Note that this vector can be obtained by $M - C$, that is, $R[i][j] = M[i][j] - C[i][j]$.

The values of these data structures keep on changing during the execution of processes. Note that the condition $A \leq B$ holds for the vectors A and B of size p , if and only if $A[i] \leq B[i]$ for all $i=1, 2, 3, \dots, p$. For example, if $A=\{2,1\}$ and $B=\{3,4\}$, then $A \leq B$.

Safety Algorithm

This algorithm is used to determine whether a system is in safe state.

To understand the algorithm for determining whether a system is in safe state, consider a vector Complete of size p . Following are the steps of the algorithm.

1. Initialize $\text{Complete}[i] = \text{False}$ for all $i=1, 2, 3, \dots, p$. $\text{Complete}[i]=\text{False}$ indicates that the i th process is still not completed.
2. Search for an i , such that $\text{Complete}[i]=\text{False}$ and $(R \leq A)$ that is, resources required by this process is less than the available resources. If no such process exists, then go to Step 4.
3. Allocate the required resources and let the process finish its execution and set $\text{Complete}[i]=\text{True}$ for that process. Go to Step 2.
4. If $\text{Complete}[i]=\text{True}$ for all i , then the system is in safe state. Otherwise, it indicates that there exist a process for which $\text{Complete}[i]=\text{False}$ and resources required by it are more than the available resources. Hence, it is in unending waiting state leading to an unsafe state.

Resource-Request Algorithm

Once it is confirmed that system is in safe state, an algorithm called **resource-request** algorithm is used for determining whether the request by a process can be satisfied or not. To understand this algorithm, let Req be a matrix of the order $p \times q$, indicating the number of resources of each type requested by each process at any given point of time. That is, $\text{Req}[i][j]$ indicates the number of resources of j th type requested by the i th process at any given point of time. Following are the steps of this algorithm:

1. If $\text{Req}[i][j] \leq R[i][j]$, go to Step 2, otherwise an error occurs as process is requesting for more resources than the maximum number of resources required by it.

2. If $\text{Req}[i][j] \leq \text{A}[i][j]$, go to Step 3, otherwise the process P_i must wait until the required resources are available.
3. Allocate the resources are allocated and make the following changes in the data structures.

$$\begin{aligned} A &= A - \text{Req} \\ C &= C + \text{Req} \\ R &= R - \text{Req} \end{aligned}$$

Conditions for Deadlock

NOTES

For example, consider a system with three processes (P_1, P_2 and P_3) and three resource types (X, Y and Z). There are 10 instances of resource type X , 5 of Y and 7 of Z . The matrix M for maximum number of resources required by the process, matrix C for the number of resources currently allocated to each process and vector A for maximum available resources are shown in Figure 8.4.

M			C		
X	Y	Z	X	Y	Z
P_1	7	5	P_1	0	1
P_2	5	2	P_2	2	0
P_3	9	0	P_3	3	0
					2

(a) Maximum Matrix	(b) Current Allocation Matrix
$\begin{array}{c} M \\ \hline X & Y & Z \\ \hline P_1 & 7 & 5 & 6 \\ P_2 & 5 & 2 & 2 \\ P_3 & 9 & 0 & 2 \end{array}$	$\begin{array}{c} C \\ \hline X & Y & Z \\ \hline P_1 & 0 & 1 & 0 \\ P_2 & 2 & 0 & 0 \\ P_3 & 3 & 0 & 2 \end{array}$
(c) Available Resources vector	(d) Required Matrix

Fig. 8.4 Initial State of System

Now, the matrix R representing the number of remaining resources required by each process can be obtained by the formula $M-C$, which is shown in Figure 8.4.

It can be observed that currently the system is in safe state and safe sequence of execution of processes is (P_2, P_3, P_1). Now suppose that process P_2 requests one more resource of each type, that is, the request vector for process P_2 is $(1,1,1)$. First, it is checked whether this request vector is less than or equal to its corresponding required vector $(3,2,2)$. If the process has requested for less number of resources than the declared maximum number of resources of each type by it at initial stage, then it is checked whether these much number of resources of each type are available. If it is then it is assumed that the request is granted, and the changes will be made in the corresponding matrices shown in Figure 8.5.

Conditions for Deadlock

NOTES

	X	Y	Z	M
P ₁	7	5	6	
P ₂	5	2	2	
P ₃	9	0	2	

	X	Y	Z	C
P ₁	0	1	0	
P ₂	3	1	1	
P ₃	3	0	2	

(a) Maximum Matrix

(b) Current Allocation Matrix

	X	Y	Z	A
P ₁	4	3	4	
P ₂				
P ₃				

(c) Available Resources vector

(d) Required Matrix

(c) Available Resources Vector (d) Required Matrix

Fig. 8.5 State after Granting Request of P2

This new state of system must be checked whether it is safe. For this, an algorithm is executed to check the safe state of the system and it is determined that the sequence (P₂, P₃, P₁) is a safe sequence. Thus, the request of process P₂ is granted immediately.

Consider another state of a system shown in Figure 8.6. Now, a request for (1, 2, 2) from process P₂ arrives. If this request is granted, the resulting state is unsafe. This is because after the complete execution of process P₂, the resultant vector A is (5, 4, 5). Clearly, the resource requirement of processes P₁ and P₃ cannot be satisfied. Thus, even though the system has resources, request cannot be granted.

	X	Y	Z	M
P ₁	7	5	6	
P ₂	5	2	2	
P ₃	9	0	2	

	X	Y	Z	C
P ₁	2	1	0	
P ₂	4	0	0	
P ₃	3	0	2	

(a) Maximum Matrix

(b) Current Allocation Matrix

	X	Y	Z	A
P ₁	1	4	5	
P ₂				
P ₃				

(c) Available Resources Vector

(d) Required Matrix

Fig. 8.6 Example of Unsafe State

Check Your Progress

1. State the four different ways to handle deadlock.
2. List the conditions necessary for a deadlock to occur.
3. If the requested resource is not available, the requesting process enters a _____ until it acquires the resource.
4. Define the term of deadlock avoidance.
5. Which of the following is not associated with the resource allocation graph to depict a deadlock?
 - (a) Request edge
 - (b) Claim edge
 - (c) Assignment edge
 - (d) None of these
6. What is a Safe State and what is its use in deadlock avoidance?
7. Write steps to process the resource-request algorithm.

NOTES

8.5 DEADLOCK DETECTION

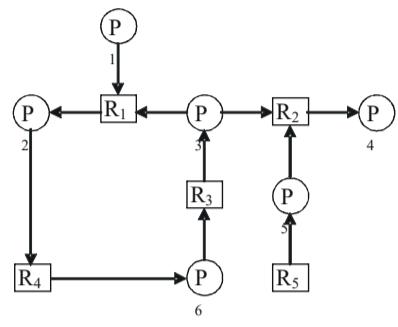
There is a possibility of deadlock if neither the deadlock prevention nor deadlock avoidance method is applied in a system. In such a situation, an algorithm must be provided for detecting the occurrence of deadlock in a system. Once the deadlock is detected, a methodology must be provided for the recovery of the system from the deadlock. In this section, we will discuss some of the ways by which deadlock can be detected.

Single Instance of Each Resource Type

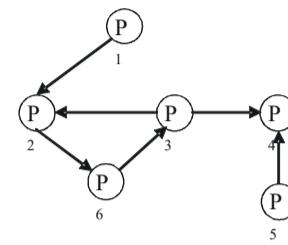
When only single resource of each type is available, the deadlock can be detected by using variation of resource allocation graph. In this variation, the nodes representing resources and corresponding edges are removed. This new variation of resource allocation graph is known as **wait-for graph**, which shows the dependency of a process on another process for the resource allocation. For example, an edge from the process P_i to P_j indicates that the process P_i is waiting for the process P_j to release the resources required by it. If there are two edges $P_n \rightarrow R_i$ and $R_i \rightarrow P_m$ in resource allocation graph, then the corresponding edge in the wait-for graph will be $P_n \rightarrow P_m$ indicating that the process P_n is waiting for the process P_m for the release of the resources. A resource allocation graph involving 6 processes and 5 resources is shown in Figure 8.7(a). The corresponding wait-for graph is shown in Figure 8.7(b).

Conditions for Deadlock

NOTES



(a) Resource Allocation Graph



(b) Wait-for Graph

Fig. 8.7 Converting Resource Allocation Graph to Wait-For Graph

If there exists a cycle in wait-for graph, there is a deadlock in the system, and the processes forming the part of cycle are blocked in the deadlock. In wait-for graph (refer Figure 8.6), the processes P₂, P₃ and P₆ form the cycle and hence are blocked in the deadlock. To take appropriate action to recover from this situation, an algorithm needs to be called periodically to detect existence of cycle in wait-for graph.

Multiple Instances of a Resource Type

When multiple instances of a resource type exist, the wait-for graph becomes inefficient to detect the deadlock in the system. For such system, another algorithm which uses certain data structures similar to the ones used in banker's algorithm is applied. The data structures used are as follows:

1. **Available Resources, A:** A vector of size q stores information about the number of available resources of each type.
2. **Current Allocation, C:** A matrix of order pxq stores information about the number of resources of each type allocated to each process. That is, C[i][j], indicates the number of resources of type j currently held by the process i.
3. **Request, Req:** A matrix of order pxq stores information about the number of resources of each type currently requested by each process. That is, R[i][j], indicates the number of resources of type j currently requested by the process i.

To understand the working of deadlock detection algorithm, consider a vector Complete of size p. Following are the steps to detect the deadlock.

1. Initialize Complete[i]=False for all i=1, 2, 3, ..., p. Complete[i]=False indicates that the ith process is still not completed.
2. Search for an i, such that Complete[i]=False and (Req<=A), that is, resources currently requested by this process is less than the available resources. If no such process exists, then go to Step 4.
3. Allocate the requested resources and let the process finish its execution and set Complete[i]=True for that process. Go to Step 2.

4. If `Complete[i]=False` for some i , then the system is in the state of deadlock and the i th process is deadlocked.

Conditions for Deadlock

8.6 DEADLOCK RECOVERY

NOTES

Once the system has detected deadlock in the system, a method is needed to recover the system from the deadlock and continue with the processing. Three different ways in which system can be recovered are—terminate one or more process to break the circular-wait condition, preempt the resources from the processes involved in the deadlock and roll back the processes to the previous checkpoint.

Terminating the Processes

There are two methods that can be used for terminating the processes to recover from the deadlock. These two methods are as follows:

- **Terminating one process at a time until the circular-wait condition is eliminated.** It involves an over head of invoking a deadlock detection algorithm after terminating each process to detect whether circular-wait condition is eliminated or not, that is, whether any processes are still deadlocked.
- **Terminating all processes involved in the deadlock.** This method will definitely ensure the recovery of a system from the deadlock. The disadvantage of this method is that many processes may have executed for a long time; close to their completion. As a result, the computations performed till the time of termination are discarded.

In both the cases, all the resources which were acquired by the processes being terminated are returned to the system. While terminating any process, it must be ensured that it does not leave any part of the system in an inconsistent state. For example, a process might be in the middle of updating a disk file and termination of such a process may leave that file in an inconsistent state. Similarly, a printer might be in the middle of printing some document. In this case when system is recovered from the deadlock, the system must reset the printer to a correct state.

In case of partial termination, while selecting the process to be terminated, the choice of processes must be such that it incurs minimum cost to the system. The factors which can effect the selection of a process for termination are as follows:

- Number of remaining resources required by it to complete its task.
- Number of processes required to be terminated.
- Number and type of resources held by the process.

NOTES

- Duration of time for which process has already been executed.
- Priority of the process.

Preempting the Resources

An alternative method to recover system from the state of deadlock is to preempt the resources from the processes one by one and allocate them to other processes until the circular-wait condition is eliminated. The steps involved in the preemption of resources from the process are as follows:

- 1. Select a Process for Preemption:** The choice of resources and processes must be such that they incur minimum cost to the system. All the factors mentioned earlier must be considered while making choice.
- 2. Roll Back of the Process:** After preempting the resources, the corresponding process must be rolled backed properly so that it does not leave the system in an inconsistent state. Since resources are preempted from the process, it cannot continue with the normal execution, hence must be brought to some safe state from where it can be restarted later. In case no such safe state can be achieved, the process must be totally rolled backed. However, partial rollback is always preferred over total rollback.
- 3. Prevent Starvation:** In case the selection of a process is based on the cost factor, it is quiet possible that same process is selected repeatedly for the rollback leading to the situation of starvation. This can be avoided by including the number of rollbacks of a given process in the cost factor.

Check Your Progress

8. What is the minimum number of resources required to ensure that deadlock?
9. What is the disadvantage of invoking the detection algorithm for every request?
10. State about deadlock detection.
11. How to prevent starvation?

8.7 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1.
 - Prevent the deadlock from occurring.
 - Adopt methods for avoiding the deadlock.
 - Allow the deadlock to occur, detect it and recover from it.
 - Ignore the deadlock.

2. A deadlock occurs when all the following four conditions are satisfied:

Conditions for Deadlock

- Mutual exclusion
- Hold and wait
- No pre-emption
- Circular wait

3. Waiting state.

4. In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution. We use Dijkstra's algorithm for deadlock avoidance.

5. B.

6. When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state. System is in safe state if there exists a safe sequence of all processes. Deadlock avoidance ensure that a system will never enter an unsafe state.

7. Once it is confirmed that system is in safe state, an algorithm called **resource-request** algorithm is used for determining whether the request by a process can be satisfied or not. To understand this algorithm, let Req be a matrix of the order $p \times q$, indicating the number of resources of each type requested by each process at any given point of time. That is, $\text{Req}[i][j]$ indicates the number of resources of j th type requested by the i th process at any given point of time. Following are the steps of this algorithm:

1. If $\text{Req}[i][j] \leq R[i][j]$, go to Step 2, otherwise an error occurs as process is requesting for more resources than the maximum number of resources required by it.
2. If $\text{Req}[i][j] \leq A[i][j]$, go to Step 3, otherwise the process P_i must wait until the required resources are available.
3. Allocate the resources are allocated and make the following changes in the data structures.

$$\begin{aligned}A &= A - \text{Req} \\C &= C + \text{Req} \\R &= R - \text{Req}\end{aligned}$$

8. To participate in a deadlock, a process must hold at least one resource. Further, a process cannot hold more than two resources so a process that currently has two resources can definitely run until it frees at least one of those resources.
9. Considerable overhead in computation time.

NOTES

NOTES

10. If resources have single instance, In this case for Deadlock detection we can run an algorithm to check for cycle in the Resource Allocation Graph. Presence of cycle in the graph is the sufficient condition for deadlock.
11. In case the selection of a process is based on the cost factor, it is quiet possible that same process is selected repeatedly for the rollback leading to the situation of starvation. This can be avoided by including the number of rollbacks of a given process in the cost factor.

8.8 SUMMARY

- A deadlock can be depicted with the help of a directed graph known as resource allocation graph.
- Deadlock prevention or deadlock avoidance techniques can be used to ensure that deadlocks never occur in a system. If any of these two techniques is not used, a deadlock may occur.
- The other method must be provided to either prevent the deadlock from occurrence or detect the deadlock and take an appropriate action if a deadlock has occurred.
- In case there are multiple instances of a resource type in a system, the deadlock cannot be avoided using resource allocation graph algorithm. An algorithm known as banker's algorithm is used in such a case.
- If each resource type has exactly one instance, cycle in resource allocation graph indicates a deadlock. If each resource type has several instances, cycle in resource allocation graph does not necessarily imply a deadlock.
- Deadlock prevention or deadlock avoidance techniques can be used to ensure that deadlocks never occur in a system.
- A deadlock can be prevented by not allowing all four conditions to be satisfied simultaneously, that is, by making sure that at least one of the four conditions does not hold.
- A deadlock occurs when all of the four conditions are satisfied at any point of time.
- The circular-wait condition can be eliminated by assigning a priority number to each available resource and a process can request resources only in increasing order.
- One of the simplest algorithms requires each process to declare the maximum number of resources (of each type) required by it during its course of execution.
- A deadlock can be avoided by never allowing allocation of a resource to a process if it leads to a deadlock. This can be achieved when some additional information is available about how the processes are going to request for resources in future.

NOTES

- A state is said to be safe if allocation of resources to processes does not lead to the deadlock. More precisely, a system is in safe state only if there is a safe sequence. A safe sequence is a sequence of process execution such that each and every process executes till its completion. If no such sequence of process execution exists then the state of the system is said to be unsafe.
- There is a possibility of deadlock if neither deadlock prevention nor deadlock avoidance method is applied in a system. In such a situation, an algorithm must be provided for detecting the occurrence of deadlock in a system.
- When only single resource of each type is available, the deadlock can be detected by using variation of resource allocation graph known as wait-for graph.
- When multiple instances of a resource type exist, the wait-for graph becomes inefficient in detecting the deadlock in the system. For such system, another algorithm which uses certain data structures similar to the ones used in banker's algorithm is applied.
- Whenever, the process requests for a resource, the claim edge is converted to request edge only if converting the corresponding request edge to assignment edge does not lead to the formation of a cycle in a graph, as cycle in a graph indicates the deadlock.
- There is a possibility of deadlock if neither the deadlock prevention nor deadlock avoidance method is applied in a system.
- Once the deadlock is detected, a methodology must be provided for the recovery of the system from the deadlock. Once it is confirmed that system is in safe state, an algorithm called resource request algorithm is used for determining whether the request by a process can be satisfied or not.
 - Terminating one process at a time until the circular-wait condition is eliminated.
 - Terminating all processes involved in the deadlock.
 - The three different ways in which system can be recovered are—terminate one or more process to break the circular-wait condition, pre-empt the resources from the processes involved in the deadlock and roll back the processes to the previous checkpoint.

8.9 KEY WORDS

- **Deadlock:** It occurs when every process in a set of processes is in a simultaneous wait state and each of them is waiting for the release of a resource held exclusively by one of the waiting processes in the set.
- **Resource Allocation Graph:** The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which

NOTES

are holding some resources or waiting for some resources. Vertices are mainly of two types, Resource and process.

- **Safe state:** A state in which the allocation of resources to processes does not lead to a deadlock; a system is in safe state only if there is a safe sequence.
- **Unsafe State:** If the system cannot fulfill the request of all processes then the state of the system is called unsafe.
- **Safe sequence:** A sequence of process execution such that each and every process executes till its completion. If no such sequence of process execution exists then the state of the system is said to be

8.10 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Question

1. Explain deadlock with an example.
2. How do you deal with deadlock?
3. How deadlock can be prevented? OR explain deadlock prevention.
4. State the process of eliminating circular-wait condition.
5. Describe the four necessary conditions for deadlock.
6. What do you understand by the terms ‘safe state’ and ‘unsafe state’?
7. Write the resource-request algorithm.
8. Explain the use of Banker’s algorithm for deadlock avoidance with illustration.
9. Define multiple instance of a resource type.

Long-Answer Questions

1. Describe four ways one might deal with deadlock with a brief description of each possibility and how it is achieved?
2. Consider a system having three instances of a resource type and two processes. Each process needs two resources to complete its execution. Can deadlock occur or not? Explain.
3. Explain the deadlock prevention techniques.
4. Describe the deadlock avoidance algorithm.
5. Consider a system is in an unsafe state. Illustrate how the processes can complete their execution without entering a deadlock state.
6. What is the significance of resource-request algorithm?
7. Describe the steps that the banker’s algorithm uses to determine whether to grant a request.

8. How to describe deadlock detected in details with the help of example?
9. What are deadlock recovery and describe two method of recovery for deadlock?
10. What are required steps to detect the deadlock?
11. Define the factors which effect the selection of a process for termination.

Conditions for Deadlock

NOTES

8.11 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.
- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.
- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.
- Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.
- Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.
- Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

UNIT 9 PROCESS SCHEDULING

NOTES

Structure

- 9.0 Introduction
 - 9.1 Objectives
 - 9.2 Concept of Process Scheduling
 - 9.3 Operations on Processes
 - 9.4 Scheduling Concepts
 - 9.5 Scheduling Criteria
 - 9.6 Scheduling Algorithms
 - 9.7 Answers to Check Your Progress Questions
 - 9.8 Summary
 - 9.9 Key Words
 - 9.10 Self Assessment Questions and Exercises
 - 9.11 Further Readings
-

9.0 INTRODUCTION

Scheduling is the method by which work is assigned to resources that complete the work. The work may be virtual computation elements such as threads, processes or data flows, which are in turn scheduled onto hardware resources such as processors, network links or expansion cards.

A scheduler is what carries out the scheduling activity. Schedulers are often implemented so they keep all computer resources busy (as in load balancing), allow multiple users to share system resources effectively, or to achieve a target quality of service. Scheduling is fundamental to computation itself, and an intrinsic part of the execution model of a computer system; the concept of scheduling makes it possible to have computer multitasking with a single Central Processing Unit (CPU).

Previously, there was a boundation's of loading only one program into the main memory for execution at a time. This program was very multifaceted and resourceful as it had access to all the hardware resources, such as memory, CPU time, I/O devices, and so on. With time improvements were accepted as new systems incorporate a variety of new and powerful features that dramatically improved the efficiency and functionality of your overall system. Modern computer systems corroborate multiprogramming, which allows a number of programs to reside in the main memory at the same time. These programs have the potential to run a number of programs simultaneously thereby requiring the system resources to be shared among them. Multiprogrammed systems need to distinguish among the multiple executing programs, which is accomplished with the concept of a process (also called task on some systems). A process is a program under execution

or a set of executable machine instructions. A process can be either a system process executing the system's code or a user process executing the user's code.

Scheduling disciplines are algorithms used for distributing resources among parties which simultaneously and asynchronously request them. Scheduling disciplines are used in routers (to handle packet traffic) as well as in operating systems (to share CPU time among both threads and processes), disk drives (I/O scheduling), printers (print spooler), most embedded systems, etc.

The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources. Scheduling deals with the problem of deciding which of the outstanding requests is to be allocated resources. There are many different scheduling algorithms.

In this unit, you will study about the basic concept of process scheduling, scheduling levels, preemptive vs nonpreemptive scheduling priorities, scheduling criteria and scheduling algorithms.

9.1 OBJECTIVES

After going through this unit, you will be able to:

- Introduce the basic concepts of processes
- Discuss about the various states of a process and the transition between these states
- Know about the various operations that can be performed on process
- Analyse the process scheduling
- Provide an overview of scheduling
- Discuss the criteria for scheduling
- Explain various scheduling algorithms
- Elaborate on the scheduling for multiprocessor systems

9.2 CONCEPT OF PROCESS SCHEDULING

The main objective of multiprogramming is to keep the jobs organized in such a manner that CPU has always one task to execute in the queue. This confirms that the CPU is utilized to the maximum level by reducing its idle time. This purpose can be very well achieved by keeping the CPU busy at all the times. However, when two or more processes compete for the CPU at the same time, then choice has to be made which process to allocate the CPU next. This procedure of determining the next process to be executed on the CPU is called **process scheduling** and the module of an operating system that makes this decision is called the **scheduler**.

NOTES

NOTES**Scheduling Queues**

For scheduling purposes, there exist different queues in the system that are as follows:

- **Job Queue:** Each process enters the system for execution. As soon as they enter the execution phase, they get into a queue called job queue (or **input queue**) on a mass storage device, such as hard disk.
- **Ready Queue:** From the job queue, the processes which are ready for the throughput are shifted into the main memory, where they are kept into a queue called ready queue. In other words, the ready queue contains all those processes that are waiting for the CPU.
- **Device Queue:** For each I/O device in the system, a separate queue called device queue is maintained. The process that needs to perform I/O during its execution is kept into the queue of that specific I/O device and waits there until it is served by the device.

Generally, both the ready queue and device queue are maintained as linked lists that contain PCBs of the processes in the queue as their nodes. Each PCB includes a pointer to the PCB of the next process in the queue (refer Figure 9.1). In addition, the header node of the queue contains pointers to the PCBs of the first and last process in the queue.

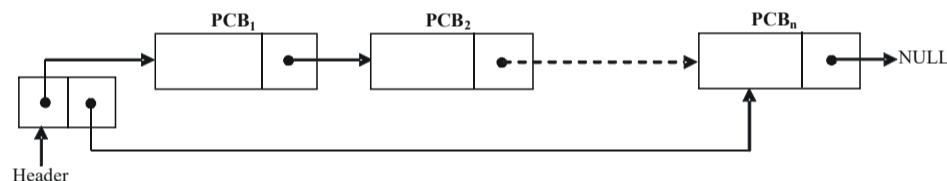
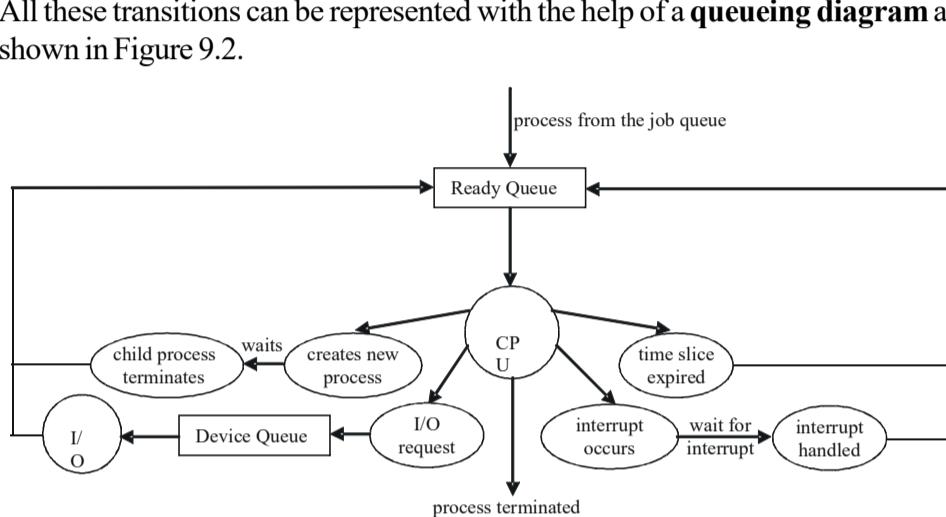


Fig. 9.1 Ready Queue and Device Queue Maintained as Linked List

Whenever, a process in the job queue is in the state of readiness and waiting to be executed, it is brought into the ready queue where it waits for the CPU allocation. Once CPU is allocated to it (that is, the process switches to the running state), the following transitions may happen.

- If the process needs to perform some I/O operation during its execution, it is removed from the ready queue and put into the appropriate device queue. After the process completes its I/O operation and is ready for the execution, it is switched from the device queue to ready queue.
- If an interrupt occurs, the CPU can be forcibly taken away from the currently executing process and the process has to wait until the interrupt is handled. After that, the process is put back into the ready queue.
- If the time slice (in case of time sharing systems) of the process has expired, the process is put back into the ready queue.
- If the process creates a new process and has to wait until the child process terminates, the parent process is suspended. After the execution of child process, it is again put back into the ready queue.

NOTES**Fig. 9.2 Queueing Diagram**

Note: In a single processor system, since there can be only one running process at a time, there is no need to maintain a queue for the running processes.

Types of Schedulers

The following types of schedulers (refer Figure 9.3) may coexist in a complex operating system.

- **Long-Term Scheduler:** It is also known as **job scheduler** or **admission scheduler**. It works with the job queue. It chooses the next process to be executed from the job queue and loads it into the main memory for execution. The long-term scheduler must select the processes in such a way that some of the processes are CPU-bound while others are I/O-bound. This is because if all the processes are CPU-bound, then for maximum number of times the other devices will remain idle and unused. On the contrary, if all the processes are I/O-bound, then CPU will remain idle most of the time. Thus, to make the best use of and optimize both, a balanced mix of CPU-bound and I/O-bound processes must be selected. The scheduler helps to control the degree of multiprogramming (that is, the number of processes in the ready queue) in order to achieve the maximum efficiency of the processor utilization at the desired level. For this, the long-term scheduler may admit new processes in the ready queue in case of poor processor utilization or may reduce the rate of admission of processes in the ready queue in case the processor utilization is high. By and large, the long-term scheduler is invoked only when a process exits from the system. Thus, the frequency of invocation of long-term scheduler entirely dependant on the system and workload and is much lower than other two types of schedulers.

NOTES

- **Short-Term Scheduler:** It is also known as **CPU scheduler or process scheduler** selects a process from the memory from the ready queue geared up for the execution and allocates CPU to it. This scheduler needs repeated invocation to carry out the execution as compared to the long-term scheduler, the primary reason being a process executes for a short period under normal working conditions and then it may have to wait either for I/O or some other reason. At that time, CPU scheduler is required to select some other process and allocate CPU to it. Thus, the CPU scheduler must be fast in order to provide the least time gap between execution procedure.
- **Medium-Term Scheduler:** It is also known as **swapper** comes into play whenever a process is to be removed from the ready queue (or from the CPU in case it is being executed) thereby reducing the degree of multiprogramming. This process is stored at some other fragment on the hard disk and later brought into the memory to restart the execution from the point where it left off. This task of temporarily switching a process in and out of main memory is known as **swapping**. The medium-term scheduler selects a process among the partially executed or unexecuted swapped-out processes and swaps it in the main memory. The medium-term scheduler is usually invoked when there is some unoccupied space in the memory made by the termination of a process or the supply of ready process reduces below a specified limit.

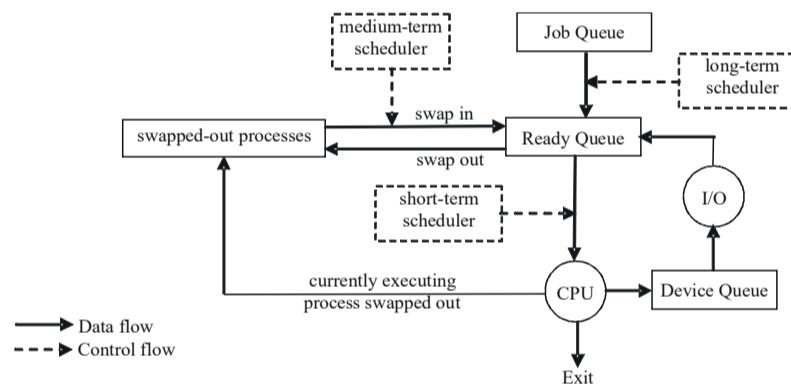


Fig. 9.3 Types of Schedulers

Context Switch

Transferring the control of CPU from one process to another demands for saving the context of the currently running process and loading the context of another ready process. This mechanism of saving and restoring the context of the ongoing process is known as **context switch**. The portion of the process control block including the process state, memory management information and CPU scheduling information together constitutes the **context** (also called **state information**) of a process. Context switch may occur due to varied reasons, some of which are as follows:

- The current process terminates and exits from the system.
- The time slice of the current process expires.
- The process has to wait for I/O or some other resource.

- Some higher priority process enters the system.
- The process relinquishes the CPU by invoking some system call.

Context switching involves a two step procedure, which are as follows:

1. **Save Context:** In this step, the kernel saves the context of the ongoing executing process in its PCB of the process so that the proficient restoration of the context can easily take place at some point in the future. This is only achieved when the processing phase of the ongoing process is successfully over and now the execution of the suspended process can be recommenced.
2. **Restore Context:** In this step, the kernel loads the saved context of a different process that is to be executed next in the line. Note that if the process to be executed is newly created and CPU has not yet been allocated to it, there will be no saved context. In this case, the kernel loads the context of the new process. However, if the process to be executed was in waiting state due to I/O or some other reason, there will be saved context that can be restored in future.

One of the major detriments of using context switching is that it incurs a heavy cost to the system in terms of real time and CPU cycles. Owing to the fact that during context switching the system does not perform any productive work. Therefore, context switching should be generally refrained from as far as possible otherwise it would amount to reckless use of time. Figure 9.4 shows context switching between two processes P_1 and P_2 .

Process Scheduling

NOTES

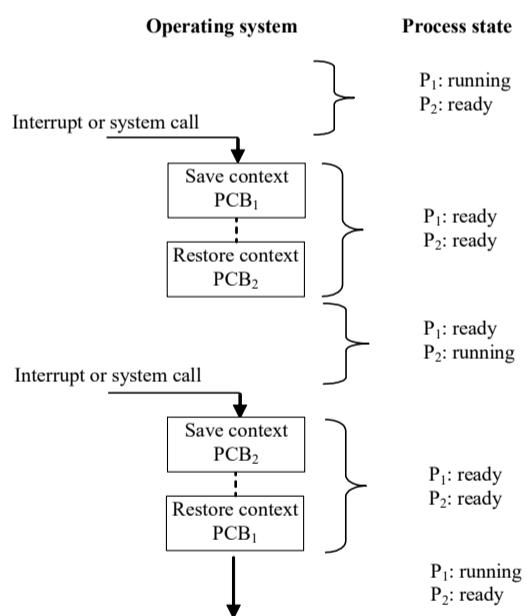


Fig. 9.4 Context Switching between Processes P_1 and P_2

9.3 OPERATIONS ON PROCESSES

There are innumerable operations that can be fulfilled on processes which encompasses creating, terminating, suspending, or resuming a process, and so on.

NOTES

To successfully execute these operations, the operating system provides run-time services (or system calls) for the process management. The user may invoke these system calls either directly by embedding the process supervisory calls in the user's program or indirectly by typing commands on the terminal which are translated by the system into system calls. In this section, we will discuss only about process creation and termination operations.

Process Creation

Whenever, an operating system is initialized on the system, a number of processes (system processes) are created automatically. Out of these, some involve user interaction (called **foreground processes**) while others are not related with any user but still perform some specific function (called **background processes**) to smooth completion of the booting task. In addition to system processes, new processes can be created afterward as well. Sometimes, a user process may need to create one or more processes during its execution. It can do the same by invoking the process creation system call (for example, `CreateProcess()` in Windows and `fork()` in UNIX) which tells the operating system to create a new process. This task of creating a new process on the request of some another process is called **process spawning**. The process that spawns a new process is called **parent** process whereas the spawned process is called the **child** process (or **sub** process). The newly created process can further create new processes thereby generating hierarchy of processes.

Whenever, a process creates a child process, there are chances of innumerable likelihoods that may arise depending on the operating system installed. Some of these likelihoods are as follows:

- Either the parent and child process may run concurrently (**asynchronous process creation**) or the parent process may wait until the child process completes its task and terminates (**synchronous process creation**).
- The newly created process may be the duplicate of the parent process in which case it contains a copy of the address space of its parent. On the other hand, the child process may have a new program loaded into its address space.
- The child process may be restricted to a subset of resources available to the parent process or the child process may obtain its resources directly from the operating system. In the former case, the resources being used by the parent process need to be divided or shared among its various child processes.

Note: Whenever, a process creates a new process, the PID of the child process is passed to the parent process.

Process Termination

Process Scheduling

Depending upon the condition, a process may be terminated either normally or forcibly by some another process. Normal termination occurs when the process successfully executes the assigned task and invokes an appropriate system call (for example, `ExitProcess()` in Windows and `exit()` in UNIX) to tell the operating system that it is finished. Consequently, all the resources held by the process are deallocated, the process returns output data (if any) to its parent and finally, the process is removed from the memory by deleting its PCB from the process table.

*Note: A process that no longer exists but still its PCB is not removed from the process table is known as a **zombie** process.*

Contrary to this, a process may cause abnormal termination of some another process. For this, the process invokes an appropriate system call (for example, `TerminateProcess()` in Windows and `kill()` in UNIX) that tells the operating system to kill some other process. Generally, the parent process can invoke such a system call to terminate its child process. As a general rule, this occurrence takes place because of the following reasons.

- Cascading termination in which the termination (whether normal or forced) of a process causes the termination of all its children. On some operating systems, a child process is not allowed to execute when its parent is being terminated. In such cases, the operating system initiates cascading termination.
- The task that was being performed by the child process is not required.
- The child process has used up the resources allocated to it more than that it was permitted.

NOTES

9.4 SCHEDULING CONCEPTS

Before we start discussing about the scheduling criteria and scheduling algorithms comprehensively, we will first take into account some comparatively important concepts of scheduling which are mentioned underneath.

Process Behaviour

The reaction and response of a process while it is in the execution phase highly influences CPU scheduling. Virtually, there is a continuous interchange of processes between CPU (for processing) and I/O devices (for performing I/O) during the period of execution. The time period elapsed in processing before performing the next I/O operation is known as **CPU burst** and the time period elapsed in performing I/O before the next CPU burst is known as **I/O burst**. Generally, the process execution starts with a CPU burst, followed by an I/O burst, then again by a CPU burst and so on until the process terminates. Thus, we can say the process execution

includes alternate cycles of CPU and I/O burst. Figure 9.5 shows the sequence of CPU and I/O bursts upon the execution of the following code segment written in C language.

NOTES

```
i=1;                                /* CPU burst
*/
sum=0;
scanf("%d", &num);           /* I/O burst */
while ( i <= 10 )             /* CPU burst
*/
{
    sum += num * i;
    i = i + 1;
```

Fig. 9.5 Alternate Cycles of CPU and I/O Bursts

The length of CPU burst and the I/O burst varies from process to process depending on whether the process is CPU-bound or I/O-bound. If the process is CPU-bound, it will have longer CPU bursts as compared to I/O bursts and vice versa in case the process is I/O-bound. From the scheduling perspective, only the length of CPU burst is taken into consideration and not the length of I/O burst.

When to Schedule

An important facet of scheduling is to determine under what circumstances the scheduler should be initiated to make scheduling decisions. The following circumstances may require the scheduler to make scheduling decisions.

- When a process switches from running to waiting state. Such a situation crops up, in case, the process has to wait for I/O or the termination of its child process or some another reason. In such situations, the scheduler has to select some ready process for execution.
- When a process switches from running to ready state due to occurrence of an interrupt. In such situation, the scheduler may decide to run a process from the ready queue. If the interrupt was caused by some I/O device that has now completed its task, the scheduler may choose the process that was blocked waiting for the I/O.
- When a process switches from waiting state to ready state, for example, in case, the process has completed its I/O operation. In such situation, the scheduler may select either the process that has now come into the ready state or the current process may be continued.
- When a process terminates and exits the system. In this case, the scheduler has to select a process for execution from the set of ready processes.

Dispatcher

The main aim of short-term scheduler is to opt for a process to be performed next on the CPU but it cannot allocate CPU to the selected process. The basic responsibility of setting up the execution of the selected process on the CPU is

performed by some other module of the operating system, called **dispatcher**. The dispatcher involves the following three steps to perform this function.

1. The dispatcher performs context switching that is switching the CPU to another process. The kernel saves the context of currently running process and restores the saved state of the process selected by the CPU scheduler. In case, the process selected by the short-term scheduler is new, the kernel loads its context.
2. The system switches from the kernel mode to user mode as a user process is to be executed.
3. The execution of the user process selected by the CPU scheduler is started by transferring the control either to the instruction that was supposed to be executed at the time the process was interrupted or to the first instruction if the process is going to be executed for the first time after its creation.

NOTES

Check Your Progress

1. Give the definition of process scheduling.
2. Explain about the term of context switch.
3. Write the definition of save context.
4. What is spawning?
5. Define the concept of scheduling?
6. How to work dispatcher in scheduling concept?
7. Elaborate the zombie process.

9.5 SCHEDULING CRITERIA

The scheduler must consider the following parameters and optimization criterias in order to maximize the performance of the system. Following mentioned are some of the criterias that help to judge the performance.

- **Fairness:** Generally, CPU is busy performing varied tasks. Fairness is defined as the degree to which each process is getting an equal chance to execute. The scheduler must ensure that each process should get a fair share of CPU time. However, it may treat different categories of processes (batch, real-time, or interactive) in a different manner.
- **CPU Utilization:** It is defined as the percentage of time the CPU is otherwise engaged in carrying out processes. To ensure better utilization, CPU must be kept as busy as possible, that is, there must be some process running everytime.
- **Balanced Utilization:** It is defined as the percentage of time all the system resources are busy. It considers not only the CPU utilization but the utilization

NOTES

of I/O devices, memory, and other resources also. To get more work done by the system, the CPU and I/O devices must be kept running simultaneously. For this, it is desirable to load a mixture of CPU-bound and I/O-bound processes in the memory.

- **Throughput:** It is defined as the total number of processes that a system can execute per unit of time. By and large, it depends on the average length of the processes to be executed. For the systems running long processes, throughput will be less as compared to the systems running short processes.
- **Turnaround Time:** It is defined as the relative amount of time that has rolled by from the time of initiation to the time of termination of a process. To put it differently, it is the difference between the time a process enters the system and the time it exits from the system. It includes all the time the process has spent waiting to enter into ready queue, within ready queue to get CPU, running on CPU, and in I/O queues. It is inversely proportional to throughput, that is, the more is the turnaround time, the less will be the throughput.
- **Waiting Time:** It is defined as the time used up by the process while waiting in the ready queue. However, it does not take into account the execution time or time consumed for I/O. In practice, waiting time is more accurately measured as compared to turnaround time.
- **Response Time:** It is defined as intervening time between the time the user initiates a request and the system starts responding to this request. For interactive systems, it is one of the best metric syetems employed to gauge the performance because in such systems, only the speed with which the system responds to user's request matters and not the time it takes to output the response.

The basic purpose of a CPU scheduling algorithm is that it should manage to make the best of fairness, CPU utilization, balanced utilization and throughput, and minimize turnaround, waiting and response time. Pratically speaking, no scheduling algorithm optimizes all the scheduling criteria. Thus the performance of an algorithm is evaluated on the basis of certain assumptions and average measures. For example, an algorithm that minimizes the average waiting time is considered as a good algorithm because this improves the overall efficiency of the program. However, in case of response time, minimizing the average is not a good criterion rather the variance in the response time of the processes should be minimized. This is because it is not desirable to have a process with long response time as compared to other processes.

9.6 SCHEDULING ALGORITHMS

The performance of any system depends almost as much on proper software setup as it does on your computer's hardware peripherals. To maximize the

NOTES

functioning, a broad range of algorithms are used for the CPU scheduling. These scheduling algorithms fall into two categories, namely, *non-preemptive* and *preemptive*.

- **Non-Preemptive Scheduling Algorithms:** Once the CPU is allocated to a process, it cannot be taken back until the process voluntarily releases it (in case the process has to wait for I/O or some other event) or the process terminates. In other words, we can say the decision to schedule a process is made only when the currently running process either switches to the waiting state or terminates. In both cases, the CPU executes some other process from the set of ready processes.
- **Preemptive Scheduling Algorithms:** The CPU can be forcibly taken back from the currently running process before its completion and allocated to some other process. The preempted process is put back in the ready queue and resumes its execution when it is scheduled again. Thus, a process may be scheduled many times before its completion. In preemptive scheduling, the decision to schedule another process is made whenever an interrupt occurs causing the currently running process to switch to ready state or a process having higher priority than the currently running process is ready to execute.

Note: A non-preemptive scheduling algorithm is also known as a *cooperative* or *voluntary* scheduling algorithm.

First-Come First-Served Scheduling

First-Come First-Serve (FCFS) is one of the simplest scheduling algorithms. As the name implies, the processes are executed in the order of their arrival in the ready queue, which means the process that enters the ready queue first gets the CPU first. To put it differently, it means the process that comes in first is managed first and the processes that come in subsequently wait in the queue until the first is completed. FCFS is a non-preemptive scheduling algorithm. Therefore, once the CPU is allocated to the process, it retains the control of CPU until it blocks or terminates.

To implement FCFS scheduling, the implementation of ready queue is managed as a FIFO (First-In First-Out) queue. When the first process enters the ready queue, it immediately gets the CPU and starts executing. Meanwhile, other processes enter the system and are added to the end of queue by inserting their PCBs in the queue. When the currently running process completes or blocks, the CPU is allocated to the process at the front of the queue and its PCB is removed from the queue. In case, a currently running process was blocked and later it comes to the ready state, its PCB is linked to the end of queue.

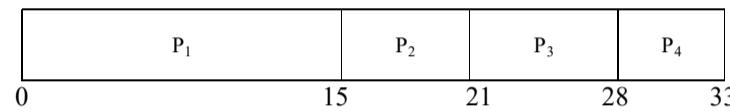
Example 9.1: Consider four processes P_1 , P_2 , P_3 , and P_4 with their arrival times and required CPU burst (in milliseconds) as shown in the following table.

NOTES

Process	P ₁	P ₂	P ₃	P ₄
Arrival time	0	2	3	5
CPU burst (ms)	15	6	7	5

How will these processes be scheduled according to FCFS scheduling algorithm? Compute the average waiting time and average turnaround time.

Solution: The processes will be scheduled as depicted in the following Gantt chart.



Initially, P₁ enters the ready queue at $t=0$ and CPU is allocated to it. While P₁ is executing, P₂, P₃ and P₄ enter the ready queue at $t=2$, $t=3$, and $t=5$, respectively. When P₁ completes, CPU is allocated to P₂ as it has entered before P₃ and P₄. When P₂ completes, P₃ gets the CPU after which P₄ gets the CPU.

Waiting time for P₁ = 0 ms as P₁ starts immediately

Waiting time for P₂ = $(15 - 2) = 13$ ms as P₂ enters at $t=2$ and starts at $t=15$

Waiting time for P₃ = $(21 - 3) = 18$ ms as P₃ enters at $t=3$ and starts at $t=21$

Waiting time for P₄ = $(28 - 5) = 23$ ms as P₄ enters at $t=5$ and starts at $t=28$

$$\text{Average waiting time} = (0 + 13 + 18 + 23)/4 = 13.5 \text{ ms}$$

Turnaround time for P₁ = $(15 - 0) = 15$ ms as P₁ enters at $t=0$ and exits at $t=15$

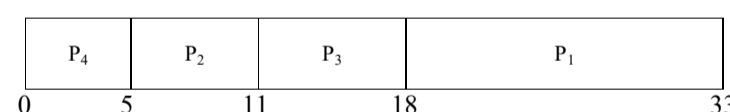
Turnaround time for P₂ = $(21 - 2) = 19$ ms as P₂ enters at $t=2$ and exits at $t=21$

Turnaround time for P₃ = $(28 - 3) = 25$ ms as P₃ enters at $t=3$ and exits at $t=28$

Turnaround time for P₄ = $(33 - 5) = 28$ ms as P₄ enters at $t=5$ and exits at $t=33$

$$\text{Average turnaround time} = (15 + 19 + 25 + 28)/4 = 21.75 \text{ ms}$$

The performance of FCFS scheduling algorithm largely depends on the order of arrival of processes in the ready queue. That is, whether the processes having long CPU burst enter before those having short CPU burst or vice versa. To illustrate this, assume that the processes (shown in Example 9.1) enter the ready queue in the order P₄, P₂, P₃ and P₁. Now, the processes will be scheduled as shown in the following Gantt chart.



$$\text{Average waiting time} = (0 + (5 - 2) + (11 - 3) + (18 - 5))/4 = 6 \text{ ms}$$

$$\text{Average turnaround time} = ((5 - 0) + (11 - 2) + (18 - 3) + (33 - 5))/4 = 14.25 \text{ ms}$$

It is clear that the average waiting and turnaround time may be cut down marginally if the processes having shorter CPU burst execute before those having longer CPU burst.

Process Scheduling

Advantages

- It is easier to understand and implement as processes are to be added at the end and removed from the front of queue. This process help to manipulate data effectively as the processes cannot be accessed from the middle of queue which means it always proceeds in sequentially.
- It is well suited for batch systems where the longer time periods for each process are often acceptable.

NOTES

Disadvantages

- The average waiting time is not minimal, that is, it varies from process to process. Therefore, this scheduling algorithm is never recommended where performance is a major issue.
- It reduces the CPU and I/O devices utilization under some circumstances. For example, assume that there is one long CPU-bound process and many short I/O-bound processes in the ready queue. Now, it may happen that while the CPU-bound process is executing, the I/O-bound processes complete their I/O and come to the ready queue for execution. There they have to wait for the CPU-bound process to release the CPU and the I/O devices also remain idle during this time. When the CPU-bound process needs to perform I/O, it comes to the device queue and the CPU is allocated to I/O-bound processes. As the I/O-bound processes require a little CPU burst, they execute quickly and come back to the device queue thereby leaving the CPU idle. Then the CPU-bound process enters the ready queue and is allocated the CPU which again makes the I/O processes waiting in ready queue at some point of time. This process takes place repeatedly until the CPU-bound process is done which results in low CPU and I/O devices utilization.
- It is not suitable for time sharing systems where it is desirable each process should get the equal amount of CPU time.

Shortest Job First Scheduling

Shortest Job First (SJF), also known as **Shortest Process Next (SPN)** or **Shortest Request Next (SRN)**, is a non-preemptive scheduling algorithm that schedules the processes according to the length of CPU burst they require. At any point of time, among all the ready processes waiting in the queue to be executed, the one having the shortest CPU burst is scheduled first. In other words, the process with the shortest execution is selected for the execution. Thus, a longer process has to wait until all the processes shorter than it have been executed. In case two processes have the same CPU burst, they are scheduled in the FCFS order.

*Self-Instructional
Material*

173

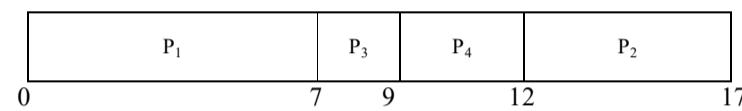
NOTES

Example 9.2: Consider four processes P_1 , P_2 , P_3 and P_4 with their arrival times and required CPU burst (in milliseconds) as shown in the following table.

Process	P_1	P_2	P_3	P_4
Arrival time	0	1	3	4
CPU burst (ms)	7	5	2	3

How will these processes be scheduled according to SJF scheduling algorithm? Compute the average waiting time and average turnaround time.

Solution: The processes will be scheduled as depicted in the following.



Initially, P_1 enters the ready queue at $t=0$ and gets the CPU as there are no other processes in the queue. While it is executing, P_2 , P_3 and P_4 enter the queue at $t=1$, $t=3$ and $t=4$, respectively. When CPU becomes free, that is, at $t=7$, it is allocated to P_3 because it is having the shortest CPU burst among the three processes. When P_3 completes, CPU is allocated first to P_4 and then to P_2 .

Waiting time for $P_1 = 0$ ms as P_1 starts immediately

Waiting time for $P_2 = (12 - 1) = 11$ ms as P_2 enters at $t=1$ and starts at $t=12$

Waiting time for $P_3 = (7 - 3) = 4$ ms as P_3 enters at $t=3$ and starts at $t=7$

Waiting time for $P_4 = (9 - 4) = 5$ ms as P_4 enters at $t=4$ and starts at $t=9$

$$\text{Average waiting time} = (0 + 11 + 4 + 5)/4 = 5 \text{ ms}$$

Turnaround time for $P_1 = (7 - 0) = 7$ ms as P_1 enters at $t=0$ and exits at $t=7$

Turnaround time for $P_2 = (17 - 1) = 16$ ms as P_2 enters at $t=1$ and exits at $t=17$

Turnaround time for $P_3 = (9 - 3) = 6$ ms as P_3 enters at $t=3$ and exits at $t=9$

Turnaround time for $P_4 = (12 - 4) = 8$ ms as P_4 enters at $t=4$ and exits at $t=12$

$$\text{Average turnaround time} = (7 + 16 + 6 + 8)/4 = 9.25 \text{ ms}$$

Advantages

- It eliminates the variance in waiting and turnaround times., It is incontrovertibly superlative with respect to average waiting time if all processes are available at the same time due to the fact that short processes are given the highest priority in comparison to the long ones which results in immense decrement of the waiting time for short processes. Accordingly, the waiting time for long processes is increased. However, the diminution in waiting time is relatively more than the increment in waiting time and thus, the average waiting time decreases.

Disadvantages

- It is quite intricate to implement because it needs to be familiarized with the exact length of CPU burst of processes in advance. Realistically, it is quite

NOTES

impractical to have the prior knowledge of required processing time of processes. Many systems expect users to provide estimates of CPU burst of processes where there are maximum feasibilities of erroneous ones.

- It favours the processes having short CPU burst because as long as the short processes continue to enter the ready queue, the long processes are not allowed to get the CPU. This results in **starvation** of long processes.

Shortest Remaining Time Next Scheduling

The Shortest Remaining Time Next (SRTN) also known as **Shortest Time To Go (STG)**, is a preemptive version of the SJF scheduling algorithm. It takes into account the length of remaining CPU burst of the processes rather than the whole length in order to schedule them. Here also the scheduler always chooses the process for execution that has the shortest remaining processing time. While a process is being executed, the CPU can be taken back from it and assigned to some newly arrived process if the CPU burst of the new process is shorter than its remaining CPU burst. Notice that if at any point of time, the remaining CPU burst of two processes becomes equal; they are scheduled in the FCFS order.

Example 9.3: Consider the same set of processes, their arrival times and CPU burst as shown in Example 9.2. How will these processes be scheduled according to SRTN scheduling algorithm? Compute the average waiting time and average turnaround time.

Solution: The processes will be scheduled as depicted in the following Gantt chart.

P ₁	P ₂	P ₃	P ₂	P ₄	P ₁	
0	1	3	5	8	11	17

Initially, P₁ enters the ready queue at $t=0$ and gets the CPU as there are no other processes in the queue. While it is executing, at time $t=1$, P₂ with CPU burst of 5 ms enters the queue. At that time the remaining CPU burst of P₁ is 6 ms which is greater than that of P₂. Therefore, the CPU is taken back from P₁ and allocated to P₂. During execution of P₂, P₃ enters at $t=3$ with a CPU burst of 2 ms. Again CPU is switched from P₂ to P₃ as the remaining CPU burst of P₂ at $t=3$ is 3 ms which is greater than that of P₃. However, when at time $t=4$, P₄ with CPU burst of 3 ms enters the queue, the CPU is not assigned to it because at that time the remaining CPU burst of currently running process (that is, P₃) is 1 ms which is shorter than that of P₄. When P₃ completes, there are three processes P₁ (6 ms), P₂ (3 ms) and P₄ (3 ms) in the queue. To break the tie between P₂ and P₄, the scheduler takes into consideration their arrival order and the CPU is allocated first to P₂, then to P₄ and finally, to P₁.

Waiting time for P₁ = $(11 - 1) = 10$ ms as P₁ enters at $t=0$, executes for 1 ms, preempts at $t=1$, and then resumes at $t=11$.

NOTES

Waiting time for $P_2 = (5 - 2 - 1) = 2$ ms as P_2 enters at $t = 1$, executes for 2 ms, preempts at $t = 3$, and then resumes at $t = 5$.

Waiting time for $P_3 = 0$ ms as P_3 enters at $t = 3$, starts immediately and executes completely.

Waiting time for $P_4 = (8 - 4) = 4$ ms as P_4 enters at $t = 4$, starts at $t = 8$ and executes completely.

$$\text{Average waiting time} = (10 + 2 + 0 + 4)/4 = 4 \text{ ms}$$

Turnaround time for $P_1 = (17 - 0) = 17$ ms as P_1 enters at $t = 0$ and exits at $t = 17$

Turnaround time for $P_2 = (8 - 1) = 7$ ms as P_2 enters at $t = 1$ and exits at $t = 8$

Turnaround time for $P_3 = (5 - 3) = 2$ ms as P_3 enters at $t = 3$ and exits at $t = 5$

Turnaround time for $P_4 = (11 - 4) = 7$ ms as P_4 enters at $t = 4$ and exits at $t = 11$

$$\text{Average turnaround time} = (17 + 7 + 2 + 7)/4 = 8.25 \text{ ms}$$

Advantages

- A long process that is near to its completion may be favoured over the short processes entering the system. This results in an improvement in the turnaround time of the long process.

Disadvantages

- Like SJF, it also requires an estimate of the next CPU burst of a process in advance.
- Favouring a long process nearing its completion over the several short processes entering the system may impact the turnaround times of short processes waiting in the queue.
- It favours only those long processes that are just about to complete and not those who have just started their operation. Thus, starvation of long processes still may occur.

Priority-Based Scheduling

As the name implies, in priority-based scheduling algorithm, each process is manually assigned a priority and the process that is assigned the higher priority process is scheduled before the lower priority process. At any point of time, the process having the highest priority among all the ready processes is scheduled first. In case, two processes are having the same priority, they are executed in the FCFS order.

The priority scheduling may be either preemptive or non-preemptive. The choice is made whenever a new process enters the ready queue while some process is executing. If the newly arrived process has the higher priority than the currently running process, the preemptive priority scheduling algorithm preempts the currently running process and allocates CPU to the new process. On the other hand, the non-preemptive scheduling algorithm allows the currently running process to complete its execution and the new process has to wait for the CPU.

Note: Both SJF and SRTN are special cases of priority-based scheduling where priority of a process is equal to inverse of the next CPU burst. Lower is the CPU burst, higher will be the priority.

Process Scheduling

A major design issue related with priority scheduling is how to compute priorities of the processes. The priority can be assigned to a process either internally defined by the system depending on the process's characteristics like memory usage, I/O frequency, usage cost, and so on, or externally defined by the user executing that process.

NOTES

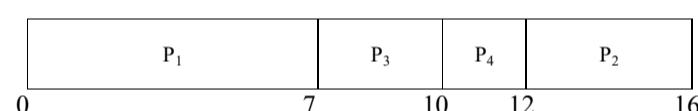
Example 9.4: Consider four processes P_1 , P_2 , P_3 , and P_4 with their arrival times, required CPU burst (in milliseconds), and priorities as shown in the following table.

Process	P_1	P_2	P_3	P_4
Arrival time	0	1	3	4
CPU burst (ms)	7	4	3	2
Priority	4	3	1	2

Assuming that the lower priority number means the higher priority, how will these processes be scheduled according to non-preemptive as well as preemptive priority scheduling algorithm? Compute the average waiting time and average turnaround time in both cases.

Solution: Non-Preemptive Priority Scheduling Algorithm

The processes will be scheduled as depicted in the following Gantt chart.



Initially, P_1 enters the ready queue at $t=0$ and gets the CPU as there are no other processes in the queue. While it is executing, P_2 , P_3 , and P_4 enter the queue at $t=1$, $t=3$, and $t=4$, respectively. When CPU becomes free, that is, at $t=7$, it is allocated to P_3 because it is having the highest priority (that is, 1) among the three processes. When P_3 completes, CPU is allocated to the next lower priority process, that is, P_4 and finally, the lowest priority process P_2 is executed.

Waiting time for $P_1 = 0$ ms as P_1 starts immediately

Waiting time for $P_2 = (12 - 1) = 11$ ms as P_2 enters at $t=1$ and starts at $t=12$

Waiting time for $P_3 = (7 - 3) = 4$ ms as P_3 enters at $t=3$ and starts at $t=7$

Waiting time for $P_4 = (10 - 4) = 6$ ms as P_4 enters at $t=4$ and starts at $t=10$

Average waiting time = $(0 + 11 + 4 + 6)/4 = 5.25$ ms

Turnaround time for $P_1 = (7 - 0) = 7$ ms as P_1 enters at $t=0$ and exits at $t=7$

Turnaround time for $P_2 = (16 - 1) = 15$ ms as P_2 enters at $t=1$ and exits at $t=16$

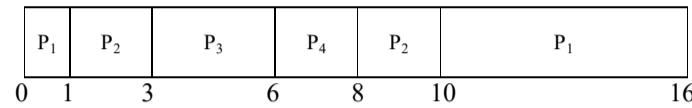
Turnaround time for $P_3 = (10 - 3) = 7$ ms as P_3 enters at $t=3$ and exits at $t=10$

Turnaround time for $P_4 = (12 - 4) = 8$ ms as P_4 enters at $t=4$ and exits at $t=12$

Average turnaround time = $(7 + 15 + 7 + 8)/4 = 9.25$ ms

NOTES**Preemptive Priority Scheduling Algorithm**

The processes will be scheduled as depicted in the following Gantt chart.



Initially, P₁ of priority 4 enters the ready queue at $t=0$ and gets the CPU as there are no other processes in the queue. While it is executing, at time $t=1$, P₂ of priority 3 greater than that of currently running process P₁, enters the queue. Therefore, P₁ is preempted (with remaining CPU burst of 6 ms) and the CPU is allocated to P₂. During execution of P₂, P₃ of priority 1 enters at $t=3$. Again CPU is switched from P₂ (with remaining CPU burst of 2 ms) to P₃ as the priority of P₃ is greater than that of P₂. However, when at time $t=4$, P₄ of priority 2 enters the queue, the CPU is not assigned to it because it has lower priority than currently running process P₃. When P₃ completes, there are three processes P₁, P₂, and P₄ in the ready queue having priorities 4, 3, and 2, respectively. The CPU is allocated first to P₄, then to P₂ and finally to P₁.

Waiting time for P₁ = $(10 - 1) = 9$ ms as P₁ enters at $t=0$, executes for 1 ms, preempts at $t=1$ and then resumes at $t=11$.

Waiting time for P₂ = $(8 - 2 - 1) = 5$ ms as P₂ enters at $t=1$, executes for 2 ms, preempts at $t=3$ and then resumes at $t=8$.

Waiting time for P₃ = 0 ms as P₃ enters at $t=3$, starts immediately and executes completely.

Waiting time for P₄ = $(6 - 4) = 2$ ms as P₄ enters at $t=4$, starts at $t=6$ and executes completely.

$$\text{Average waiting time} = (9 + 5 + 0 + 2)/4 = 4 \text{ ms}$$

Turnaround time for P₁ = $(16 - 0) = 16$ ms as P₁ enters at $t=0$ and exits at $t=16$

Turnaround time for P₂ = $(10 - 1) = 9$ ms as P₂ enters at $t=1$ and exits at $t=10$

Turnaround time for P₃ = $(6 - 3) = 3$ ms as P₃ enters at $t=3$ and exits at $t=6$

Turnaround time for P₄ = $(8 - 4) = 4$ ms as P₄ enters at $t=4$ and exits at $t=8$

$$\text{Average turnaround time} = (16 + 9 + 3 + 4)/4 = 8 \text{ ms}$$

Advantages

- Important processes are always executed first without letting them to wait in the queue because of the least priority execution of less important processes.

Disadvantages

- It suffers from the problem of starvation of lower priority processes, because of the constant arrival of higher priority processes. It seldom gives chance to lower priority processes to acquire the CPU. One possible solution to this problem is **aging** which is a process of gradually increasing the priority

NOTES

of a low priority process with increase in its waiting time. If the priority of a low priority process is increased after each fixed time of interval, it is ensured that at some time it will become a highest priority process and will ultimately get executed.

Highest Response Ratio Next Scheduling

The Highest Response Ratio Next (HRN) scheduling is a non-preemptive scheduling algorithm that schedules the processes based on their response ratio. Whenever, CPU becomes available, the process having the highest value of response ratio among all the ready processes is scheduled next. The Response Ratio (RR) of a process in the queue is computed by using the following equation.

$$\text{Response Ratio (RR)} = \frac{\text{Time since arrival} + \text{CPU burst}}{\text{CPU burst}}$$

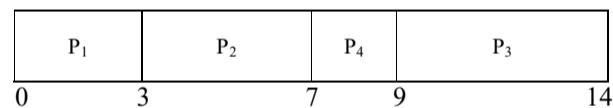
Initially, when a process enters, its response ratio is 1. It goes on increasing at the rate of $(1/\text{CPU burst})$ as the process's waiting time increases.

Example 9.5: Consider four processes P_1 , P_2 , P_3 , and P_4 with their arrival times and required CPU burst (in milliseconds) as shown in the following table.

Process	P_1	P_2	P_3	P_4
Arrival time	0	2	3	4
CPU burst (ms)	3	4	5	2

How will these processes be scheduled according to HRN scheduling algorithm? Compute the average waiting time and average turnaround time.

Solution: The processes will be scheduled as depicted in the following Gantt chart.



Initially, P_1 enters the ready queue at $t=0$ and CPU is allocated to it. By the time P_1 completes, P_2 and P_3 have arrived at $t=2$ and $t=3$, respectively. At $t=3$, the response ratio of P_2 is $((3-2)+4)/4 = 1.25$ and of P_3 is 1 as it has just arrived. Therefore P_2 is scheduled next. During execution of P_2 , P_4 enters the queue at $t=4$. When P_2 completes at $t=7$, the response ratio of P_3 is $((7-3)+5)/5 = 1.8$ and of P_4 is $((7-4)+2)/2 = 2.5$. As P_4 has higher response ratio, the CPU is allocated to it and after its completion, P_3 is executed.

Waiting time for $P_1 = 0$ ms as P_1 starts immediately

Waiting time for $P_2 = (3-2) = 1$ ms as P_2 enters at $t=2$ and starts at $t=3$

Waiting time for $P_3 = (9-3) = 6$ ms as P_3 enters at $t=3$ and starts at $t=9$

Waiting time for $P_4 = (7-4) = 3$ ms as P_4 enters at $t=4$ and starts at $t=7$

Average waiting time = $(0 + 1 + 6 + 3)/4 = 2.5$ ms

NOTES

Turnaround time for $P_1 = (3 - 0) = 3$ ms as P_1 enters at $t = 0$ and exits at $t = 3$

Turnaround time for $P_2 = (7 - 2) = 5$ ms as P_2 enters at $t = 2$ and exits at $t = 7$

Turnaround time for $P_3 = (14 - 3) = 11$ ms as P_3 enters at $t = 3$ and exits at $t = 14$

Turnaround time for $P_4 = (9 - 4) = 5$ ms as P_4 enters at $t = 4$ and exits at $t = 9$

Average turnaround time = $(3 + 5 + 11 + 5)/4 = 6$ ms

Advantages

- It favours short processes because with increase in waiting time, the response ratio of short processes increases speedily as compared to long processes. Thus, they are scheduled earlier than long processes.
- Unlike SJF, starvation does not occur since with increase in waiting time, the response ratio of long processes also increases and eventually they are scheduled.

Disadvantages

- Like SJF and SRTN, it also requires an estimate of the expected service time (CPU burst) of a process.

Round Robin Scheduling

Round Robin (RR) scheduling is one of the most widely used preemptive scheduling algorithms which considers all the processes as equally important and treats them in a favourable manner. Each process in the ready queue gets a fixed amount of CPU time (generally from 10 to 100 ms) known as **time slice or time quantum** for its execution. If the process does not execute completely till the end of time slice, it is preempted and the CPU is allocated to the next process in the ready queue. However, if the process blocks or terminates before the time slice expires, the CPU is switched to the next process in the ready queue at that moment only.

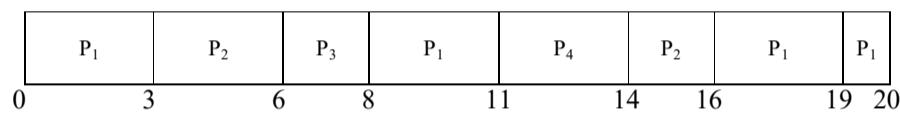
To implement the round robin scheduling algorithm, the ready queue is treated as a circular queue. All the processes arriving in the ready queue are put at the end of queue. The CPU is allocated to the first process in the queue, and the process executes until its time slice expires. If the CPU burst of the process being executed is less than one time quantum, the process itself releases the CPU and is deleted from the queue. The CPU is then allocated to the next process in the queue. However, if the process does not execute completely within the time slice, an interrupt occurs when the time slice expires. The currently running process is preempted, put back at the end of the queue and the CPU is allocated to the next process in the queue. The preempted process again gets the CPU after all the processes before it in the queue have been allocated their CPU time slice. The whole process continues until all the processes in queue have been executed.

Example 9.6: Consider four processes P_1 , P_2 , P_3 and P_4 with their arrival times and required CPU burst (in milliseconds) as shown in the following table.

Process	P_1	P_2	P_3	P_4
Arrival time	0	1	3	4
CPU burst (ms)	10	5	2	3

Assuming that the time slice is 3 ms, how will these processes be scheduled according to round robin scheduling algorithm? Compute the average waiting time and average turnaround time.

Solution: The processes will be scheduled as depicted in the following Gantt chart.



Initially, P_1 enters the ready queue at $t = 0$ and gets the CPU for 3 ms. While it executes, P_2 and P_3 enter the queue at $t = 1$ and $t = 3$, respectively. Since, P_1 does not execute within 3 ms, an interrupt occurs when the time slice gets over. P_1 is preempted (with remaining CPU burst of 7 ms), put back in the queue after P_3 because P_4 has not entered yet and the CPU is allocated to P_2 . During execution of P_2 , P_4 enters in the queue at $t = 4$ and put at the end of queue after P_1 . When P_2 times out, it is preempted (with remaining CPU burst of 2 ms) and put back at the end of queue after P_4 . The CPU is allocated to the next process in the queue, that is, to P_3 and it executes completely before the time slice expires. Thus, the CPU is allocated to the next process in the queue which is P_1 . P_1 again executes for 3 ms, then preempted (with remaining CPU burst of 4 ms) and put back at the end of the queue after P_2 and the CPU is allocated to P_4 . P_4 executes completely within the time slice and the CPU is allocated to next process in the queue, that is, P_2 . As P_2 completes before the time out occurs, the CPU is switched to P_1 at $t = 16$ for another 3 ms. When the time slice expires, CPU is again allocated to P_1 as it is the only process in the queue.

Waiting time for $P_1 = (5 + 5) = 10$ ms as P_1 enters at $t = 0$, starts immediately, waits for $t = 3-8$ and then again waits for $t = 11-16$

Waiting time for $P_2 = (3 - 1 + 8) = 10$ ms as P_2 enters at $t = 1$, starts at $t = 3$, waits for $t = 6-14$ and then resumes at $t = 14$

Waiting time for $P_3 = (6 - 3) = 3$ ms as P_3 enters at $t = 3$, starts at $t = 6$ and executes completely

Waiting time for $P_4 = (11 - 4) = 7$ ms as P_4 enters at $t = 4$, starts at $t = 11$ and executes completely

$$\text{Average waiting time} = (10 + 10 + 3 + 7)/4 = 7.5 \text{ ms}$$

Turnaround time for $P_1 = (20 - 0) = 20$ ms as P_1 enters at $t = 0$ and exits at $t = 20$

Turnaround time for $P_2 = (16 - 1) = 15$ ms as P_2 enters at $t = 1$ and exits at $t = 16$

NOTES

NOTES

Turnaround time for $P_3 = (8 - 3) = 5$ ms as P_3 enters at $t = 3$ and exits at $t = 8$

Turnaround time for $P_4 = (14 - 4) = 10$ ms as P_4 enters at $t = 4$ and exits at $t = 14$

Average turnaround time = $(20 + 15 + 5 + 10)/4 = 12.5$ ms

The performance of round robin scheduling is greatly affected by the size of the time quantum. If the time quantum is too small, a number of context switches occur which in turn increase the system overhead. The more time will be spent in performing context switching rather than executing the processes. On the other hand, if the time quantum is too large, the performance of round robin simply degrades to FCFS.

Note: If the time quantum is too small, say 1 ms, the round robin scheduling is called **processor sharing**.

Advantages

- It is efficient for time sharing systems where the CPU time is divided among the competing processes.
- It increases the fairness among the processes.

Disadvantages

- The processes (even the short processes) may take a long time to execute. This decreases the system throughput.
- It requires some extra hardware support such as a timer to cause interrupt after each time out.

Note: Ideally, the size of time quantum should be such that 80% of the processes could complete their execution within one time quantum.

Multilevel Queue Scheduling

The multilevel queue scheduling is designed especially for the environments where the processes can be categorized into different groups on the basis of their different response time requirements or different scheduling needs. One possible categorization may be based on whether the process is a system process, batch process or an interactive process (refer Figure 9.6). Each group of processes is associated with a specific priority. The system processes, for example, may have the highest priority whereas the batch processes may have the least priority.

To implement multilevel scheduling algorithm, the ready queue is split up into as many separate queues as there are groups. Whenever, a new process enters, it is assigned permanently to one of the ready queues depending on its properties like memory requirements, type and priority. Each ready queue has its own scheduling algorithm. For example, for batch processes, FCFS scheduling algorithm may be used, and for interactive processes, one may use the round robin scheduling algorithm. In addition, the processes in higher priority queues are executed before those in lower priority queues. This implies no batch process can

run unless all the system processes and interactive processes have been executed completely. Moreover, if a process enters into a higher priority queue while a process in lower priority queue is executing, then the lower priority process would be preempted in order to allocate the CPU to the higher priority process.

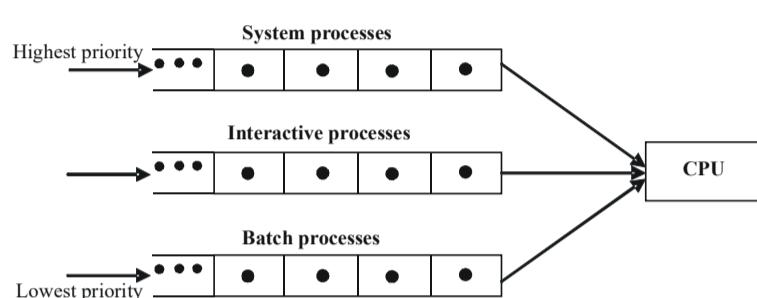


Fig. 9.6 Multilevel Queue Scheduling

NOTES

Advantages

- Processes are permanently assigned to their respective queues and do not move between queues. This results in low scheduling overhead.

Disadvantages

- The processes in lower priority queues may have to starve for CPU in case processes are continuously arriving in higher priority queues. One possible way to prevent starvation is to time slice among the queues. Each queue gets a certain share of CPU time which it schedules among the processes in it. Note that the time slice of different priority queues may differ.

Multilevel Feedback Queue Scheduling

The multilevel feedback queue scheduling also known as **multilevel adaptive scheduling** is an improved version of multilevel queue scheduling algorithm. In this scheduling algorithm, processes are not permanently assigned to queues; instead they are allowed to move between the queues. The decision to move a process between queues is based on the time taken by it in execution so far and its waiting time. If a process uses too much CPU time, it is moved to a lower priority queue. Similarly, a process that has been waiting for too long in a lower priority queue is moved to a higher priority queue in order to avoid starvation.

To understand this algorithm, consider a multilevel feedback queue scheduler (refer Figure 9.7) with three queues, namely, Q_1 , Q_2 and Q_3 . Further, assume that the queues Q_1 and Q_2 employ round robin scheduling algorithm with time quantum of 5 ms and 10 ms, respectively while in queue Q_3 , the processes are scheduled in FCFS order. The scheduler first executes all processes in Q_1 . When Q_1 is empty, the scheduler executes the processes in Q_2 . Finally, when both Q_1 and Q_2 are empty, the processes in Q_3 are executed. While executing processes in Q_2 , if a new process arrives in Q_1 , the currently executing process is preempted and the new process starts executing. Similarly, a process arriving in Q_2 preempts a process executing in Q_3 . Initially, when a process enters into ready queue; it is placed in Q_1 where it is allocated the CPU for 5 ms. If the process finishes its execution within

NOTES

5 ms, it exits from the queue. Otherwise, it is preempted and placed at the end of Q_2 . Here, it is allocated the CPU for 10 ms (if Q_1 is empty) and still if it does not finish, it is preempted and placed at the end of Q_3 .

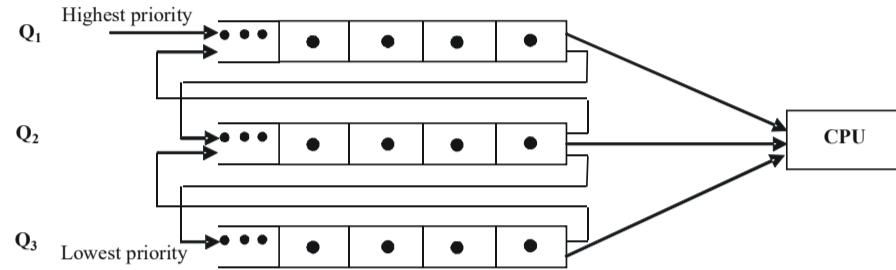


Fig. 9.7 Multilevel Feedback Queue Scheduling

Example 9.7: Consider four processes P_1 , P_2 , P_3 and P_4 with their arrival times and required CPU burst (in milliseconds) as shown in the following table.

Process	P_1	P_2	P_3	P_4
Arrival time	0	12	25	32
CPU burst (ms)	25	18	4	10

Assume that there are three ready queues Q_1 , Q_2 and Q_3 . The CPU time slice for Q_1 and Q_2 is 5 ms and 10 ms, respectively and in Q_3 , processes are scheduled on FCFS basis. How will these processes be scheduled according to multilevel feedback queue scheduling algorithm? Compute the average waiting time and average turnaround time.

Solution: The processes will be scheduled as depicted in the following Gantt chart.

P_1 Q_1	P_1 Q_2	P_2 Q_1	P_1 Q_2	P_3 Q_1	P_2 Q_2	P_4 Q_1	P_1 Q_2	P_2 Q_2	P_4 Q_2	
0	5	12	17	25	29	32	37	42	52	57

Initially, P_1 enters the system at $t=0$, placed in Q_1 and allocated the CPU for 5 ms. Since, it does not execute completely, it is moved to Q_2 at $t=5$. Now Q_1 is empty so the scheduler picks up the process from the head of Q_2 . Since, P_1 is the only process in Q_2 , it is again allocated the CPU for 10 ms. But during its execution, P_2 enters Q_1 at $t=12$, therefore P_1 is preempted and P_2 starts executing. At $t=17$, P_2 is moved to Q_2 and placed after P_1 . The CPU is allocated to the first process in Q_2 , that is, P_1 . While P_1 is executing, P_3 enters Q_1 at $t=25$ so P_1 is preempted, placed after P_2 in Q_2 and P_3 starts executing. As P_3 executes completely within time slice, the scheduler picks up the first process in Q_2 which is P_2 at $t=29$. While P_2 is executing, P_4 enters Q_1 at $t=32$ because of which P_2 is preempted and placed after P_1 in Q_2 . The CPU is assigned to P_4 for 5 ms and at $t=37$, P_4 is moved to Q_2 and placed after P_2 . At the same time, the CPU is allocated to P_1 .

NOTES

(first process in Q_2). When it completes at $t=42$, the next process in Q_2 which is P_2 , starts executing. When it completes, the last process in Q_2 , that is, P_4 is executed.

Waiting time for $P_1 = (5 + 12) = 17$ ms as P_1 first waits for $t=12-17$ and then again waits for $t=25-37$

Waiting time for $P_2 = (12+10) = 22$ ms as P_2 first waits for $t=17-29$ and then again waits for $t=32-42$

Waiting time for $P_3 = 0$ ms as P_3 enters at $t=25$, starts immediately and executes completely

Waiting time for $P_4 = (52 - 37) = 15$ ms as P_4 waits for $t=37-52$

Average waiting time = $(17 + 22 + 0 + 15)/4 = 13.5$ ms

Turnaround time for $P_1 = (42 - 0) = 42$ ms as P_1 enters at $t=0$ and exits at $t=42$

Turnaround time for $P_2 = (52 - 12) = 40$ ms as P_2 enters at $t=12$ and exits at $t=52$

Turnaround time for $P_3 = (29 - 25) = 4$ ms as P_3 enters at $t=25$ and exits at $t=29$

Turnaround time for $P_4 = (57 - 32) = 25$ ms as P_4 enters at $t=32$ and exits at $t=57$

Average turnaround time = $(42 + 40 + 4 + 25)/4 = 27.75$ ms

Advantages

- It is fair to I/O-bound (short) processes as these processes are not required to wait for an inordinate amount of time. Hence, are executed quickly.
- It prevents starvation by moving a lower priority process to a higher priority queue if it has been waiting for too long a period..

Disadvantages

- It is the most complex and cryptic scheduling algorithm.
- Moving the processes between queues causes a number of context switches which results in an increased overhead.
- The turnaround time for long processes may increase significantly.

Check Your Progress

8. What are the purpose of CPU utilization?
9. What is the name of two categories fall in scheduling algorithm?
10. Explain the first-come-first-served scheduling.
11. Define shortest remaining time next scheduling.
12. Give the definition of highest response ratio next scheduling.
13. What is advantage and disadvantage of multilevel queue scheduling?

9.7 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

NOTES

1. The main objective of multiprogramming is to keep the jobs organized in such a manner that CPU has always one task to execute in the queue. This confirms that the CPU is utilized to the maximum level by reducing its idle time. This purpose can be very well achieved by keeping the CPU busy at all the times. However, when two or more processes compete for the CPU at the same time, then choice has to be made which process to allocate the CPU next. This procedure of determining the next process to be executed on the CPU is called **process scheduling** and the module of an operating system that makes this decision is called the **scheduler**.
2. Transferring the control of CPU from one process to another demands for saving the context of the currently running process and loading the context of another ready process. This mechanism of saving and restoring the context of the ongoing process is known as **context switch**.
3. The “**context saving**” means the first thing an interrupt does is to **save** all the intermediary data (**save W, PROD, tmp**), and the last operation is to restore all these data.
4. Process spawning is a technique in which OS creates a child process by the request of another process. When OS creates a child process by the request of the parent process.
5. Scheduling is the process of arranging, controlling and optimizing work and workloads in a production process or manufacturing process.
6. The main aim of short-term scheduler is to opt for a process to be performed next on the CPU but it cannot allocate CPU to the selected process. The basic responsibility of setting up the execution of the selected process on the CPU is performed by some other module of the operating system, called **dispatcher**.
7. A process that no longer exists but still its PCB is not removed from the process table is known as a **zombie** process.
8. **CPU Utilization:** It is defined as the percentage of time the CPU is otherwise engaged in carrying out processes. To ensure better utilization, CPU must be kept as busy as possible, that is, there must be some process running everytime.
9. These scheduling algorithms fall into two categories, namely, *non-preemptive* and *preemptive*.

10. To implement FCFS scheduling, the implementation of ready queue is managed as a FIFO (First-In First-Out) queue. When the first process enters the ready queue, it immediately gets the CPU and starts executing. Meanwhile, other processes enter the system and are added to the end of queue by inserting their PCBs in the queue. When the currently running process completes or blocks, the CPU is allocated to the process at the front of the queue and its PCB is removed from the queue. In case, a currently running process was blocked and later it comes to the ready state, its PCB is linked to the end of queue.
11. The Shortest Remaining Time Next (SRTN) also known as **Shortest Time To Go (STG)**, is a preemptive version of the SJF scheduling algorithm. It takes into account the length of remaining CPU burst of the processes rather than the whole length in order to schedule them. Here also the scheduler always chooses the process for execution that has the shortest remaining processing time. While a process is being executed, the CPU can be taken back from it and assigned to some newly arrived process if the CPU burst of the new process is shorter than its remaining CPU burst. Notice that if at any point of time, the remaining CPU burst of two processes becomes equal; they are scheduled in the FCFS order.
12. The Highest Response Ratio Next (HRN) scheduling is a non-preemptive scheduling algorithm that schedules the processes based on their response ratio. Whenever, CPU becomes available, the process having the highest value of response ratio among all the ready processes is scheduled next. The Response Ratio (RR) of a process in the queue is computed by using the following equation.

$$\text{Response Ratio (RR)} = \frac{\text{Time since arrival} + \text{CPU burst}}{\text{CPU burst}}$$

13. Advantages

- It is fair to I/O-bound (short) processes as these processes are not required to wait for an inordinate amount of time. Hence, are executed quickly.
- It prevents starvation by moving a lower priority process to a higher priority queue if it has been waiting for too long a period..

Disadvantages

- It is the most complex and cryptic scheduling algorithm.
- Moving the processes between queues causes a number of context switches which results in an increased overhead.
- The turnaround time for long processes may increase significantly.

NOTES

NOTES**9.8 SUMMARY**

- A deadlock can be depicted with the help of a directed graph known as resource allocation graph.
- The main objective of multiprogramming is to keep the jobs organized in such a manner that CPU has always one task to execute in the queue. This confirms that the CPU is utilized to the maximum level by reducing its idle time.
- When two or more processes compete for the CPU at the same time, then choice has to be made which process to allocate the CPU next. This procedure of determining the next process to be executed on the CPU is called process scheduling and the module of an operating system that makes this decision is called the scheduler.
- Each process enters the system for execution. As soon as they enter the execution phase, they get into a queue called job queue (or input queue) on a mass storage device, such as hard disk.
- The ready queue contains all those processes that are waiting for the CPU.
- For each I/O device in the system, a separate queue called device queue is maintained.
- If an interrupt occurs, the CPU can be forcibly taken away from the currently executing process and the process has to wait until the interrupt is handled.
- If the time slice (in case of time sharing systems) of the process has expired, the process is put back into the ready queue.
- If the process creates a new process and has to wait until the child process terminates, the parent process is suspended. After the execution of child process, it is again put back into the ready queue.
- If the process has successfully completed its task, it is terminated. The PCB and all the resources allocated to the process are reallocated.
- Transferring the control of CPU from one process to another demands for saving the context of the currently running process and loading the context of another ready process. This mechanism of saving and restoring the context of the ongoing process is known as context switch.
- There are innumerable operations that can be fulfilled on processes which encompasses creating, terminating, suspending, or resuming a process, and so on.
- To successfully execute these operations, the operating system provides run-time services (or system calls) for the process management. The user may invoke these system calls either directly by embedding the process

supervisory calls in the user's program or indirectly by typing commands on the terminal which are translated by the system into system calls.

- Whenever, an operating system is initialized on the system, a number of processes (system processes) are created automatically. Out of these, some involve user interaction (called foreground processes) while others are not related with any user but still perform some specific function (called background processes) to smooth completion of the booting task.
- New processes can be created afterward as well. Sometimes, a user process may need to create one or more processes during its execution.
- Process spawning is a technique in which OS creates a child process by the request of another process. When OS creates a child process by the request of the parent process.
- The process that spawns a new process is called parent process whereas the spawned process is called the child process.
- Either the parent and child process may run concurrently (asynchronous process creation) or the parent process may wait until the child process completes its task and terminates.
- The newly created process may be the duplicate of the parent process in which case it contains a copy of the address space of its parent.
- The child process may have a new program loaded into its address space.
- Depending upon the condition, a process may be terminated either normally or forcibly by some another process.
- The reaction and response of a process while it is in the execution phase highly influences CPU scheduling.
- The time period elapsed in processing before performing the next I/O operation is known as CPU burst and the time period elapsed in performing I/O before the next CPU burst is known as I/O burst.
- An important facet of scheduling is to determine under what circumstances the scheduler should be initiated to make scheduling decisions.
- When a process switches from running to waiting state. Such a situation crops up, in case, the process has to wait for I/O or the termination of its child process or some another reason. In such situations, the scheduler has to select some ready process for execution.
- The main aim of short-term scheduler is to opt for a process to be performed next on the CPU but it cannot allocate CPU to the selected process.
- The system switches from the kernel mode to user mode as a user process is to be executed.
- Generally, CPU is busy performing varied tasks. Fairness is defined as the degree to which each process is getting an equal chance to execute.

Process Scheduling

NOTES

NOTES

- First-Come First-Serve (FCFS) is one of the simplest scheduling algorithms. As the name implies, the processes are executed in the order of their arrival in the ready queue, which means the process that enters the ready queue first gets the CPU first.
- To implement FCFS scheduling, the implementation of ready queue is managed as a FIFO (First-In First-Out) queue.
- The Shortest Remaining Time Next (SRTN) also known as Shortest Time To Go (STG), is a pre-emptive version of the SJF scheduling algorithm.
- The priority scheduling may be either preemptive or non-preemptive. The choice is made whenever a new process enters the ready queue while some process is executing.
- The Highest Response Ratio Next (HRN) scheduling is a non-preemptive scheduling algorithm that schedules the processes based on their response ratio.
- Round Robin (RR) scheduling is one of the most widely used preemptive scheduling algorithms which considers all the processes as equally important and treats them in a favourable manner.
- To implement the round robin scheduling algorithm, the ready queue is treated as a circular queue. All the processes arriving in the ready queue are put at the end of queue.
- The multilevel queue scheduling is designed especially for the environments where the processes can be categorized into different groups on the basis of their different response time requirements or different scheduling needs.
- The multilevel feedback queue scheduling also known as multilevel adaptive scheduling is an improved version of multilevel queue scheduling algorithm.
- Moving the processes between queues causes a number of context switches which results in an increased overhead.

9.9 KEY WORDS

- **Process Scheduling:** determining the next process to be executed on the CPU is called process scheduling.
- **Safe state:** A state in which the allocation of resources to processes does not lead to a deadlock; a system is in safe state only if there is a safe sequence.
- **Swapping:** Process spawning is a technique in which OS creates a child process by the request of another process. When OS creates a child process by the request of the parent process.

NOTES

- **Safe sequence:** A sequence of process execution such that each and every process executes till its completion. If no such sequence of process execution exists then the state of the system is said to be safe.
- **CPU Burst:** Every process in a computer system requires some amount of time for its execution. This time is both the CPU time and the I/O time. The CPU time is the time taken by CPU to execute the process. So, Burst time is the total time taken by the process for its execution on the CPU.
- **Zombie Process:** A process that no longer exists but still its PCB is not removed from the process table is known as a zombie process.

9.10 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Question

1. What do you understand by purposes of queues?
2. State about the context switch with the help of example.
3. Difference between save context and restore context.
4. Define throughput, turnaround time, waiting time and response time.
5. Give the definition of termination and also describe reasons for process termination.
6. Explain the term of process behaviour.
7. What is the difference between turnaround time and waiting time and response time?
8. Distinguish between non-preemptive and preemptive scheduling algorithms.
9. Explain priority-based scheduling.
10. Discuss about the multilevel queue scheduling.
11. Difference between the multilevel queue and multilevel feedback queue scheduling.

Long-Answer Questions

1. Elaborate the concept of process scheduling and also explain its type with the help of a diagram.
2. Distinguish among long-term scheduler, short-term scheduler and medium-term scheduler.
3. Describe the operation and processes and its types.
4. Explain the term of scheduling and also explain the decisions for scheduling concept.



NOTES

5. How to perform parameters and optimization scheduling criteria with the help of example?
 6. Which non-pre-emptive scheduling algorithms suffer from starvation and under what conditions?
 7. Consider four processes P1, P2, P3, and P4 with their arrival times, required CPU burst (in milliseconds), and priorities as shown in the following table.

Process	P ₁	P ₂	P ₃	P ₄
Arrival Time(ms)	0	1	3	4
CPU Burst(ms)	7	4	3	2
Priority	4	3	1	2

Assume that the lower the number, higher the priority. Compute the average waiting time and average turnaround time of the processes for each of the following scheduling algorithms. Also determine which of the following scheduling algorithms result in minimum waiting time.

9.11 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.

Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.

Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.



Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.

Process Scheduling

Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.

NOTES

Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.

Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.

Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.

Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

BLOCK - IV
REAL MEMORY AND VIRTUAL MEMORY
MANAGEMENT

UNIT 10 MEMORY MANAGEMENT

Structure

- 10.0 Introduction
- 10.1 Objectives
- 10.2 Memory Management
 - 10.2.1 Address Binding
 - 10.2.2 Contiguous Allocation
 - 10.2.3 Dynamic Loading and Linking
 - 10.2.4 Overlay
- 10.3 Memory Organization in Operating System
- 10.4 Memory Hierarchy
- 10.5 Memory Management Strategies
- 10.6 Answers to Check Your Progress Questions
- 10.7 Summary
- 10.8 Key Words
- 10.9 Self Assessment Questions and Exercises
- 10.10 Further Readings

10.0 INTRODUCTION

Most systems allow multiple processes to reside in the main memory at the same time to increase CPU utilization. It is the job of the memory manager, a part of the operating system, to manage memory between these processes in an efficient way. For this, it keeps track of which part of the memory is occupied and which part is free, allocates and deallocates memory to processes whenever required, and so on. Moreover, it provides a protection mechanism to protect the memory allocated to each process from being accessed by other processes.

Memory is used for storing programs and data that are required to perform a specific task. For CPU to operate at its maximum speed, it required an uninterrupted and high speed access to these memories that contain programs and data.

The memory manager keeps a record of the usage of each block of memory available for allocation with the help of various techniques, such as address binding, contiguous allocation, dynamic loading and linking as well as overlay. The main function of designing memory management techniques is to divide the main memory to incorporate with multiple processes and allocate memory for each process.

Both logical addresses and physical addresses are used in computer memory. By associating a physical address to a logical address, which is also known as a

virtual address, address binding assigns a physical memory location to a logical pointer. Address binding is part of the management of computer memory and is carried out by the operating system on behalf of programmes that require memory access.

Proper management of memory is vital for a computer system to operate properly. Modern operating systems have complex systems to properly manage memory. Failure to do so can lead to bugs, slow performance, and at worst case, takeover by viruses and malicious software. Nearly everything computer programmers do requires them to consider how to manage memory. Even storing a number in memory requires the programmer to specify how the memory should store it.

In this unit, you will study about the introduction of memory storage, memory organization, memory management, hierarchy of memory and management strategies.

10.1 OBJECTIVES

After going through this unit, you will be able to:

- Describe the significance of memory management
- Elaborate on the concept of address binding, contiguous allocation, overlay, dynamic linking and loading
- Know the Difference between logical and physical addresses
- Understand the concept of address binding
- Explain the basic memory management strategies

10.2 MEMORY MANAGEMENT

Memory management refers to moving the processes back and forth between main memory and disk during paging and swapping. The prime objective of memory management is to divide the main memory for accommodating multiple processes. For this, it uses various techniques, such as address binding, contiguous memory allocation and overlay process. These techniques are discussed below:

10.2.1 Address Binding

All programs reside on the hard disk and they must be brought to main memory for the execution. While doing so, the process moves between the hard disk and main memory. If we consider the various software and hardware components of a system they can be identified by three names:

- **Symbolic Name:** It refers to the name with which a user identifies the components. Examples include file names, program names, printer/device names, user names. We must specify the names in the context of location or the path.

Memory Management

NOTES

NOTES

- **Logical Name:** A logical name refers to the name used to label a specific entity. Examples include inodes for file, major/minor device numbers, process id (pid), user id (uid), gid, etc.
- **Physical Name:** It refers to an actual physical address of entity, such as address of file or data on disk or memory. It can be fixed once it enters or variable relocatable address.

Thus, the binding of instructions and data to actual memory location can occur at three different stages, namely compile time, load time and execution time.

Compile Time

The absolute code can be generated if the exact location of the process in the memory can be observed at compile time. This type of binding is termed as compile time binding. The compiler can be structured to generate code starting at address location. This code will work till the starting memory changes. If there is any change then the whole program will have to be rewritten. Thus for a system which allows a program to leave the main memory for some reason and to be reloaded later, the starting address of the program may change and the compiler will have to recompile the code to generate an absolute address. Hence, a compiler will typically bind the symbolic address to a relocatable address. If data binding is done at the compiler level then this process is called early binding.

The advantages of this binding scheme are as follows:

- The compiler produces an efficient code.
- It performs checking at an early stage.
- It can estimate the running time and space required for execution of the code.

Load Time

If the exact address where the program has to reside in the main memory is not known at compile time then the compiler must generate a relocatable code. The final binding will be done at load time. The loader loads the system library and other required data for the execution of program and also incorporates the changed value if the starting address is changed. If the code is generated at load time then it is called delayed binding. The linker and loader are used to generate the code. The advantages of delayed binding are as follows:

- It produces an efficient code.
- It allows the same code to be used at different places using separate compilations. Thus, it allows the sharing of object code.
- It is portable as the program can be loaded at any place.

Execution Time

If a process can move from one segment to another during execution, then binding is delayed till run time. We need special hardware support for performing the

required address translation, for example base and limit registers. The generation of code at run time is called late binding. The advantages of late binding are as follows:

- It supports techniques, for example virtual memory, dynamic linking/loading, overlaying, interpreting, etc.
- It is flexible in nature which allows dynamic reconfiguration.

However, the code is less efficient as the checking is done at run time.

10.2.2 Contiguous Allocation

The main memory is divided into two segments in which one is used to load the operating system and the other is used as user memory. The operating system is loaded and present in the system as long as the system is running. The user memory is for storing the user processes. In a multiprogramming environment, we want several processes to be in the memory. In contiguous memory allocation, each process is present in a contiguous memory location. Memory can be allocated in the following ways:

Fixed Size Partition

The entire memory is divided into fixed size blocks or partitions. The size of the partition is fixed. A process is loaded into a single partition. To accommodate every process in the partition, the partition size is selected based on the largest process size. If the partition can accommodate the largest process, then it can easily accommodate all other processes.

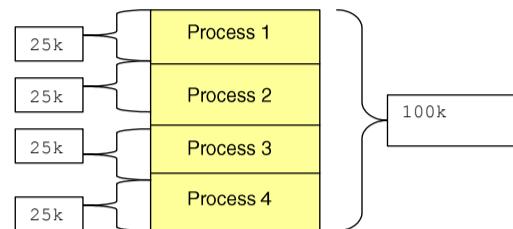
The problem in the fixed size partition memory allocation scheme is that when the process size is less than the partition size, there would be wastage of memory. For example, if the partition size is 100K (assuming that all the processes are less than the 100 k) and the process size is 25K, then if 25K is loaded into the 100K partition, there would be wastage of 75K memory. If all the process sizes are approximately equal to the partition size, then wastage of memory is minimum. The wastage of memory is internal to the process hence it is called internal fragmentation. Fixed size partition allocation scheme suffers with internal fragmentation.

Multiple Size Partition

In this allocation scheme, the memory initially is seen as a single large partition. When a process with size 'x' needs memory, then the memory is divided into two parts—a part for the process with size x and another part which is free. When another process requests the memory, then the free part is further divided into two parts to accommodate the new process. Hence, in this scheme, the memory is divided into variable size partitions. Consider the memory size as 100K and let four processes each with size 25K request the memory. Then the memory is allocated as follows:

Memory Management

NOTES

NOTES**Fig. 10.1 Multiple-size Partition**

If process 1 and process 3 finish execution and their memory is freed, then there would be $25\text{K} + 25\text{K} = 50\text{K}$ memory free. The free 50K memory is distributed in two partitions of size 25K each. If a new process comes with memory request 50K, then the available 50K memory cannot be allocated, as the memory is not present contiguously (Refer Figure 11.1). This problem is called external fragmentation. Then, external fragmentation is defined as the division of free storage into small pieces over a period of time, due to an inefficient memory allocation algorithm, resulting in the lack of sufficient storage for another program. This process is done because these small pieces are not contiguous. The variable size partition scheme suffers with external fragmentation.

Holes

An operating system keeps track of memory segments that are free and occupied. The operating system allocates memory to the process when process is to be executed and de-allocates the memory when the process terminates. When the process requests for memory, then the operating system searches for the block (hole) of memory that is large enough to accommodate the process. In the variable size partition, different sizes of memory are scattered throughout the memory. When a process requests the memory, the system searches for the set of variable memory holes and if it finds the hole that is large enough to hold the process, then that hole is allocated to the process. If the hole is very large then the hole is partitioned into two. One is allocated to the process and the other is returned to the free hole pool. In dynamic storage allocation, the free hole is selected using the following strategies:

First Fit: As the searching takes place from the beginning, the first hole that is big enough to accommodate is allocated to the process. Once the allocation is done, the searching takes place from the current position.

Best Fit: The entire memory is searched and the hole that is smallest to hold the process is allocated to the process. Here the entire memory is to be searched.

Worst Fit: In this strategy, the largest hole is allocated to the process. Scanning the entire memory searches the largest hole.

All the above strategies suffer from external fragmentation problems.

NOTES**Compaction**

Compaction is a technique that is used to solve the external fragmentation problem. In this technique, the non-contiguous memory holes are combined together that is all the used memory is moved towards one end of the memory and the unused memory is moved towards the other end. To do this, we need to move the running process to a new location. Moving a running process affects the performance as all the dependants are to be informed about the new location.

Logical Physical Address Space

An address generated by the CPU is referred to as logical address and physical address is the address of the physical memory. The set of all logical addresses, also called virtual addresses, generated by a program is called logical address space. This space is to be bound into physical address space by the memory management techniques. The compile time and load time binding schemes generate the same logical and physical addresses. However, they differ during execution time in the address binding scheme.

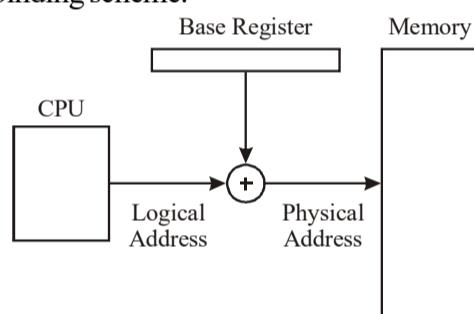


Fig. 10.2 Translation of Logical Address to Physical Address

The base address register is also called relocation register. The CPU generates the logical address from 0 to max. This CPU code can be loaded anywhere in the physical memory. The value of relocation register is added to every logical address generated by the CPU to get physical address where data actually resides. This helps to relocate the code into memory. Suppose that the base register has value of 15,000 and logical address generated is 330. The physical address will be given by 15,330. Thus, the memory management technique maps the logical address space to physical address space (Refer Figure 11.2).

10.2.3 Dynamic Loading and Linking

Dynamic loading refers to the process of copying the loadable image into memory, connecting it with any other programs already loaded and updating addresses as needed. Dynamic linking refers to the process of combining a set of programs including library routines to create a loadable image. These two techniques are discussed below:

NOTES

Dynamic Loading

The dynamic loading technique does not load a routine until it is called. All routines are kept on disk in relocatable load format and are loaded into memory whenever needed for execution. As unused routines are never loaded, this scheme provides better memory space utilization. It is also useful in the cases where there are large amounts of code which are not frequently required. Now no special support from the operating system is required the user implement this concept by proper program. The concern of design operating system is to provide library routine to implement dynamic loading.

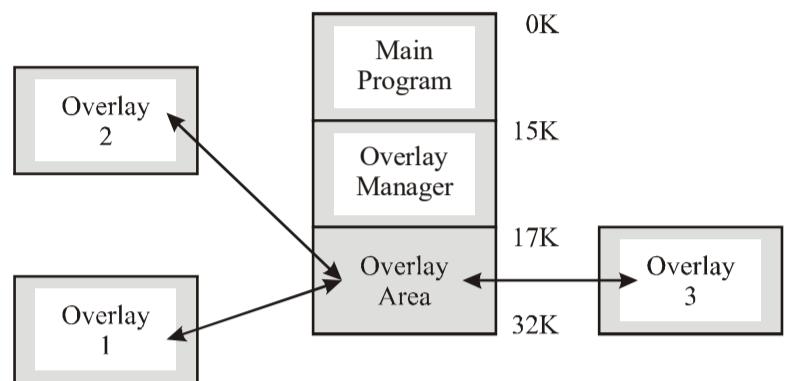
Dynamic Linking

Some operating systems support only static linking where the system library is treated like another object module and is combined to the code by the loader. But dynamic linking, such as dynamic loading, postpones the linking of code until execution time. This technique is used to load system library functions, subroutine libraries, etc. A small piece of code called stub is included in the image for each library routine reference. The stub replaces itself with the address of the routine and executes the routine. Thus, it is used for locating the appropriate memory resident library routine and loading it if the routine is not already present.

10.2.4 Overlay

The function overlay helps a process to be allocated even if it has a size larger than the amount of memory allocated to it. The underline principle is to keep only those instructions and data that are needed at any given time in the memory. When other data or instruction is required, it is loaded into space occupied by the processor that is no longer needed. The implementation is performed by the user with simple file structure. Reading from the file into memory and then jumping to another memory location and execute the newly read instruction. There is no special support from operating system there is just an overlay manager that bringing in and out the required overlay code as per demand of program.

Figure 10.3 shows the management of an overlay in which overlay manager loads an overlay segment when it is not in RAM. Overlay 1, overlay 2 and overlay 3 are interconnected with overlay area. The function of an overlay area is to signal the overlay handler which swaps out the old program segment and swaps in the next program segment with the help of device drivers, interrupt handlers and exception handlers.

NOTES**Fig. 10.3 Management of an Overlay**

Let us consider an example in which there is a two pass assembler: Pass 1 creates a Symbol Table and Pass 2 generates the machine code. In order to perform this task, both tasks require Symbol Table and some other common subroutines. But as both tasks are executed independently, there are some routines which are specific for particular task. We can divide the routines into the following components:

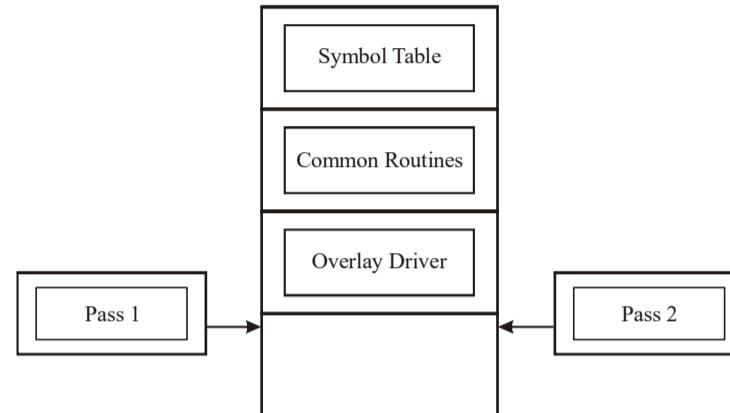
Pass1	80 KB
Pass2	70 KB
Symbol Table	30 KB
Common Routines	40 KB

Consider a memory of size 200 KB. If we try to load both on them simultaneously, it is not possible as both of them do not execute simultaneously. It is assumed that two overlays A and B are defined in which overlay driver (10KB) starts with overlay A in memory.

Let us assume that Overlay A consists of Symbol Table, Common Routines and Pass 1. Overlay B consists of Symbol Table, Common Routines and Pass 2 (Refer Figure 10.4).

An Overlay Manager of 15 K that manages the complete process.

Initially, Overlay A is loaded into memory. Once it is over, only Pass 2 of Overlay B will be loaded, as Symbol Table and Common Routine are same. It will be faster as lesser number of data is to be fetched.

NOTES**Fig. 10.4 Overlays for a Two Pass Assembler****Check Your Progress**

1. Define the concept of memory management.
2. Write the software and hardware component for address binding.
3. Explain the term of execution time.
4. State about fixed size partition.
5. What do you understand by dynamic loading and linking?
6. Define the term of overlay.

10.3 MEMORY ORGANIZATION IN OPERATING SYSTEM

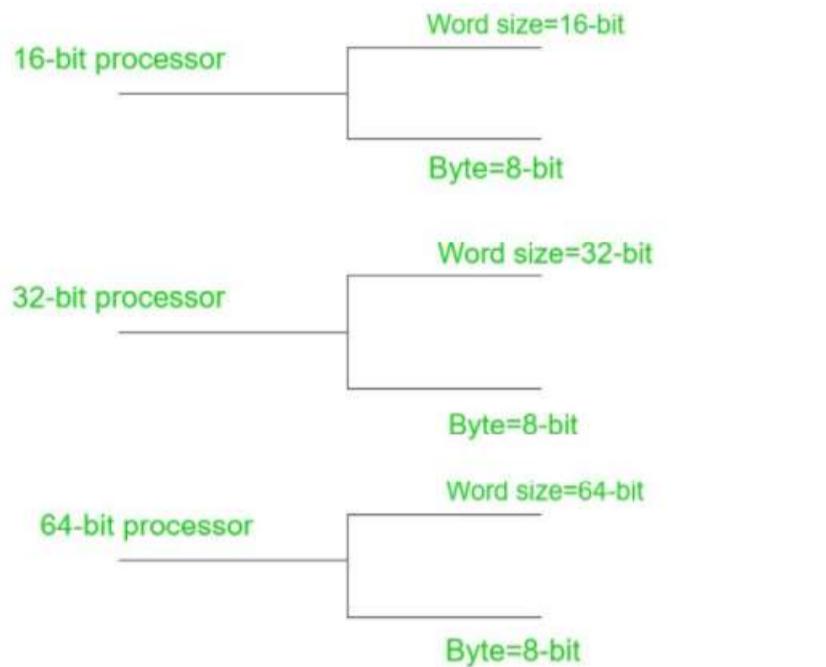
The memory is organized in the form of a cell, each cell is able to be identified with a unique number called address. Each cell is able to recognize control signals, such as ‘Read’ and ‘Write’, generated by CPU when it wants to read or write address. Whenever CPU executes the program there is a need to transfer the instruction from the memory to CPU because the program is available in memory. To access the instruction CPU generates the memory request.

Memory Request: Memory request contains the address along with the control signals. For Example, when inserting data into the stack, each block consumes memory (RAM) and the number of memory cells can be determined by the capacity of a memory chip.

Word Size: It is the maximum number of bits that a CPU can process at a time and it depends upon the processor. Word size is a fixed size piece of data handled as a unit by the instruction set or the hardware of a processor. Word size varies as per the processor architectures because of generation and the present technology, it could be low as 4-bits or high as 64-bits depending on what a

particular processor can handle. Word size is used for a number of concepts like Addresses, Registers, Fixed-point numbers, and Floating-point numbers.

Memory Management



NOTES

A memory unit is the collection of storage units or devices together. The memory unit stores the binary information in the form of bits. Generally, memory/storage is classified into 2 categories:

- **Volatile Memory:** This loses its data, when power is switched off.
- **Non-Volatile Memory:** This is a permanent storage and does not lose any data when power is switched off.

Computer Memory

In computing, memory refers to a device that is used to store information for immediate use in a computer or related computer hardware device. It typically refers to semiconductor memory, specifically Metal–Oxide–Semiconductor (MOS) memory, where data is stored within MOS memory cells on a silicon integrated circuit chip. The term ‘Memory’ is often synonymous with the term ‘Primary Storage’. Computer memory operates at a high speed, for example Random-Access Memory (RAM), as a distinction from storage that provides slow-to-access information but offers higher capacities. If needed, contents of the computer memory can be transferred to secondary storage; a very common way of doing this is through a memory management technique called virtual memory. An archaic synonym for memory is store.

The term ‘Memory’, meaning ‘Primary Storage’ or ‘Main Memory’, is often associated with addressable semiconductor memory, i.e., integrated circuits consisting of silicon-based MOS transistors, used for example as primary storage

NOTES

but also other purposes in computers and other digital electronic devices. There are two main kinds of semiconductor memory, volatile and non-volatile. Examples of non-volatile memory are flash memory (used as secondary storage) and ROM, PROM, EPROM and EEPROM memory (used for storing firmware, such as BIOS). Examples of volatile memory are primary storage, which is typically Dynamic Random-Access Memory (DRAM), and fast CPU cache memory, which is typically Static Random-Access Memory (SRAM) that is fast but energy-consuming, offering lower memory areal density than DRAM.

Most semiconductor memory is organized into memory cells or bistable flip-flops, each storing one bit (0 or 1). Flash memory organization includes both one bit per memory cell and multiple bits per cell (called MLC, Multiple Level Cell). The memory cells are grouped into words of fixed word length, for example 1, 2, 4, 8, 16, 32, 64 or 128 bits. Each word can be accessed by a binary address of N bit, making it possible to store 2^N words in the memory. This implies that processor registers normally are not considered as memory, since they only store one word and do not include an addressing mechanism.

History

In the early 1940s, memory technology often permitted a capacity of a few bytes. The first electronic programmable digital computer, the ENIAC, using thousands of octal-base radio vacuum tubes, could perform simple calculations involving 20 numbers of ten decimal digits which were held in the vacuum tube. The next significant advance in computer memory came with acoustic delay line memory, developed by J. Presper Eckert in the early 1940s. Through the construction of a glass tube filled with mercury and plugged at each end with a quartz crystal, delay lines could store bits of information in the form of sound waves propagating through mercury, with the quartz crystals acting as transducers to read and write bits. Delay line memory would be limited to a capacity of up to a few hundred thousand bits to remain efficient.

Two alternatives to the delay line, the Williams tube and selectron tube, originated in 1946, both using electron beams in glass tubes as means of storage. Using cathode ray tubes, Fred Williams would invent the Williams tube, which would be the first random-access computer memory. The Williams tube would prove more capacious than the selectron tube (the selectron was limited to 256 bits, while the Williams tube could store thousands) and less expensive. The Williams tube would nevertheless prove to be frustratingly sensitive to environmental disturbances.

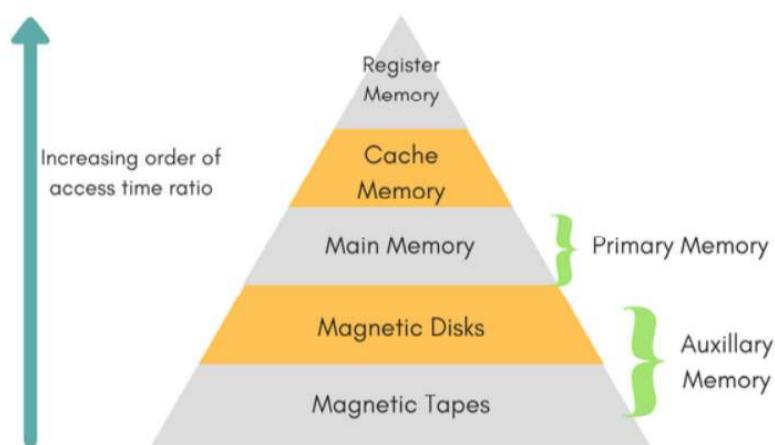
Efforts began in the late 1940s to find non-volatile memory. Magnetic-core memory allowed for recall of memory after power loss. It was developed by Frederick W. Viehe and A Wang in the late 1940s, and improved by Jay Forrester and Jan A. Rajchman in the early 1950s, before being commercialised with the Whirlwind computer in 1953. Magnetic-core memory would become the dominant

form of memory until the development of MOS semiconductor memory in the 1960s. Semiconductor memory began in the early 1960s with bipolar memory, which used bipolar transistors. Bipolar semiconductor memory made from discrete devices was first shipped by Texas Instruments to the United States Air Force in 1961. The same year, the concept of solid-state memory on an Integrated Circuit (IC) chip was proposed by applications engineer Bob Norman at Fairchild Semiconductor. The first bipolar semiconductor memory IC chip was the SP95 introduced by IBM in 1965. While bipolar memory offered improved performance over magnetic-core memory, it could not compete with the lower price of magnetic-core, which remained dominant up until the late 1960s. Bipolar memory failed to replace magnetic-core memory because bipolar flip-flop circuits were too large and expensive.

NOTES

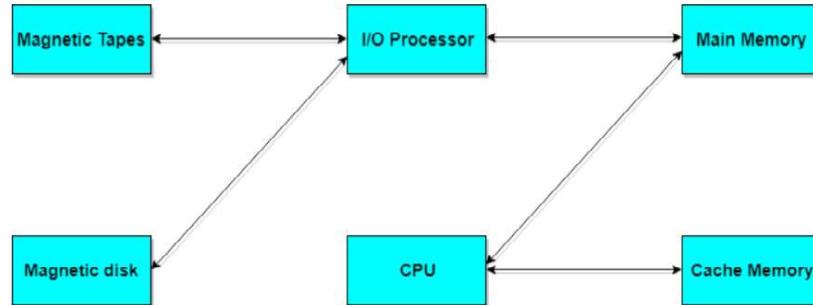
10.4 MEMORY HIERARCHY

The total memory capacity of a computer can be visualized by hierarchy of components. The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory. Auxiliary memory access time is generally 1000 times that of the main memory, hence it is at the bottom of the hierarchy.



The main memory occupies the central position because it is equipped to communicate directly with the CPU and with auxiliary memory devices through Input/output processor (I/O).

When the program not residing in main memory is needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space in main memory for other programs that are currently in use. The cache memory is used to store program data which is currently being executed in the CPU. Approximate access time ratio between cache memory and main memory is about 1 to 7~10.

NOTES**10.5 MEMORY MANAGEMENT STRATEGIES**

Proper management of memory is vital for a computer system to operate properly. Modern operating systems have complex systems to properly manage memory. Failure to do so can lead to bugs, slow performance, and at worst case, takeover by viruses and malicious software. Nearly everything computer programmers do requires them to consider how to manage memory. Even storing a number in memory requires the programmer to specify how the memory should store it.

Bugs: Improper management of memory is a common cause of bugs, including the following types:

In an arithmetic overflow, a calculation results in a number larger than the allocated memory permits. For example, a signed 8-bit integer allows the numbers “128 to +127. If its value is 127 and it is instructed to add one, the computer cannot store the number 128 in that space. Such a case will result in undesired operation, such as changing the number’s value to “128 instead of +128.

A memory leak occurs when a program requests memory from the operating system and never returns the memory when it’s done with it. A program with this bug will gradually require more and more memory until the program fails as it runs out. A segmentation fault results when a program tries to access memory that it does not have permission to access. Generally, a program doing so will be terminated by the operating system.

A buffer overflow means that a program writes data to the end of its allocated space and then continues to write data to memory that has been allocated for other purposes. This may result in erratic program behaviour, including memory access errors, incorrect results, a crash, or a breach of system security. They are thus the basis of many software vulnerabilities and can be maliciously exploited.

- 1. Rollout/Rollin (RO/RI):** Rollout/Rollin (RO/RI) is a computer operating system memory management technique where the entire non-shared code and data of a running program is swapped out to auxiliary memory (disk or drum) to free main storage for another task. Programs may be rolled out ‘by demand end or when waiting for some long event.’ Rollout/Rollin was commonly used in time-sharing systems, where the user’s ‘think time’ was

relatively long compared to the time to do the swap. Unlike virtual storage—paging or segmentation, rollout/Rollin does not require special memory management hardware; however, unless the system has relocation hardware such as a memory map or base and bounds registers, the program must be rolled back in to its original memory locations. Rollout/Rollin has been largely superseded by virtual memory.

Rollout/Rollin was an optional feature of OS/360 Multiprogramming with a Variable number of Tasks (MVT). Rollout/Rollin allows the temporary, dynamic expansion of a particular job beyond its originally specified region. When a job needs more space, rollout/Rollin attempts to obtain unassigned storage for the job's use. If there is no such unassigned storage, another job is rolled out, i.e., is transferred to auxiliary storage

So that its region may be used by the first job. When released by the first job, this additional storage is again available, either (1) As unassigned storage, if that was its source, or (2) To receive the job to be transferred back into main storage (rolled in).

2. Early computer systems: In early computer systems, programs typically specified the location to write memory and what data to put there. This location was a physical location on the actual memory hardware. The slow processing of such computers did not allow for the complex memory management systems used today. Also, as most such systems were single-task, sophisticated systems were not required as much. This approach has its pitfalls. If the location specified is incorrect, this will cause the computer to write the data to some other part of the program. The results of an error like this are unpredictable. In some cases, the incorrect data might overwrite memory used by the operating system. Computer crackers can take advantage of this to create viruses and malware.

3. Virtual memory: Virtual memory is a system where all physical memory is controlled by the operating system. When a program needs memory, it requests it from the operating system. The operating system then decides in what physical location to place the program's code and data. This offers several advantages. Computer programmers no longer need to worry about where their data is physically stored or whether the user's computer will have enough memory. It also allows multiple types of memory to be used. For example, some data can be stored in physical RAM chips while other data is stored on a hard drive (For example, in a swap file), functioning as an extension of the cache hierarchy. This drastically increases the amount of memory available to programs. The operating system will place actively used data in physical RAM, which is much faster than hard disks. When the amount of RAM is not sufficient to run all the current programs, it can result in a situation where the computer spends more time moving data from RAM to disk and back than it does accomplishing tasks; this is known as thrashing.

NOTES

NOTES

- 4. Protected memory:** Protected memory is a system where each program is given an area of memory to use and is not permitted to go outside that range. Use of protected memory greatly enhances both the reliability and security of a computer system.

Without protected memory, it is possible that a bug in one program will alter the memory used by another program. This will cause that other program to run off of corrupted memory with unpredictable results. If the operating system's memory is corrupted, the entire computer system may crash and need to be rebooted. At times programs intentionally alter the memory used by other programs. This is done by viruses and malware to take over computers. It may also be used benignly by desirable programs which are intended to modify other programs; in the modern age, this is generally considered bad programming practice for application programs, but it may be used by system development tools such as debuggers, for example to insert breakpoints or hooks. Protected memory assigns programs their own areas of memory.

If the operating system detects that a program has tried to alter memory that does not belong to it, the program is terminated (or otherwise restricted or redirected). This way, only the offending program crashes, and other programs are not affected by the misbehaviour (whether accidental or intentional). Protected memory systems almost always include virtual memory as well.

Following are the techniques to manage memory in an operating system,

- 5. Single contiguous allocation:** Single allocation is the simplest memory management technique. All the computer's memory, usually with the exception of a small portion reserved for the operating system, is available to the single application. MS-DOS is an example of a system which allocates memory in this way. An embedded system running a single application might also use this technique. A system using single contiguous allocation may still multitask by swapping the contents of memory to switch among users. Early versions of the MUSIC operating system used this technique.

- 6. Partitioned allocation:** Partitioned allocation divides primary memory into multiple memory partitions, usually contiguous areas of memory. Each partition might contain all the information for a specific job or task. Memory management consists of allocating a partition to a job when it starts and unallocating it when the job ends. Partitioned allocation usually requires some hardware support to prevent the jobs from interfering with one another or with the operating system. The IBM System/360 used a lock-and-key technique. Other systems used base and bounds registers which contained the limits of the partition and flagged invalid accesses. The UNIVAC 1108 Storage Limits Register had separate base/bound sets for instructions and data. The system took advantage of memory interleaving to place what were called the i bank and d bank in separate memory modules.

NOTES

Partitions may be either static that is defined at Initial Program Load (IPL) or boot time or by the computer operator or dynamic, that is automatically created for a specific job. IBM System/360 Operating System Multiprogramming with a Fixed Number of Tasks (MFT) is an example of static partitioning, and Multiprogramming with a Variable Number of Tasks (MVT) is an example of dynamic. MVT and successors use the term region to distinguish dynamic partitions from static ones in other systems. Partitions may be relocatable using hardware typed memory, like the Burroughs Corporation B5500, or base and bounds registers like the PDP-10 or GE-635. Relocatable partitions are able to be compacted to provide larger chunks of contiguous physical memory. Compaction moves ‘in-use’ areas of memory to eliminate ‘Holes’ or unused areas of memory caused by process termination in order to create larger contiguous free areas.

Some systems allow partitions to be swapped out to secondary storage to free additional memory. Early versions of IBM’s Time-Sharing Option (TSO) swapped users in and out of a single time-sharing partition.

7. **Paged memory management:** Paged allocation divides the computer’s primary memory into fixed-size units called page frames, and the program’s virtual address space into pages of the same size. The hardware memory management unit maps pages to frames. The physical memory can be allocated on a page basis while the address space appears contiguous.

Usually, with paged memory management, each job runs in its own address space. However, there are some single address space operating systems that run all processes within a single address space, such as IBM i, which runs all processes within a large address space, and IBM OS/VS2 SVS, which ran all jobs in a single 16MiB virtual address space. Paged memory can be demand-paged when the system can move pages as required between primary and secondary memory.

8. **Segmented memory management:** Segmented memory is the only memory management technique that does not provide the user’s program with a ‘linear and contiguous address space.’ Segments are areas of memory that usually correspond to a logical grouping of information such as a code procedure or a data array. Segments require hardware support in the form of a segment table which usually contains the physical address of the segment in memory, its size, and other data, such as access protection bits and status (swapped in, swapped out, etc.)

Segmentation allows better access protection than other schemes because memory references are relative to a specific segment and the hardware will not permit the application to reference memory not defined for that segment. It is possible to implement segmentation with or without paging. Without paging support the segment is the physical unit swapped in and out of memory if required. With paging support the pages are usually the unit of swapping and segmentation only adds an additional level of security.

NOTES

Addresses in a segmented system usually consist of the segment id and an offset relative to the segment base address, defined to be offset zero. The Intel IA-32 (x86) architecture allows a process to have up to 16,383 segments of up to 4GiB each. IA-32 segments are subdivisions of the computer's linear address space, the virtual address space provided by the paging hardware. The Multics operating system is probably the best-known system implementing segmented memory. Multics segments are subdivisions of the computer's physical memory of up to 256 pages, each page being 1K 36-bit words in size, resulting in a maximum segment size of 1MiB (with 9-bit bytes, as used in Multics). A process could have up to 4046 segments.

Check Your Progress

7. What is memory organization in operating system?
8. Explain the term of memory request.
9. State about computer memory.
10. Write the definition of memory hierarchy.
11. What is memory management strategies?
12. Explain the term of virtual memory.
13. Define segmented memory management.

10.6 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Memory management refers to moving the processes back and forth between main memory and disk during paging and swapping. The prime objective of memory management is to divide the main memory for accommodating multiple processes. For this, it uses various techniques, such as address binding, contiguous memory allocation and overlay process.
2. Symbolic Name
Logical Name
Physical Name
3. If a process can move from one segment to another during execution, then binding is delayed till run time. We need special hardware support for performing the required address translation, for example base and limit registers. The generation of code at run time is called late binding.⁴ This is the oldest and simplest technique used to put more than one processes in the main memory. In this partitioning, number of partitions (non-overlapping) in RAM are fixed but size of each partition may or may not be same. As it is contiguous allocation, hence no spanning is allowed.

NOTES

4. This is the oldest and simplest technique used to put more than one processes in the main memory. In this partitioning, number of partitions (non-overlapping) in RAM are fixed but size of each partition may or may not be same. As it is contiguous allocation, hence no spanning is allowed.
5. Dynamic loading refers to the process of copying the loadable image into memory, connecting it with any other programs already loaded and updating addresses as needed. Dynamic linking refers to the process of combining a set of programs including library routines to create a loadable image.
6. The function overlay helps a process to be allocated even if it has a size larger than the amount of memory allocated to it. The underline principle is to keep only those instructions and data that are needed at any given time in the memory. When other data or instruction is required, it is loaded into space occupied by the processor that is no longer needed.
7. The memory is organized in the form of a cell, each cell is able to be identified with a unique number called address. Each cell is able to recognize control signals, such as ‘Read’ and ‘Write’, generated by CPU when it wants to read or write address. Whenever CPU executes the program there is a need to transfer the instruction from the memory to CPU because the program is available in memory.
8. Memory Request: Memory request contains the address along with the control signals. For Example, when inserting data into the stack, each block consumes memory (RAM) and the number of memory cells can be determined by the capacity of a memory chip.
9. In computing, memory refers to a device that is used to store information for immediate use in a computer or related computer hardware device. It typically refers to semiconductor memory, specifically metal–oxide–semiconductor memory, where data is stored within MOS memory cells on a silicon integrated circuit chip.
10. The total memory capacity of a computer can be visualized by hierarchy of components. The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory.
11. Proper management of memory is vital for a computer system to operate properly. Modern operating systems have complex systems to properly manage memory.
12. Virtual memory: Virtual memory is a system where all physical memory is controlled by the operating system. When a program needs memory, it requests it from the operating system. The operating system then decides in what physical location to place the program’s code and data.
13. Segmented memory management: Segmented memory is the only memory management technique that does not provide the user’s program with a ‘linear and contiguous address space.’ Segments are areas of memory that

NOTES

usually correspond to a logical grouping of information such as a code procedure or a data array. Segments require hardware support in the form of a segment table which usually contains the physical address of the segment in memory, its size, and other data, such as access protection bits and status (swapped in, swapped out, etc.)

10.7 SUMMARY

- Memory management refers to move processes back and forth between main memory and disk during paging and swapping. The prime objective of memory management is to divide the main memory for accommodating multiple processes.
- Memory management is an activity, which is carried out in the kernel of the operating system.
- The kernel itself is the central part of an operating system, it manages the operations of the computer and its hardware, and however it's most known for managing the memory and the CPU time.
- One of the key functions of the memory management system in a computer is assigning memory to a number of different running programs to keep the performance of the system stable.
- Memory in the system is dynamically allocated depending on the requirement, and it is freed up when the process no longer requires the memory, thus allocating that slot of memory to another process if needed.
- Ultimately memory management will depend on the how effective the configuration is in the hardware, operating system, and programs or applications.
- At hardware level, memory management involves physical devices that store the data. An example of this would Random Access Memory (RAM), furthermore this also includes memory caches and flash based SSD's (Solid State Drives).
- At the operating system level, memory management involves the allocation of specific memory blocks to individual programs as user demand changes.
- A logical name refers to the name used to label a specific entity. Examples include inodes for file, major/minor device numbers, process id (pid), user id (uid), gid, etc.
- The binding of instructions and data to actual memory location can occur at three different stages, namely compile time, load time and execution time.
- The absolute code can be generated if the exact location of the process in the memory can be observed at compile time. This type of binding is termed as compile time binding.

NOTES

- A process is loaded into a single partition. To accommodate every process in the partition, the partition size is selected based on the largest process size. The main memory is divided into two segments in which one is used to load the operating system and the other is used as user memory.
- The entire memory is divided into fixed size blocks or partitions. The size of the partition is fixed. A process is loaded into a single partition. To accommodate every process in the partition, the partition size is selected based on the largest process size.
- An operating system keeps track of memory segments that are free and occupied. The operating system allocates memory to the process when process is to be executed and de-allocates the memory when the process terminates.
- Compaction technique is used to solve the external fragmentation problem. In this technique, the non-contiguous memory holes are combined together that is all the used memory is moved towards one end of the memory and the unused memory is moved towards the other end.
- The memory initially is seen as a single large partition. When a process with size ‘x’ needs memory, then the memory is divided into two parts—a part for the process with size x and another part which is free.
- When another process requests the memory, then the free part is further divided into two parts to accommodate the new process.
- An operating system keeps track of memory segments that are free and occupied. The operating system allocates memory to the process when process is to be executed and de-allocates the memory when the process terminates.
- When the process requests for memory, then the operating system searches for the block (hole) of memory that is large enough to accommodate the process.
- As the searching takes place from the beginning, the first hole that is big enough to accommodate is allocated to the process. Once the allocation is done, the searching takes place from the current position.
- The entire memory is searched and the hole that is smallest to hold the process is allocated to the process. Here the entire memory is to be searched.
- The largest hole is allocated to the process. Scanning the entire memory searches the largest hole.
- Compaction is a technique that is used to solve the external fragmentation problem.
- An address generated by the CPU is referred to as logical address and physical address is the address of the physical memory. The set of all logical addresses, also called virtual addresses, generated by a program is called logical address space.

NOTES

- Dynamic loading refers to the process of copying the loadable image into memory, connecting it with any other programs already loaded and updating addresses as needed.
- The function overlay helps a process to be allocated even if it has a size larger than the amount of memory allocated to it.
- The implementation is performed by the user with simple file structure. Reading from the file into memory and then jumping to another memory location and execute the newly read instruction.
- There is no special support from operating system there is just an overlay manager that bringing in and out the required overlay code as per demand of program.
- The memory is organized in the form of a cell, each cell is able to be identified with a unique number called address.
- Each cell is able to recognize control signals, such as ‘Read’ and ‘Write’, generated by CPU when it wants to read or write address.
- Whenever CPU executes the program there is a need to transfer the instruction from the memory to CPU because the program is available in memory.
- To access the instruction CPU generates the memory request.
- Memory request contains the address along with the control signals.
- A memory unit is the collection of storage units or devices together. The memory unit stores the binary information in the form of bits. Generally, memory/storage is classified into 2 categories:
- Memory refers to a device that is used to store information for immediate use in a computer or related computer hardware device. It typically refers to semiconductor memory, specifically Metal–Oxide–Semiconductor (MOS) memory, where data is stored within MOS memory cells on a silicon integrated circuit chip.
- The total memory capacity of a computer can be visualized by hierarchy of components.
- The main memory occupies the central position because it is equipped to communicate directly with the CPU and with auxiliary memory devices through Input/output processor (I/O).
- When the program not residing in main memory is needed by the CPU, they are brought in from auxiliary memory.
- Proper management of memory is vital for a computer system to operate properly.
- Modern operating systems have complex systems to properly manage memory. Failure to do so can lead to bugs, slow performance, and at worst case, takeover by viruses and malicious software.

NOTES

- Nearly everything computer programmers do requires them to consider how to manage memory.
- Even storing a number in memory requires the programmer to specify how the memory should store it.
- Improper management of memory is a common cause of bugs.
- Rollout/Rollin (RO/RI) is a computer operating system memory management technique where the entire non-shared code and data of a running program is swapped out to auxiliary memory (disk or drum) to free main storage for another task. Programs may be rolled out ‘by demand end or when waiting for some long event.
- In early computer systems, programs typically specified the location to write memory and what data to put there. This location was a physical location on the actual memory hardware.
- Virtual memory is a system where all physical memory is controlled by the operating system. When a program needs memory, it requests it from the operating system.
- Protected memory is a system where each program is given an area of memory to use and is not permitted to go outside that range.
- Use of protected memory greatly enhances both the reliability and security of a computer system.
- If the operating system detects that a program has tried to alter memory that does not belong to it, the program is terminated (or otherwise restricted or redirected).
- Single allocation is the simplest memory management technique. All the computer’s memory, usually with the exception of a small portion reserved for the operating system, is available to the single application.
- Partitioned allocation divides primary memory into multiple memory partitions, usually contiguous areas of memory.
- Partitions may be either static that is defined at Initial Program Load (IPL) or boot time or by the computer operator or dynamic, that is automatically created for a specific job.
- Some systems allow partitions to be swapped out to secondary storage to free additional memory. Early versions of IBM’s Time-Sharing Option (TSO) swapped users in and out of a single time-sharing partition.
- Paged allocation divides the computer’s primary memory into fixed-size units called page frames, and the program’s virtual address space into pages of the same size.
- Segmented memory is the only memory management technique that does not provide the user’s program with a ‘Linear and Contiguous Address Space.’

NOTES

- Segmentation allows better access protection than other schemes because memory references are relative to a specific segment and the hardware will not permit the application to reference memory not defined for that segment.
- Addresses in a segmented system usually consist of the segment id and an offset relative to the segment base address, defined to be offset zero.

10.8 KEY WORDS

- **Symbolic Name:** It refers to the name with which a user identifies the components. Examples include file names, program names, printer/device names, user names. We must specify the names in the context of location or the path.
- **Logical Name:** A logical name refers to the name used to label a specific entity. Examples include inodes for file, major/minor device numbers, process id (pid), user id (uid), gid, etc.
- **Physical Name:** It refers to an actual physical address of entity, such as address of file or data on disk or memory. It can be fixed once it enters or variable relocatable address.
- **Compile time:** The absolute code can be generated if the exact location of the process in the memory can be observed at compile time.
- **Bugs:** Improper management of memory is a common cause of bugs.
- **Overlay:** overlaying means “The Process of Transferring a Block of Program Code or Other Data into Main Memory, Replacing What Is Already Stored”. Overlaying is a programming method that allows programs to be larger than the computer’s main memory.
- **Word Size:** It is the maximum number of bits that a CPU can process at a time and it depends upon the processor.

10.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Write the names which are used in address binding.
2. Explain the term of compile time.
3. What is holes and its types?
4. Define logical physical address space.
5. Write the name of two techniques for dynamic loading and linking.
6. How to generate memory request for memory organization?
7. What is the meaning of memory?

8. How to work bugs for memory management strategies?
9. Define the term of pages memory management.

Memory Management

Long-Answer Questions

1. Define memory management. What is the role of contiguous allocation of memory in memory management?
2. Discuss about three different stages for address binding.
3. Discuss briefly contiguous allocation.
4. What are dynamic loading and linking and its techniques?
5. Explain the management of overlay with the help of diagram.
6. Discuss about computer memory in details.
7. How to work memory hierarchy for operating system with the help of diagram?
8. Elaborate on the difference between virtual memory and protected memory.
9. Elaborate on the term single contiguous and partitioned allocation.
10. Write short notes on the following:
 - (a) Bugs
 - (b) Rollout/Rollin(RO/RI)
 - (c) Paged memory management
 - (d) Segmented memory management

NOTES

10.10 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.
- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.
- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.
- Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.
- Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.
- Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

NOTES

UNIT 11 MEMORY MANAGEMENT STRATEGIES

Structure

- 11.0 Introduction
 - 11.1 Objectives
 - 11.2 Preliminaries
 - 11.3 Contiguous Memory Allocation
 - 11.4 Non-Contiguous Memory Allocation
 - 11.5 Answers to Check Your Progress Questions
 - 11.6 Summary
 - 11.7 Key Words
 - 11.8 Self Assessment Questions and Exercises
 - 11.9 Further Readings
-

11.0 INTRODUCTION

Most systems allow multiple processes to reside in the main memory at the same time to increase CPU utilization. It is the job of the memory manager, a part of the operating system, to manage memory between these processes in an efficient way. For this, it keeps track of which part of the memory is occupied and which part is free, allocates and deallocates memory to processes, whenever required, and so on. Moreover, it provides a protection mechanism to protect the memory allocated to each process from being accessed by other processes.

All these strategies require the entire process to be in main memory before their execution. Thus, the size of the process is limited to the size of the physical memory. To overcome this limitation, a memory management scheme called overlaying can be used, which allows a process to execute irrespective of the size of the system's having insufficient physical memory. The programmer splits a program into smaller parts called overlays in such a way that no two overlays are required to be in main memory at the same time. An overlay is loaded into memory only when it is needed. Initially, overlay 0 would run. When it is completed, it would call another overlay, and so on until the process terminates. These overlays reside on the disk and are swapped in and out of memory dynamically as needed, thereby reducing the amount of memory needed by the process. Despite all these advantages, one major disadvantage of this technique is that it requires major involvement of the programmer. Moreover, splitting a program into smaller parts is quite time consuming.

This resulted in the formulation of another memory management technique known as virtual memory. Virtual memory gives the illusion that the system has a

NOTES

much larger memory than is actually available. The combined size of code, data and stack may exceed the amount of physical memory. Virtual memory frees programs from the constraints of physical memory limitation. The virtual memory can be implemented by demand paging or demand segmentation. Of these two ways, demand paging is commonly used as it is easier to implement.

In computer operating systems, paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory. In this scheme, the operating system retrieves data from secondary storage in same-size blocks called pages.

In this unit, you will study about the concept of the contiguous vs non-contiguous memory allocation, fixed partition multiprogramming and variable partition multiprogramming.

11.1 OBJECTIVES

After going through this unit, you will be able to:

- Differentiate between logical and physical addresses
- Understand the basic concept of address binding
- Discuss the basic memory management strategies
- Explain the memory management scheme, called paging
- Understand how the memory management scheme, segmentation, is used to overcome the drawbacks of paging
- Describe a memory management scheme that is a combination of segmentation and paging (called segmentation with paging)

11.2 PRELIMINARIES

Every byte in the memory has a specific address that may range from 0 to some maximum value as defined by the hardware. This address is known as **physical address**. Whenever, a program is brought into the main memory for execution, it occupies certain number of memory locations. The set of all physical addresses used by the program is known as **physical address space**. However, before a program can be executed, it must be compiled to produce the machine code. A program is compiled to run starting from some fixed address and accordingly all the variables and procedures used in the source program are assigned some specific addresses known as **logical addresses**. Thus, in machine code, all references to data or code are made by specifying the logical addresses and not by the variable or procedure names, and so on. The range of addresses that user programs can use is system-defined and the set of all logical addresses used by a user program is known as its **logical address space**.

NOTES

When a user program is brought into main memory for execution, its logical addresses must be mapped to physical addresses. This mapping from addresses associated with a program to memory addresses is known as **address binding**. The address binding can take place at one of the following times:

- **Compile Time:** Address binding takes place at compile time if it is known which addresses the program will occupy in the main memory at that time. In this case, the program generates **absolute code** at the compile time only, that is, logical addresses are same as that of physical addresses.
- **Load Time:** Address binding occurs at load time if it is not known at compile time which addresses the program will occupy in the main memory. In this case, the program generates **relocatable code** at the compile time which is then converted into the absolute code at the load time.
- **Run Time:** The address binding occurs at run time if the process is supposed to move from one memory segment to other during its execution. In this case also, the program generates relocatable code at compile time which is then converted into the absolute code at the run time.

Note: The run time address binding is performed by the hardware device known as Memory-Management Unit (MMU).

11.3 CONTIGUOUS MEMORY ALLOCATION

In contiguous memory allocation, each process is allocated a single contiguous part of the memory. The different memory management schemes that are based on this approach are *single partition* and *multiple partitions*.

Single Partition

One of the simplest ways to manage memory is to partition the main memory into two parts. One of them is permanently allocated to an operating system while the other part is allocated to the user process (See Figure 11.1). In this figure, an operating system is in lower part of the memory. However, it is not essential that operating system must reside at the bottom of the memory; it can reside at the upper part of the memory also. In order to provide a contiguous area for the user process, it usually resides at one extreme end of the memory. The factor that decides the location of the operating system in the memory is the location of the interrupt vector. An operating system is placed at the same end of the memory where the interrupt vector is located.

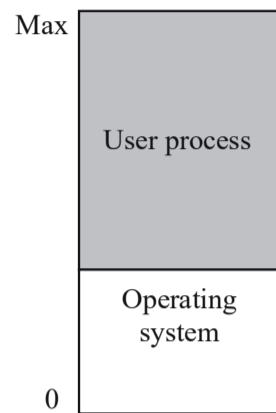
NOTES

Fig. 11.1 Memory Having Single Partition

In this scheme, only one process can be executed at a time. Whenever, a process is to be executed, the operating system loads it into the main memory for execution. After termination of that process, the operating system waits for another process. When another process arrives, the operating system loads it into the main memory, thus overwriting the first one.

This scheme is easy to implement. Generally, the operating system needs to keep track of the first and the last location allocated to the user processes. However, in this case, the first location is immediately following the operating system and the last location is determined by the capacity of the memory. It needs no hardware support except for protecting the operating system from the user process.

Note: The memory management scheme having single partition is used by single process microcomputer operating systems, such as CP/M and PC DOS.

Multiple Partitions

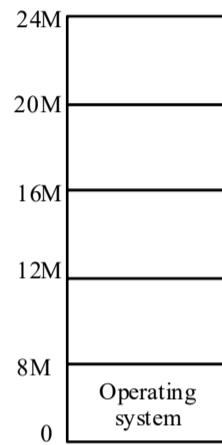
A single partition scheme restricts the system to have only one process in memory at a time that reduces utilization of the CPU as well as of memory. Thus, monoprogramming systems are rarely used. Most of the systems used today support multiprogramming which allows multiple processes to reside in the memory at the same time. The simple way to achieve multiprogramming is to divide main memory into a number of partitions which may be of fixed or variable size.

Fixed Partitions

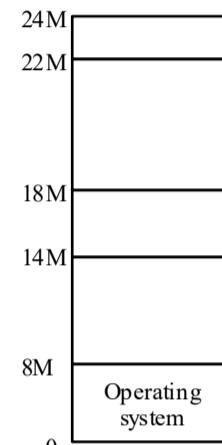
In this technique, each partition is of fixed size and can contain only one process. There are two alternatives for this technique—equal-sized partition and unequal-sized partition (Refer Figure 11.2). First, consider the case of equal-sized partitions where any process can be loaded into any partition. Whenever, a partition is free, a process whose size is less than or equal to the partition size is selected from the input queue and loaded into this partition. When the process terminates, the partition becomes free to be allocated to another process. The implementation of this method

NOTES

does not require much effort since all partitions are of same size. The operating system is required to keep track of only the partition occupied by each process. For this, it maintains a table that keeps either the starting address of each process or the partition number occupied by each process.



(a) Equal-Size Partitions



(b) Unequal-Size Partitions

Fig. 11.2 Memory having Multiple Fixed Partitions

There is one problem with this method that is the memory utilization is not efficient. Any process regardless of how small it is, occupies an entire partition which leads to the wastage of memory within the partition. What is fragments. This phenomenon which results in the wastage of memory within the partition is called **internal fragmentation**. For example, loading a process of size $4M-n$ bytes into a partition of size $4M$ (where, M stands for Megabytes) would result in a wasted space of n bytes within the partition.

This problem cannot be resolved completely but can be reduced to some extent by using unequal-sized partition method where a separate input queue is maintained for each partition. Whenever a process arrives, it is placed into the input queue of the smallest partition large enough to hold it. When this partition becomes free, it is allocated to the process. According to Figure 11.2(b), if a process of size $5M$ arrives, it will be accommodated in the partition of size $6M$. In this case also, some memory is wasted, that is, internal fragmentation still exists, but less than that of equal-sized partition method.

With this method, there may be possibility that the input queue for a large partition is empty but the queue for small partition is full (Refer Figure 11.3(a)). That is, the small jobs have to wait to be loaded into memory, though a large amount of memory is free. To prevent this, a single input queue can be maintained (Refer Figure 11.3(b)). Whenever, a partition becomes free, the process that fits in it can be chosen from the input queue using some scheduling algorithm.

NOTES

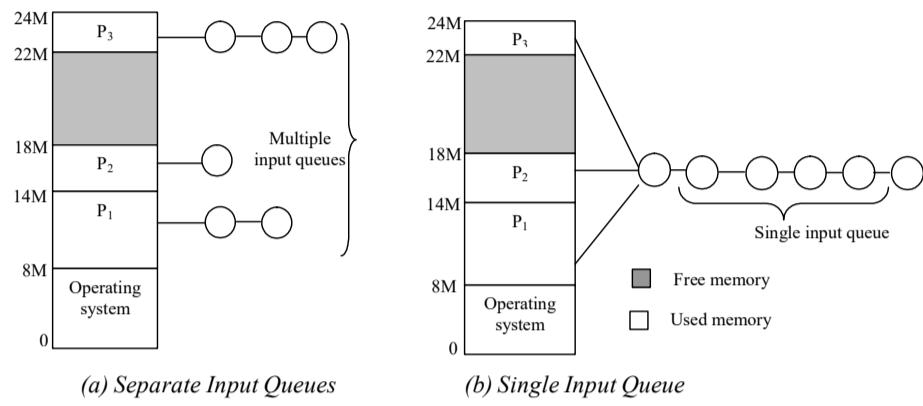


Fig. 11.3 Memory Allocation in Fixed Partitions

The fixed-partitioning technique is easy to implement and requires less overhead but has some disadvantages which are as follows:

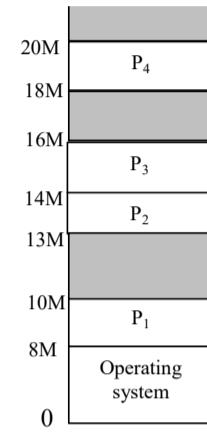
- The number of processes in memory depends on the number of partitions. Thus, the degree of multiprogramming is limited.
- The memory cannot be used efficiently in case of small-sized processes.

Note: The technique having fixed-partitions is no longer in use.

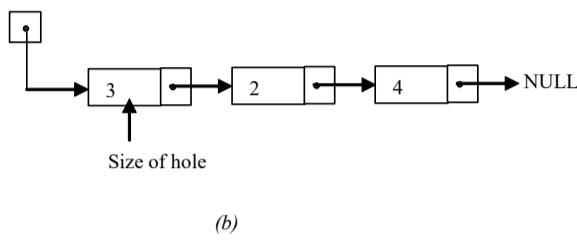
Variable Partitions

To overcome the disadvantages of fixed-partitions technique, a technique called Multiprogramming with a Variable number of Tasks (MVT) is used. It is the generalization of the fixed partitions technique in which the partitions can vary in number and size. In this technique, the amount of memory allocated is exactly the amount of memory a process requires. To implement this, the table maintained by the operating system stores both the starting address and ending address of each process.

Initially, when there is no process in the memory, the whole memory is available for allocation and is considered as a single large partition of available memory (**a hole**). Whenever a process requests for the memory, the hole large enough to accommodate that process is allocated. The rest of the memory is available to other processes. As soon as the process terminates, the memory occupied by it is deallocated and can be used for other processes. Thus, at a given point of time, some parts of memory may be in use while others may be free (Refer Figure 11.4(a)). Now to make further allocations, the memory manager must keep track of the free space in memory. For this, the memory manager maintains a free-storage list that keeps track of the unused part (holes of variable sizes) of memory. The free-storage list is implemented as a linked list where each node contains the size of the hole and the address of the next available hole (Refer Figure 11.4 (b)).

NOTES

(b)

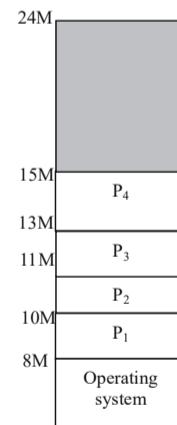


(b)

Fig. 11.4 Memory Map and Free-Storage List

In general, at a certain point of time, there will be a set of holes of various sizes dispersed in the memory. As a result, there may be possibility that the total available memory is large enough to accommodate the waiting process. However, it cannot be utilized as it is scattered. This wastage of the memory space is called **external fragmentation** (also known as **checkerboarding**) since, the wasted memory is not a part of any partition. For example, if a request for a partition of size 5M arrives, it cannot be granted because no single partition is available that is large enough to satisfy the request (Refer Figure 11.4). However, the combined free space can definitely satisfy the request.

To get rid of this problem, it is desirable to relocate (or shuffle) some or all portions of the memory in order to place all the free holes together at one end of memory to make one large hole. This technique of reforming the storage is termed as **compaction**. Compaction results in the memory partitioned into two contiguous blocks—one of used memory and another of free memory. Figure 11.5 shows the memory map after performing compaction. Compaction may take place at the moment any node frees some memory or when a request for allocating memory fails provided the combined free space is enough to satisfy the request. Since it is expensive in terms of CPU time, it is rarely used.

**Fig. 11.5** Memory After Compaction

NOTES**Partition Selection Algorithms**

Whenever a process arrives and there are various holes large enough to accommodate it, the operating system may use one of the following algorithms to select a partition for the process.

- **First Fit:** In this algorithm, the operating system scans the free-storage list and allocates the first hole that is large enough to accommodate that process. This algorithm is fast because search is little as compared to other algorithms.
- **Best Fit:** In this algorithm, the operating system scans the free-storage list and allocates the smallest hole whose size is larger than or equal to the size of the process. Unlike first fit algorithm, it allocates a partition that is close to the size required for that process. It is slower than the first fit algorithm as it has to search the entire list every time. Moreover, it leads to more wastage of memory as it results in the smallest leftover holes that cannot be used to satisfy the memory allocation request.
- **Worst Fit:** In this algorithm, the operating system scans the free-storage list and allocates the largest hole to that process. Unlike best fit, this algorithm results in the largest leftover holes. However, simulation indicates that worst fit allocation is not very effective in reducing the wastage of memory.

Note: First fit and best fit are among the most popular algorithms for dynamic memory allocation.

Example 11.1: Consider the memory map given in Figure 11.4, how would each of the first fit, best fit and worst fit algorithms allocate memory to a process P of size 2M.

Solution: According to the different algorithms, memory will be allocated to the process P as shown in Figure 11.6.

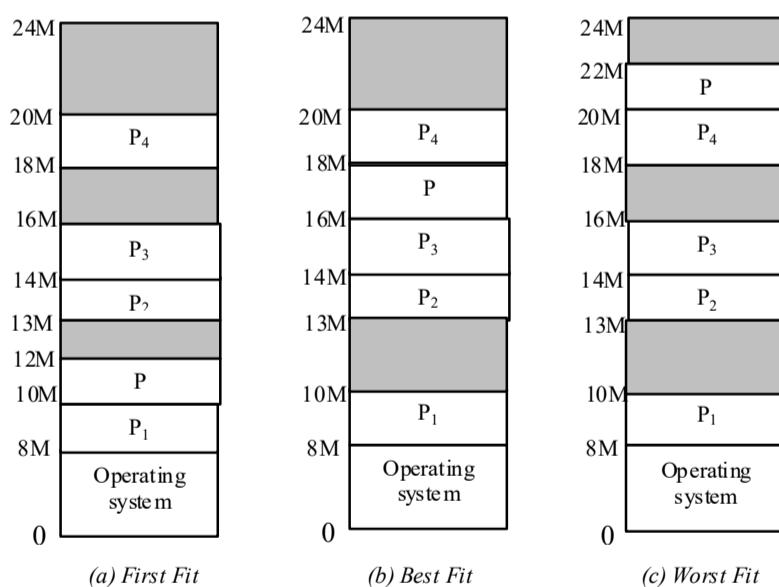


Fig. 11.6 Memory Allocation Using Different Algorithms

NOTES

Relocation and Protection

In multiprogramming environment, multiple processes are executed due to which two problems can arise which are relocation and protection.

Relocation

From the earlier discussion, it is clear that different processes run at different partitions. Now, suppose a process contains an instruction to call a procedure at absolute address 5. If this process is loaded into a partition at address 10M, this instruction will jump to absolute address 5 which is inside the operating system. Instead, it is required to call memory address (10M+5). Similarly, if the process is loaded into some other partition, say at address 20M, then it should call at memory address 20M+5. This problem is known as **relocation problem**. This relocation problem can be solved by equipping the system with a hardware register called **relocation register** which contains the starting address of the partition into which the process is to be loaded. Whenever, an address is generated during the execution of a process, the memory management unit adds the content of the relocation register to the address resulting in physical memory address.

Example 11.2: Consider the logical address of an instruction in a program is 7632 and the content of relocation register is 2500. To which location in the memory will this address be mapped?

Solution: Here, Logical address = 7632

Content of relocation register = 2500

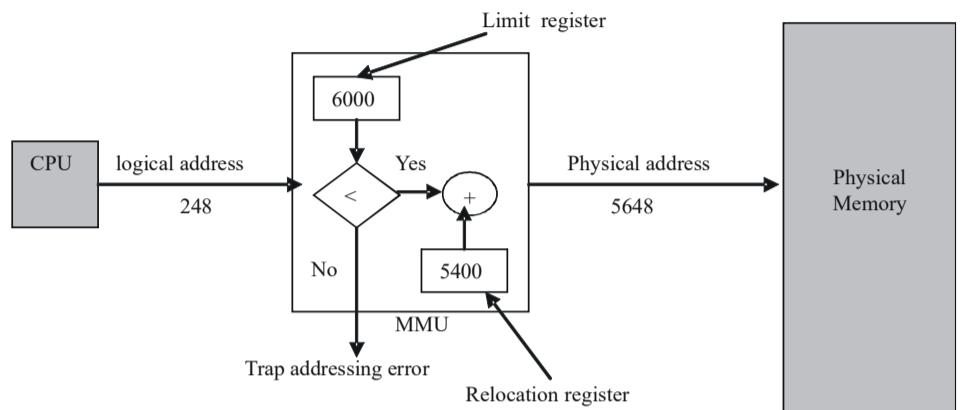
Since, Physical address = Logical address + Content of relocation register

$$\text{Physical address} = 7632 + 2500 = 10,132$$

Thus, the logical address 7632 will be mapped to the location 10,132 in memory.

Protection

Using relocation register, the problem of relocation can be solved but there is a possibility that a user process may access the memory address of other processes or the operating system. To protect the operating system from being accessed by other processes and the processes from one another, another hardware register called **limit register** is used. This register holds the range of logical addresses. Each logical address of a program is checked against this register to ensure that it does not attempt to access the memory address outside the allocated partition. Figure 11.7 shows relocation and protection mechanism using relocation and limit register.

NOTES**Fig. 11.7** Relocation and Protection using Relocation and Limit Register**Check Your Progress**

1. Give the definition of physical address and physical address space.
2. Explain the about the logical address and logical address space.
3. What is the significance of address binding?
4. Define the term Memory-Management Unit (MMU).
5. Elaborate on the Single partition?
6. What is fragmentation?

11.4 NON-CONTIGUOUS MEMORY ALLOCATION

In noncontiguous allocation approach, parts of a single process can occupy noncontiguous physical addresses. In this section, we will discuss memory management schemes based on noncontiguous allocation.

Paging

In paging, the physical memory is divided into fixed-sized blocks called **page frames** and logical memory is also divided into fixed-size blocks called **pages** which are of same size as that of page frames. When a process is to be executed, its pages can be loaded into any unallocated frames (not necessarily contiguous) from the disk. Figure 4.8 shows two processes A and B with all their pages loaded into the memory. In this figure, the page size is of 4KB. However, some systems support even larger page sizes such as 8KB, 4MB, and so on. Nowadays, the pages sizes between 4KB and 8KB are used.

NOTES

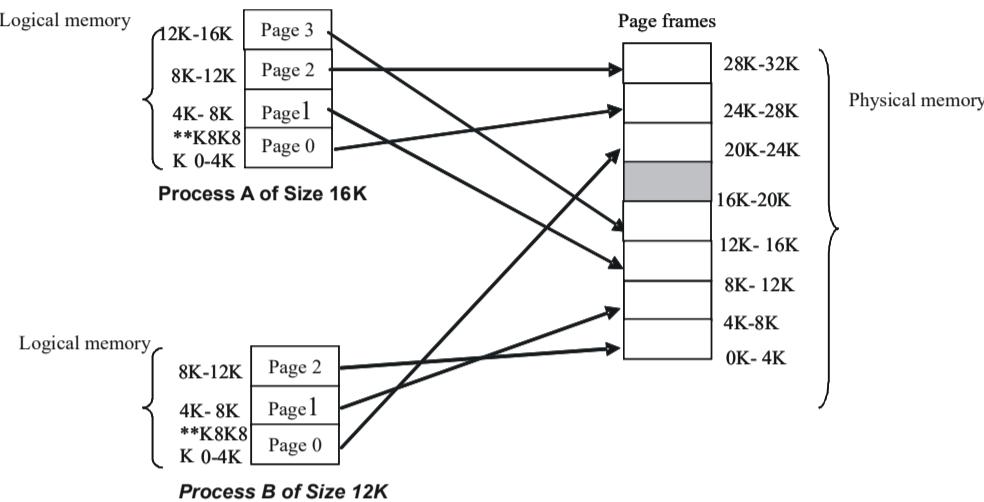


Fig. 11.8 Concept of Paging

Note: In real systems, the page size can vary from 512 bytes to 64 KB.

When the CPU generates a logical address, it is divided into two parts: A page number (p) [high-order bits] and A page offset (d) [low-order bits] where d specifies the address of the instruction within the page p. Since the logical address is a power of 2, the page size is always chosen as a power of 2 so that the logical address can be converted easily into page number and page offset. To understand this, consider the size of logical address space is 2^m . Now, if we choose a page size of 2^n (bytes or words), then n bits will specify the page offset and $m - n$ bits will specify the page number.

Example 11.3: Consider a system that generates logical address of 16 bits and page size is 4KB. How many bits would specify the page number and page offset?

Solution: Here, the logical address is of 16 bits, that is, the size of logical address space is 2^{16} ,

Page size is 4KB, that is, 4×1024 bytes = 2^{12} bytes

Thus, the page offset will be of 12 bits and page number will be of $(16 - 12) = 4$ bits.

Now let us see how a logical address is translated into a physical address. In paging, address translation is performed using a mapping table, called **page table**. The operating system maintains a page table for each process to keep track of which page frame is allocated to which page. It stores the frame number allocated to each page and the page number is used as index to the page table (Refer Figure 11.9).

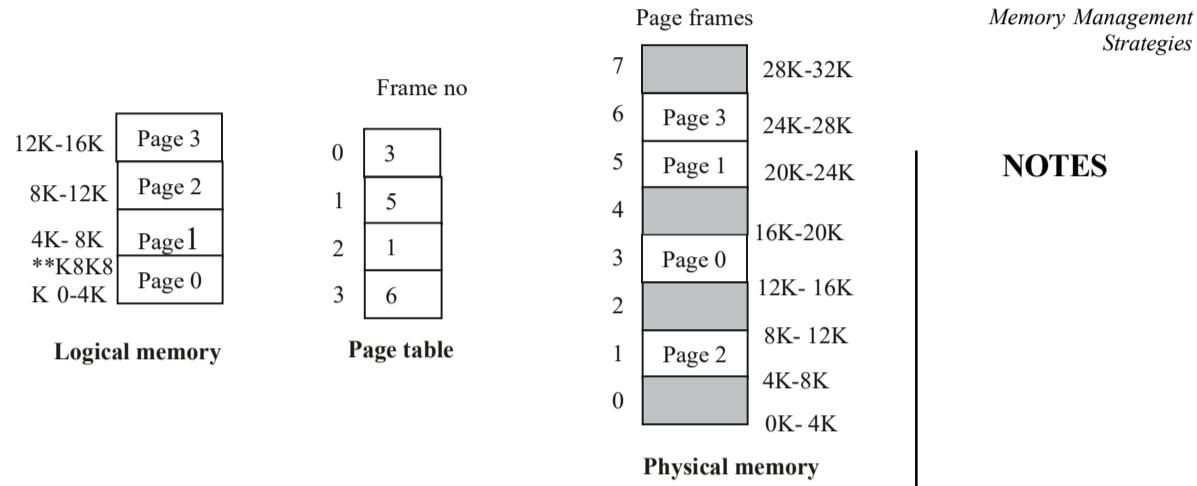


Fig. 11.9 Page Table

When CPU generates a logical address, that address is sent to MMU. The MMU uses the page number to find the corresponding page frame number in the page table. That page frame number is attached to the high-order end of the page offset to form the physical address that is sent to the memory. The mechanism of translation of logical address into physical address is shown in Figure 11.10.

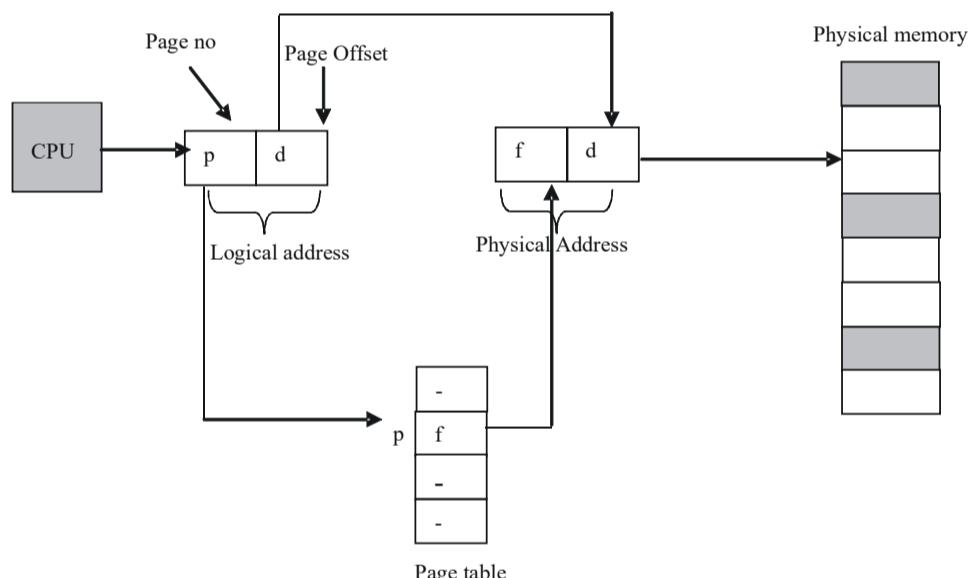


Fig. 11.10 Address Translation in Paging

Note: Since both the page and page frames are of same size, the offset within them are identical, and need not be mapped.

Example 11.4: Consider a paged memory system with eight pages of 8KB page size each and 16 page frames in memory. Using the following page table, compute the physical address for the logical address 18325.

NOTES

NOTES

7	1010
6	0100
5	0000
4	0111
3	1101
2	1011
1	1110
0	0101

Page Table

Solution: Since, total number of pages = 8, that is, 2^3 and each page size = 8KB, that is, 2^{13} bytes, the logical address will be of 16 bits. Out of these 16 bits, the three high-end order bits represent the page number and the 13 low-end order bits represent offset within the page. In addition, there are 16, that is, 2^4 page frames in memory, thus, the physical address will be of 17 bits.

Given logical address = 18325 which is equivalent to 010001110010101. In this address, page number = 010, that is, 2 and page offset = 001110010101. From the page table it is clear that the page number 2 is in page frame 1011.

Therefore, the physical address = 1011001110010101, which is equivalent to 92053.

Advantages

- Since the memory allocated is always in fixed unit, any free frame can be allocated to a process. Thus, there is no external fragmentation.

Disadvantages

- There may be some internal fragmentation. To illustrate this, consider a page size is of 4KB and a process requires memory of 8195 bytes, that is 2 page + 3 bytes. In this case, for only 3 bytes, an entire frame is wasted resulting in internal fragmentation.

Hardware Implementation of Page Table

A page table can be implemented in several ways. The simplest way is to use registers to store the page table entries indexed by page number. Though this method is faster and does not require any memory reference, its disadvantage is that it is not feasible in case of large page table as registers are expensive. Moreover, at every context switch, the page table needs to be changed which in turn requires all the registers to be reloaded. This degrades the performance.

NOTES

Another way is to keep the entire page table in main memory and the pointer to page table stored in a register called **Page-Table Base Register (PTBR)**. Using this method, page table can be changed by reloading only one register, thus reduces context switch time to a great extent. The disadvantage of this scheme is that it requires two memory references to access a memory location; first, to access page table using PTBR to find the page frame number, and second, to access the desired memory location. Thus, memory accessing is slowed down by a factor of two.

To overcome this problem, the system can be equipped with a special hardware device known as **Translation Look-Aside Buffer (TLB)** (or **associative memory**). The TLB is inside MMU and contains a limited number of page table entries. When CPU generates a logical address and presents it to the MMU, it is compared with the page numbers present in the TLB. If a match is found in TLB (called **TLB hit**), the corresponding page frame number is used to access the physical memory. In case a match is not found in TLB (called **TLB miss**), memory is referenced for the page table. Further, this page number and the corresponding frame number are added to the TLB so that next time if this page is required, it can be referenced quickly. Since the size of TLB is limited so when it is full, one entry needs to be replaced. Figure 11.11 shows the mechanism of paging using TLB.

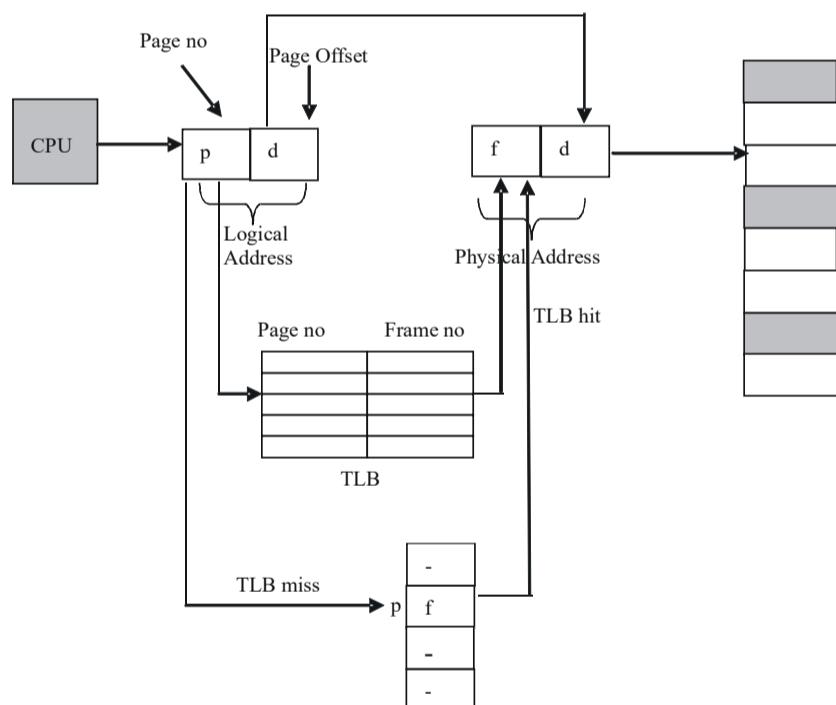


Fig. 11.11 Paging with TLB

TLB can contain entries for more than one process at the same time, so there is a possibility that two processes map the same page number to different frames. To resolve this ambiguity, a Process Identifier (PID) can be added with each entry of

NOTES

TLB. For each memory access, the PID present in the TLB is matched with the value in a special register that holds the PID of the currently executing process. If it matches, the page number is searched to find the page frame number; otherwise it is treated as a TLB miss.

Segmentation

A user views a program as a collection of segments such as main program, routines, variables, and so on. All of these segments are variable in size and their size varies during execution. Each segment is identified by a name (or segment number) and the elements within a segment are identified by their offset from the starting of the segment. Figure 11.12 shows the user view of a program.

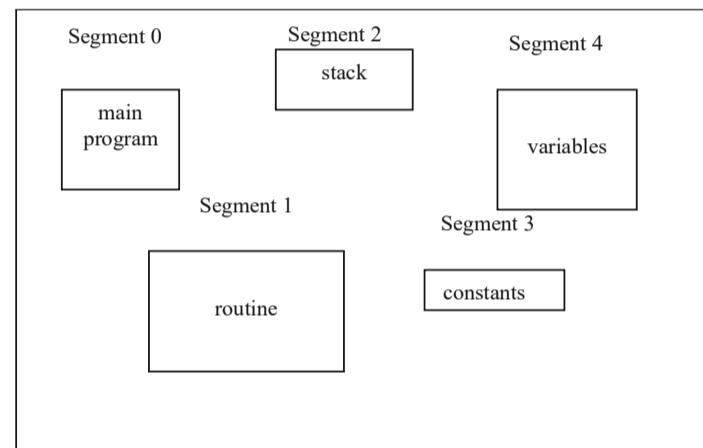
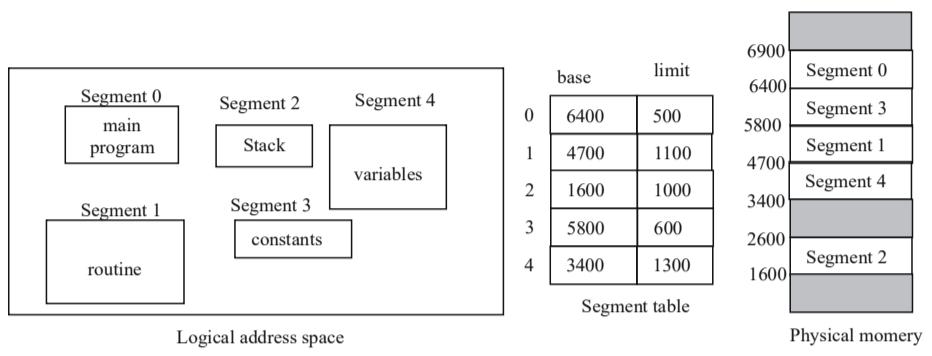


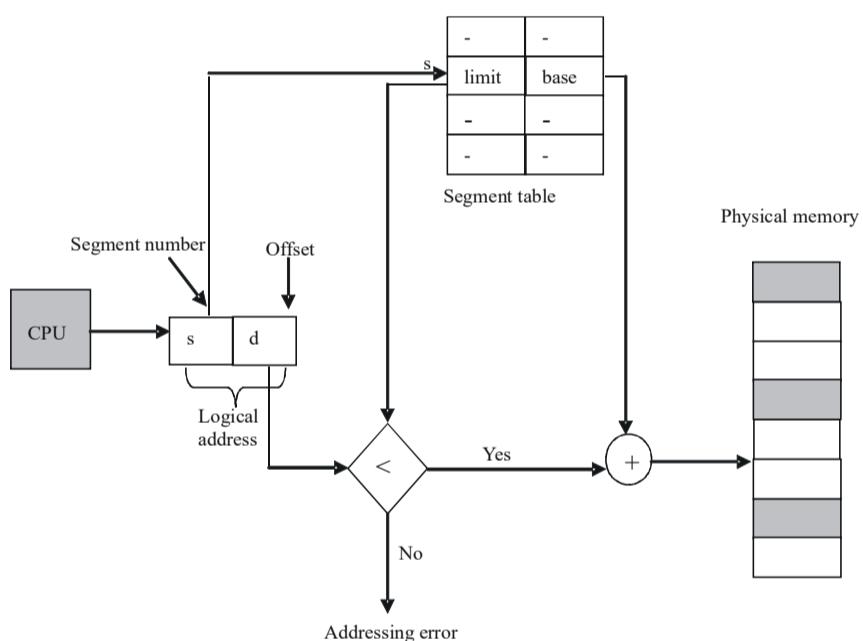
Fig. 11.12 User View of a Program

Segmentation is a memory management scheme that implements the user view of a program. In this scheme, the entire logical address space is considered as a collection of segments with each segment having a number and a length. The length of a segment may range from 0 to some maximum value as specified by the hardware and may also change during the execution. The user specifies each logical address consisting of a segment number (s) and an offset (d). This differentiates segmentation from paging in which the division of logical address into page number and page offset is performed by the hardware.

To keep track of each segment, a **segment table** is maintained by the operating system (Refer Figure 11.13). Each entry in the segment table consists of two fields: segment base and segment limit. The **segment base** specifies the starting address of the segment in physical memory and the **segment limit** specifies the length of the segment. The segment number is used as an index to the segment table.

NOTES

Fig. 4.13 Segment Table

When CPU generates a logical address, that address is sent to MMU. The MMU uses the segment number of logical address as an index to the segment table. The offset is compared with the segment limit and if it is greater, invalid-address error is generated. Otherwise, the offset is added to the segment base to form the physical address that is sent to the memory. Figure 11.14 shows the hardware to translate logical address into physical address in segmentation.


Fig. 11.14 Segmentation Hardware
Advantages

- Since a segment contains one type of object, each segment can have different types of protection. For example, a procedure can be specified as execute only whereas a char type array can be specified as read only.

NOTES

- It allows sharing of data or code between several processes. For example, a common function or shared library can be shared between various processes. Instead of having them in address space of every process, they can be put in one segment and that segment can be shared.

Example 4.5: Using the following segment table, compute the physical address for the logical address consisting of segment and offset as given below.

	Base	Limit
0	5432	350
1	115	100
2	2200	780
3	4235	1100
4	1650	400

Segment Table

- (a) Segment 2 and offset 247
(b) Segment 4 and offset 439

Solution:

- (a) Here, offset = 247 and segment is 2

It is clear from the segment table that limit of segment 2 = 780 and segment base = 2200

Since, the offset is less than the segment limit, physical address is computed as

$$\begin{aligned}\text{physical address} &= \text{offset} + \text{segment base} \\ &= 247 + 2200 = 2447\end{aligned}$$

- (b) Here, offset = 439 and segment is 4

It is clear from the segment table that limit of segment 4 = 400 and segment base = 1650

Since, the offset is greater than the segment limit, invalid-address error is generated.

Segmentation with Paging

The idea behind the segmentation with paging is to combine the advantages of both paging, (such as uniform page size) and segmentation, (such as protection and sharing) together into a single scheme. In this scheme, each segment is divided into a number of pages. To keep track of these pages, a page table is maintained for each segment. The segment offset in the logical address (comprising segment number and offset) is further divided into a page number and a page offset. Each entry of segment table contains the segment base, segment limit and one more entry that contains the address of the segment's page table.

The logical address consists of three parts: Segment number (s), Page number (p) and Page offset (d). Whenever, address translation is to be performed, firstly, the MMU uses the segment number as an index to segment table to find the address of

NOTES

page table. Then the page number of logical address is attached to the high-order end of the page table address and used as an index to page table to find the page table entry. Finally, the physical address is formed by attaching the frame number obtained from the page table entry to the high-order end of the page offset. Figure 11.15 shows the address translation in segmentation with paging scheme.

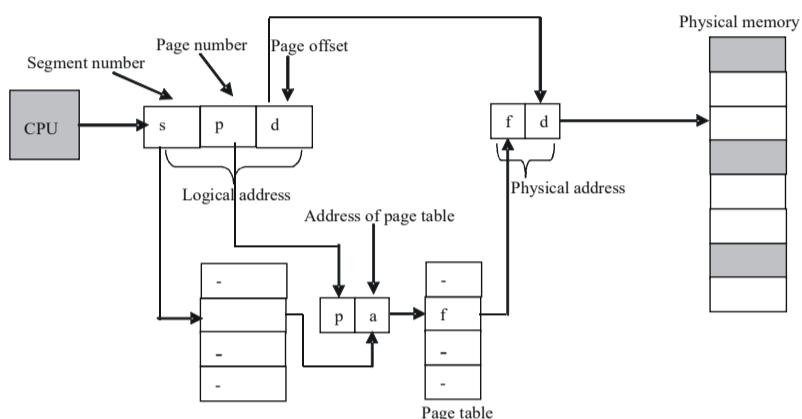


Fig. 4.15 Segmentation with Paging

Check Your Progress

7. What is the basic function of paging?
8. Which of the fundamental approaches used to implement non-contiguous memory allocation?
9. What is the best page size when designing an operating system?
10. Explain the term of translation look-aside buffer.
11. Write down the advantage of segmentation.

11.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Every byte in the memory has a specific address that may range from 0 to some maximum value as defined by the hardware. This address is known as **physical address**. Whenever, a program is brought into the main memory for execution, it occupies certain number of memory locations. The set of all physical addresses used by the program is known as **physical address space**.
2. A program is compiled to run starting from some fixed address and accordingly all the variables and procedures used in the source program are assigned some specific addresses known as **logical addresses**. Thus,

NOTES

in machine code, all references to data or code are made by specifying the logical addresses and not by the variable or procedure names, and so on. The range of addresses that user programs can use is system-defined and the set of all logical addresses used by a user program is known as its **logical address space**.

3. When a user program is brought into main memory for execution, its logical addresses must be mapped to physical addresses. This mapping from addresses associated with a program to memory addresses is known as **address binding**.
4. The run time address binding is performed by the hardware device known as Memory-Management Unit (MMU).
5. The relocation-register scheme of single partition allocation is used to shield user processes from each other and from modifying the code and data of the operating system. The relocation register contains the lowest physical address value, while the boundary register contains a number of logical addresses.
6. Fragmentation is memory wasted. It can be internal we are dealing with system that have fixed-size allocation units, or external if we are dealing with system that have variable allocation unit.
7. Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous. It avoids the considerable problem of having to fit varied sized memory chunks onto the backing store.
8. Paging and segment.
9. The best paging size varies from system to system, so there is no single best when it comes to page size. There are different factors to consider in order to come up with a suitable page size, such as page table, paging time, and its effect on the overall efficiency of the operating system.
10. A Translation Look Aside Buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location. If the requested address is not in the TLB, it is a miss, and the translation proceeds by looking up the page table in a process called a page walk.
11. Add Answers to check Your progress questions(11)from hardcopy

11.6 SUMMARY

- Many systems allow multiple processes to reside in the main memory at the same time, to increase the CPU utilization. It is the job of memory manager, a part of the operating system, to manage memory between these processes in an efficient way.

NOTES

- Every byte in the memory has a specific address that may range from 0 to some maximum value as defined by the hardware. This address is known as physical address.
- A program is compiled to run starting from some fixed address and accordingly all the variables and procedures used in the source program are assigned some specific address known as logical address.
- The mapping from addresses associated with a program to memory addresses is known as address binding. The addresses binding can take place at compilation time, load time or run time.
- Address binding takes place at compile time if it is known which addresses the program will occupy in the main memory at that time. In this case, the program generates absolute code at the compile time only. That is, logical addresses are same as that of physical addresses.
- Address binding occurs at load time if it is not known at compile time which addresses the program will occupy in the main memory. In this case, the program generates relocatable code at the compile time which is then converted into the absolute code at the load time.
- The run time address binding is performed by the hardware device known as Memory-Management Unit (MMU).
- One of the simplest ways to manage memory is to partition the main memory into two parts.
- A single partition scheme restricts the system to have only one process in memory at a time that reduces utilization of the CPU as well as of memory.
- Fragmentation is memory wasted. It can be internal we are dealing with system that have fixed-size allocation units, or external if we are dealing with system that have variable allocation unit.
- Each partition is of fixed size and can contain only one process. There are two alternatives for this technique—equal-sized partition and unequal sized partition.
- For managing the memory, the memory manager may use one strategy from a number of available memory management strategies.
- There is one problem with this method that is the memory utilization is not efficient. Any process regardless of how small it is, occupies an entire partition which leads to the wastage of memory within the partition. This phenomenon which results in the wastage of memory within the partition is called internal fragmentation.
- The number of processes in memory depends on the number of partitions. Thus, the degree of multiprogramming is limited.
- The memory cannot be used efficiently in case of small-sized processes.

NOTES

- To overcome the disadvantages of fixed-partitions technique, a technique called Multiprogramming with a Variable number of Tasks (MVT) is used.
- When there is no process in the memory, the whole memory is available for allocation and is considered as a single large partition of available memory (a hole).
- All the memory management strategies allocate memory to the processes using either of two approaches: contiguous memory allocation or non-contiguous memory allocation.
- In contiguous memory allocation, each process is allocated a single contiguous part of the memory. The different memory management schemes that are based on this approach are single partition and multiple partitions.
- In the single partition technique, main memory is partitioned into two parts. One of them is permanently allocated to the operating system while the other part is allocated to the user process.
- Whenever a process arrives and there are various holes large enough to accommodate it, the operating system may use one of the algorithms to select a partition for the process: first fit, best fit and worst fit.
- In multiprogramming environment, multiple processes are executed due to which two problems can arise which are relocation and protection.
- The relocation problem can be solved by equipping the system with a hardware register called relocation register which contains the starting address of the partition into which the process is to be loaded.
- To protect the operating system from access by other processes and the processes from one another, another hardware register called limit register is used.
- In non-contiguous allocation approach, parts of a single process can occupy non-contiguous physical addresses.
- Paging and segmentation are the memory management techniques based on the non-contiguous allocation approach.
- In paging, the physical memory is divided into fixed-sized blocks called page frames and logical memory is also divided into fixed-size blocks called pages which are of same size as that of page frames. The address translation is performed using a mapping table, called page table.
- In multiprogramming environment, multiple processes are executed due to which two problems can arise which are relocation and protection.
- Since the memory allocated is always in fixed unit, any free frame can be allocated to a process. Thus, there is no external fragmentation.

NOTES

- Segmentation is a memory management scheme that implements the user view of a program. In this scheme, the entire logical address space is considered as a collection of segments with each segment having a number and a length. To keep track of each segment, a segment table is maintained by the operating system.
- The idea behind the segmentation with paging is to combine the advantages of both paging, (such as uniform page size) and segmentation, (such as protection and sharing) together into a single scheme. In this scheme, each segment is divided into number of pages. To keep track of these pages, a page table is maintained for each segment.
- A Translation Look Aside Buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location. If the requested address is not in the TLB, it is a miss, and the translation proceeds by looking up the page table in a process called a page walk.

11.7 KEY WORDS

- **Address binding:** It is the process of mapping addresses associated with a program to memory addresses.
- **Memory-Management Unit (MMU):** The run time address binding is performed by the hardware device known as Memory-Management Unit (MMU).
- **Translation Look Aside Buffer (TLB):** A Translation Look Aside Buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location. If the requested address is not in the TLB, it is a miss, and the translation proceeds by looking up the page table in a process called a page walk.
- **Paging:** A process in which the physical memory is divided into fixed-sized blocks called page frames and logical memory is also divided into fixed-size blocks called pages which are the same size as the page frames. Address translation is done using a mapping table, called page table.
- **Segmentation:** A memory management scheme that implements the user view of a program, in which the entire logical address space is considered as a collection of segments with each segment having a number and a length.
- **Segmentation with paging:** It is a combination of the advantages of both paging, (such as uniform page size) and segmentation, (such as protection and sharing) together into a single scheme. In this scheme, each segment is divided into number of pages. To keep track of these pages, a page table is maintained for each segment.

NOTES

- **Virtual memory:** A technique that allows execution of a program that is bigger than the physical memory of the computer system.

11.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What is address binding?
2. State the main difference between logical and physical space.
3. Explain the term of single partition.
4. State and explain Storage management strategies.
5. Contiguous memory allocation faces which type of problem?
6. Difference between relocation and protection.
7. Write the advantage and disadvantage for paging.
8. State about translation look-aside buffer.
9. Define the term of segmentation with paging.
10. How does swapping result in better memory management?
11. Write short notes on:
 - (a) Fixed partitions
 - (b) Paging table
 - (c) Segmentation

Long-Answer Questions

1. Discuss briefly about the address binding? When does it take place?
2. Explain briefly about the different memory allocation?
3. Discuss partition selection algorithm in brief.
4. Consider a system that generates logical address of 16 bits and page size is 4KB. How many bits would specify the page number and page offset?
5. Can a process on a paged memory system access memory allocated to some other process? Why or why not?
6. How to hardware implementation of paging with translation look-aside buffer?
7. What are the two major differences between segmentation and paging?

NOTES

8. How does the segmentation scheme allow different processes to share data or code?
9. Using the following segment table, compute the physical address for the logical address consisting of segment and offset as given below.

Base	Limit
5432	350
115	100
2200	780
4235	1100
1650	400

Segment table

- (a) Segment 0 and offset 193
 - (b) Segment 2 and offset 546
 - (c) Segment 3 and offset 1265
10. Consider a paged memory system with 16 pages of 2048 bytes each in logical memory and 32 frames in physical memory. How many bits will each of the following comprise?
 - a. Logical Address
 - b. Page Number
 - c. Page Offset
 - d. Physical Address
 11. Which of the following memory management schemes suffer from internal or external fragmentation?
 - a. Multiple fixed-partition
 - b. Multiple variable-partition
 - c. Paging
 - d. Segmentation

11.9 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.

NOTES

- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.
- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.
- Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.
- Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.
- Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

NOTES

UNIT 12 VIRTUAL MEMORY MANAGEMENT

Structure

- 12.0 Introduction
- 12.1 Objectives
- 12.2 Allocation
- 12.3 Address Binding, Relocation and Protection
- 12.4 Swapping
- 12.5 Segmentation
- 12.6 Virtual Memory
 - 12.6.1 Basic Concepts
- 12.7 Logical Versus Physical Address Space
- 12.8 Paging
 - 12.8.1 Page Size and Page Realease
- 12.9 Page Table and Its Entries
- 12.10 Page Fault
 - 12.10.1 Demand Paging
 - 12.10.2 Multilevel Page Table
- 12.11 Page Replacement
- 12.12 Answers to Check Your Progress Questions
- 12.13 Summary
- 12.14 Key Words
- 12.15 Self Assessment Questions and Exercises
- 12.16 Further Readings

12.0 INTRODUCTION

Virtual memory, or virtual storage is a memory management technique that provides an idealized abstraction of the storage resources that are actually available on a given machine which “Creates the Illusion to Users of a Very Large Memory”.

Using a combination of hardware and software, the machine's operating system maps memory addresses used by a programme, called virtual addresses, into physical addresses in computer memory. Main storage occurs as a contiguous address space or array of contiguous segments, as seen by a process. Virtual address spaces are handled by the operating system and actual memory is allocated to virtual memory. In the CPU, also referred to as a memory management unit (MMU), address translation hardware automatically converts virtual addresses to physical addresses.

A memory allocation is a vital aspect of the multiprogramming. Memory space allocation assumes that a process is loaded into a contiguous section of memory. The process of adjusting the absolute memory reference addresses in the executable program is called relocation. Swapping is a simple memory

NOTES

management technique used by the Operating System (OS) to increase the utilization of the processor by moving some blocked process from the main memory to the secondary memory; thus forming a queue of temporarily suspended process and the execution continues with the newly arrived process.

You will also learn that segmentation is one of the most common ways to achieve memory protection; another common one is paging. Virtual memory is an integral part of the computer architecture; all implementations require hardware support, typically in the form of a memory management unit built into the Central Processing Unit (CPU). Logical or virtual addresses are those used while generating executable files by the linker. Physical addresses are the actual addresses in the main memory where the program is loaded, and the physical address space is the set of addresses used by the program when loaded into the main memory.

In computer operating systems, paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory. In this scheme, the operating system retrieves data from secondary storage in same-size blocks called pages.

In this unit, you will study about the concept of virtual memory management, page replacement, strategies, page fault frequency and page size.

12.1 OBJECTIVES

After going through this unit, you will be able to:

- Know about the virtual memory
- Explain the concept of address binding, relocation and protection
- Describe the process of swapping and segmentation
- Understand the basic concepts of virtual memory
- Explain the difference between virtual and physical addresses
- Elaborate on the concept of paging
- Explain the terms page table , page faults and page size
- Analyse the page replacement

12.2 ALLOCATION

Virtual memory is a method for memory management where it is possible to use secondary memory as though it were a part of the main memory. A very common technique used in the operating systems (OS) of computers is virtual memory.

Virtual memory uses hardware and software to allow a device to compensate for physical memory shortages, by temporarily moving data from random access memory (RAM) to disc storage. Virtual memory, in turn, enables a machine to handle secondary memory as if it were the main memory.

Various techniques of memory allocation are available for multiprogramming are discussed below:

1. Fixed Partitions

With the mono-programming model, only one program can reside in the memory at any given time. With the introduction of swapping techniques, it is possible to write out the current program that is waiting for Input/Output (I/O) to the backing store, and read in a new program and allocate the Control Processing Unit (CPU) to it. However, it is still not possible to make the CPU busy with two programs since I/O is about 1000 times slower than the CPU. To accommodate more programs simultaneously in the memory, the memory can be partitioned so that it is possible to load as many programs as the number of partitions. Two fixed (fixed in the sense that the size cannot be changed dynamically during the execution of the OS) partition schemes are shown in Figure 12.1. In one scheme, known as equal size partitions, all the partitions are of the same size. So, jobs of size larger than the partition size cannot be loaded and executed. If the job size is smaller, a second job cannot be loaded into a partition simultaneously with the first one, resulting in unutilized memory space. This is called internal fragmentation loss, because it is internal to the allocated partition. In the other scheme, known as variable size fixed partitions, the partition sizes are different. So, jobs of different sizes can be loaded and executed. The size of the largest job that can be executed will be the size of the largest partition. This scheme, as in the case of the equal size partitions, also suffers from internal fragmentation loss.

NOTES

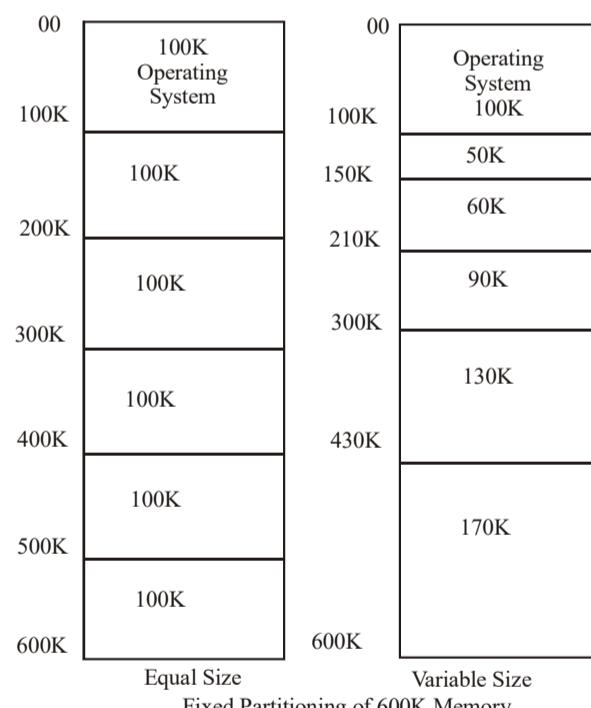


Fig. 12.1 Fixed Partition Memory Allocation

NOTES

2. Placement Algorithms

Placement algorithms are concerned with the allocation of a memory partition for execution of a job. A variety of allocation strategies (placement of processes in the free space) is possible. Given below are the popular Best Fit, First Fit, and Worst Fit techniques:

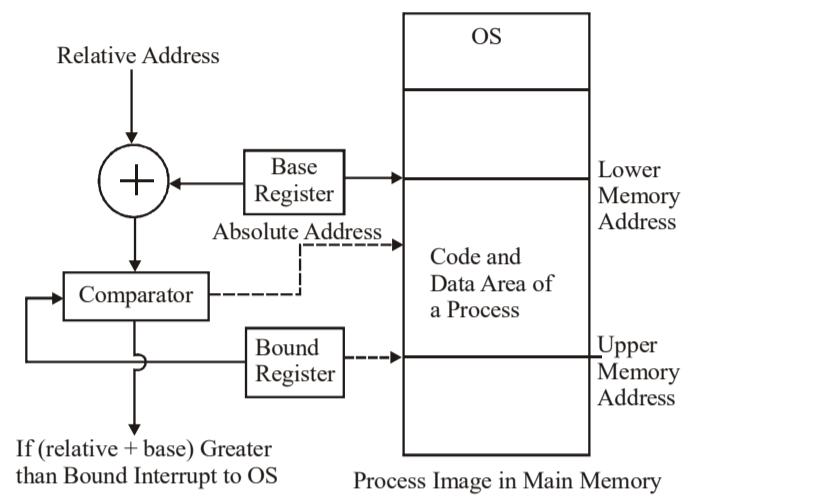
Best Fit: The memory allocator places a process in the smallest block, among the free memory blocks, in which it will fit. Suppose a process requests 40MB of memory and the memory manager currently has a list of free blocks of 64MB, 38MB, 44MB, 128MB, and 43MB blocks. The best fit strategy will allocate 40MB from the 43MB block to the process, leaving a new free block of 3MB. The drawback is that this approach creates smaller size free blocks. Normally no process will fit in such blocks, and hence they will be wasted.

Worst Fit: This technique creates and allocates a memory partition, from the largest available block whose size is the same as that of the new process to be loaded. This technique thus leave larger free blocks than other techniques and hence another process may be loaded in that newly created free block whenever required. Now, with the same scenario of free blocks, a 40MB process will be allocated 40MB from the 128MB free block leaving a large 88MB free block, where a new process may be allocated.

First Fit: The techniques (both Best Fit and Worst Fit) illustrated above need to search among the many free blocks to find an appropriate block for partitioning and allocation. This is a time-consuming process. The First Fit algorithm uses a faster technique which scans the free blocks until it finds a block whose size is big enough to accommodate the new process. So, in the above list of free blocks, First Fit finds the 64MB free block for allocating 40MB for the new process, leaving a 24MB free block. First Fit, in general, produces intermediate free block between those created by the Best Fit and Worst Fit algorithms.

Both the Best Fit and First Fit techniques produce small pieces of memory blocks that are smaller than the normal size of processes.

If all the partitions are of equal size, a job can be loaded in any of the partitions, if its size fits in; otherwise, it cannot be executed. In the case of variable sized partitions, there are options. In one approach, each partition maintains a queue of jobs whose sizes are nearest or equal to its size. So, there will be as many queues of jobs as there are partitions. Jobs in each queue may be taken for execution based on the First Come, First Serve (FCFS) or priority techniques. In another approach, all the jobs are placed in a single queue and the allocation of partition may be done by the Best Fit technique applied on the available partition(s).

NOTES**Fig. 12.2** Hardware Support for Relocation

The linker assembles the code starting from address zero and going up to the size of the program. If there is more than one segment in the program, each of these starts from zero address. That is, the logical or virtual address starts from zero and the absolute addresses within the program are relative to zero. These program segments will be loaded for execution starting from some non-zero addresses. So, it is necessary to relocate the addresses suitable for the new start addresses. This is done by adding the relative address to the new start address of the program/segments before accessing the memory. Moreover, it is necessary to check that the addresses generated are within the address space of the program. This is done by verifying that the relative addresses are less than the program/segment length. If a relative address is not less than the program/segment length, an exception error interrupt is raised to transfer control to the Operating System (OS) which then terminates program execution. The base and bound register and a hardware comparison mechanism shown in Figure 12.2 execute the relocation and memory protection technique just explained. The start address of the program/segment is loaded in the base register. The length of the program/segment is loaded in the bound register. Every relative address is checked with the bound register content to ensure that it is less than the program length. It is then added to the base register to get the absolute address for accessing the physical memory. When the OS prepares to run a process on the CPU, it resets the base register with the start address and the bound register with the program size for the particular process. The address spaces of different users and the OS are now easily protected. A process, even a malfunctioning one, cannot make references to the memory locations of another process as these locations are not in the address space bracketed by the base and bound registers. Any attempt to make such memory references will lead to the generation of error exception interrupts by the memory management hardware to terminate the process.

NOTES

3. Dynamic Partitions

Fixed partitions suffer from internal fragmentation causing wastage of free memory blocks. Also, a program that is greater than the largest sized partition cannot be executed. In order to overcome both of these drawbacks, a technique was developed for partitioning the physical memory as and when required. The physical memory of the computer is dynamically portioned and allocated to a program at the time of execution. In this way, programs as large as the total physical memory can be loaded (preempting all the other processes, of course) and executed. Moreover, the partition size can be exactly the same as the process size, thus eliminating internal fragmentation. When the process terminates, the memory is deallocated, forming a free block whose size is same as the deallocated process size. Various placement techniques, such as Best Fit, Worst Fit and First Fit are possible.

Figure 12.3 illustrates the scenarios of memory allocation and deallocation: (b)–(e) shows the allocation of memory to four processes – 1, 2, 3 and 4; (f) shows the deallocation of memory following the termination of process 1; (g) shows the allocation of memory for process 5; (h) shows the deallocation following termination of process 3; and (i) finally allocating memory for process 6.

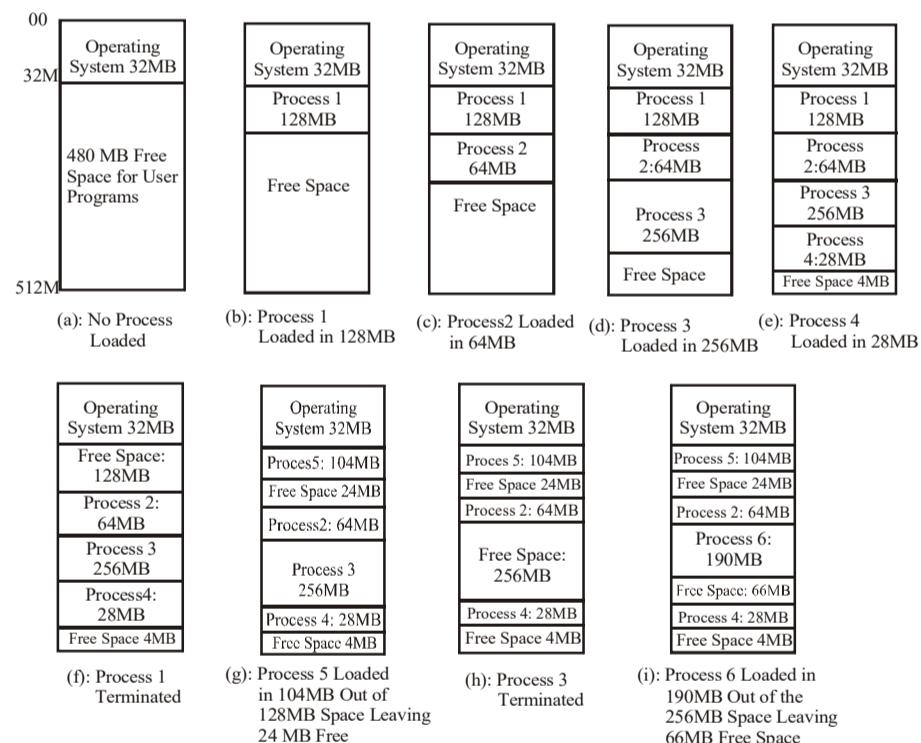


Fig. 12.3 (a) – (i): Scenarios of Memory Allocations and Deallocation in Dynamic Partition Technique

NOTES**4. External Fragmentation**

When programs terminate, memory is deallocated and this memory reallocated for new programs. Therefore, many number of free memory regions are formed between two allocated regions or at the upper end of the physical memory. The sizes of these free blocks will keep decreasing as the number of free blocks increases with increasing deallocation and reallocation of memory. At some point, the size of the blocks become too small to accommodate a new program. However, the total amount of free memory (summing up all the free blocks) may be greater than that required for loading a program. This situation is called external fragmentation. This is a drawback of dynamic partition allocation. The problem of external fragmentation is severe with the Best Fit and First Fit memory allocation techniques of process placement. A solution to this is the technique of memory compaction discussed below.

5. Compaction or Defragmentation

Compaction is the process of moving all the scattered free regions of the physical memory to the upper end or to a single region of the physical memory so that all the free blocks may be combined to form a large free block. This process of moving free blocks involves the moving of allocated blocks also. In Figure 12.3(i), there are three free blocks of sizes 24MB, 66MB and 4MB. To move these free blocks to the upper (higher) address part of the memory, one way is to, move processes 4, 2 and 6 to lower address regions forming free areas in the upper address region as shown in Figure 12.4.

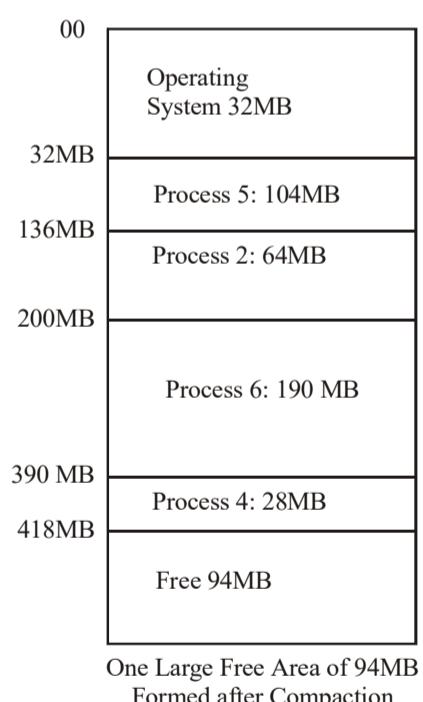


Fig. 12.4 Compaction of Free Blocks of Fig. 12.3(i)

NOTES

6. Managing Free Memory

It is important to keep track of allocated and free blocks of memory by using some efficient technique. We can do so using any of the following two methods:

- Memory management with bit-maps
- Memory management with linked list

Memory Management with Bit Maps

Even though memory size as low as one byte can be allocated, a process size will be normally very large and hence a certain, minimum size, called basic allocation unit, is fixed. An allocation unit of the main memory for processes may be from a few bytes to many KiloBytes (KB). If the basic allocation unit is 1KB and if there is 1000MB of physical memory, there will be 10,00,000 allocation units. When using bit-maps for keeping track of memory, the value (0 or 1) of one bit of memory indicates whether an allocation unit is free or has been allocated to a process as shown in Figure 12.5. So, one megabit is needed to for the bitmaps. Smaller, size of allocation units larger is the amount of memory required for bitmaps. In Figure 12.5, the state of the memory when allocated to five processes P1,P2,P3,P4 and P5 are shown. The sizes of the processes are 4, 6, 3, 1 and 5 allocation units, respectively. There are free memory areas between two process areas. Two process areas may be adjacent; that is, there need not be any free area between them.

Even though bitmaps are a simple technique for keeping track of physical memory, it is difficult to search in the bitmap for a contiguous memory block of, say k allocation units needed for loading a new process. One has to look in the large bitmap for k consecutive zeros, which is a time consuming task.

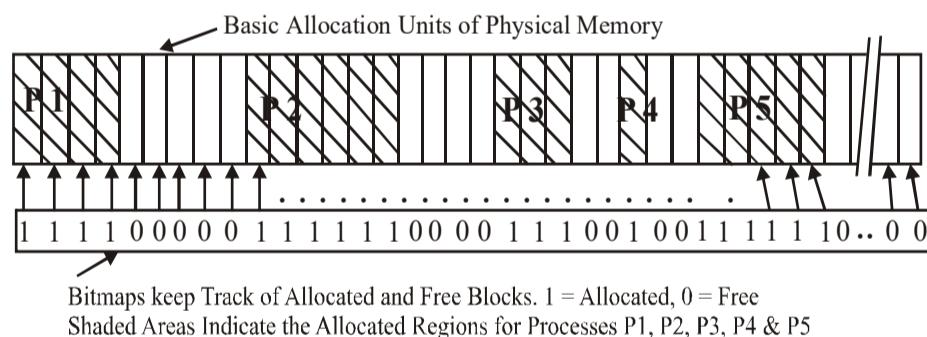


Fig. 12.5 Use of Bit-Map for keeping Track of Allocated and Free Physical Memory Areas

Memory Management with Linked List

Virtual Memory
Management

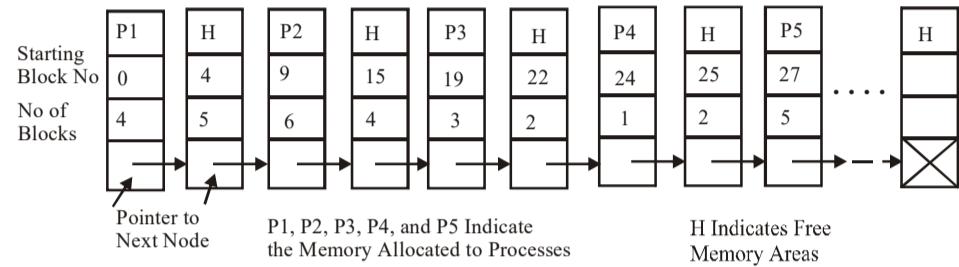


Fig. 12.6 Use of Linked List for keeping Track of Allocated and Free Physical Memory Area

Figure 12.6 illustrates the Linked list technique for keeping track of allocated and free memory areas. Every node uses 4 fields—one to indicate whether the block is free or a process is loaded; the second to hold the starting allocation unit number; the third to hold the number of units; and the fourth is a pointer to the next node of the linked list. H stands for holes or free areas that may be allocated to processes when requested by the process manager. The advantages of this representation is that the nodes of the lists are in the order of their starting address of memory areas and their sizes are available in the node itself. So, searching for contiguous areas is easier as compared to bit-maps. Also, when processes terminate, it is easy to find and merge with the adjacent free areas.

Various search techniques like First Fit, Best Fit and Worst Fit (explained earlier) may be used to find free memory areas of suitable size for loading new processes. A variant of First Fit called Next Fit may also be used for a slightly faster search of free memory areas. It works the same way as First Fit, except that it keeps track of the location where it found a suitable block, and when it searches the next time, the search starts from the location next to the last found location. First Fit, Best Fit and Worst Fit start searching from the beginning and hence are slower than Next Fit.

Buddy System

The amount of memory that is allocated by the buddy system is always a power of 2; that is, the allocatable sizes are 2^k , where k is an integer whose maximum value depends on the memory available and the minimum value is decided by the operating system. For example, with 512MB of memory available the upper limit of k is 29 (Refer Figure 12.7(a)). The minimum value might be 10 to keep the allocatable size always larger than 1 KB. Smaller blocks lead to high overheads for keeping track of allocated and unallocated blocks of memory. Now, we will see how the buddy system allocates memory by analysing some examples.

NOTES

NOTES

Process 1 requests 75MB of memory. The buddy system memory manager divides the 512 MB memory into two halves of 256 MB each. It then takes the left 256MB block and again divides into two halves of 128MB each. As 128MB is the smallest size that is larger than or equal to 75MB, this block (128MB) is allocated for process 1 to satisfy the request, as shown in Figure 12.7(b)

Similarly, for a request for 48MB from process 2, the system divides the other 128MB into two 64MB blocks, and allocates the left 64MB block (refer Figure 12.7(c)). For a request for 180MB from process 3, the buddy system allocates the left 256MB block as shown in Figure 12.7(d) for a request for 40MB from process 4, the system allocates the left 64MB block (Refer Figure 12.7(e)).

Now, how does the buddy system release the memory when the processes terminate? When process 1 terminates, it deallocates 128MB memory occupied by the 75 MB-size process 1 (Refer Figure 12.7(f)). If there is an adjacent block of the same size already free, this newly released free block will be merged with that and form a larger block of twice the size. This merging process will be repeated as long as a free adjacent block of the same size exists. In the present case there is no such situation. In Figure 12.7(g), a further request for 120MB by process 5 is met by allocating the newly freed 128MB block.

The problem of external fragmentation is minimum in the Buddy algorithm when compared to other techniques like dynamic allocation. Only a little overhead is needed in this system to free the allocated memory and to compact the free blocks.

The Buddy system, however, suffers from internal fragmentation as the memory size requested is almost always not a power of 2. For instance, out of the 128MB allocated for process 1, only 75MB is used. The remaining 53MB is wasted as internal fragmentation. In each partition, the internal fragmentation loss is roughly zero to as much as one half of the partition size. So, there can be, on the average, about 25 per cent loss due to internal fragmentation.

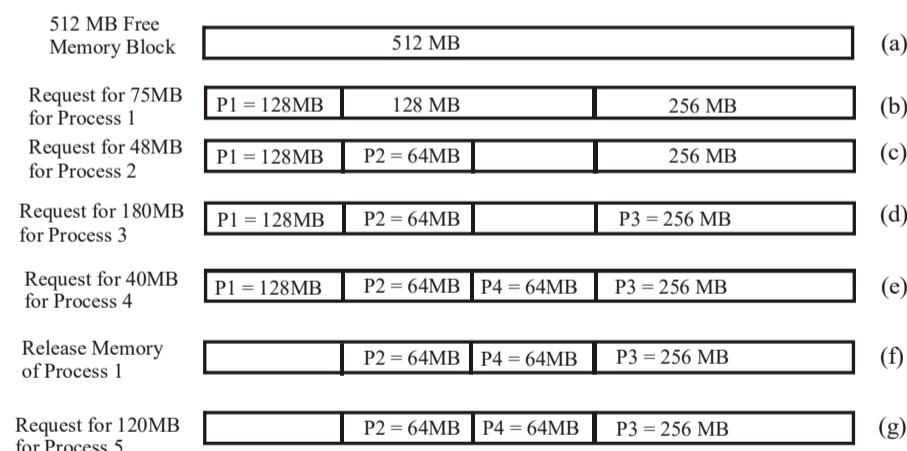


Fig. 12.7 Memory Allocation and Deallocation in Buddy System

NOTES

Check Your Progress

1. Name any three techniques of memory allocation.
2. Define internal fragmentation loss?
3. Name the techniques that produce small pieces of memory blocks.
4. What happens to the memory when a program terminates?

12.3 ADDRESS BINDING, RELOCATION AND PROTECTION

Executable images of programs are created assuming that the programs will be loaded into the memory address starting from zero. For execution, the programs will be loaded normally to starting addresses different from zero. Moreover, when programs are swapped out for finding space for some urgent execution, they have to be swapped in again into the memory, but normally to different available memory locations. So, a memory management system must provide a mechanism to adjust the absolute memory reference addresses within the code of the program which is appropriate for the new physical address space of the program. This process of adjusting the absolute memory reference addresses in the executable program is called relocation.

Protection

A process should be prevented from accessing the physical address space of the operating system or the address space of other processes without permission. That is, the operating system must protect each process address space against access by unauthorized or malicious programs and interference from malfunctioning programs.

Sharing

For sharing information between processes, they must be permitted to share memory. This is required in cooperated execution of tasks in multiple process applications. This shared access of memory must be permitted in a controlled manner.

12.4 SWAPPING

Swap space is the secondary storage space that is meant for use by the memory manager to maintain memory images of executable programs. In virtual memory systems, only a part of it is loaded into physical memory at the beginning of execution.

When the swapping technique was first developed, the term swapping was used for moving the whole memory image of a program at a time between secondary

NOTES

store (swap device) and main memory. It was useful for increasing the CPU utilization of computer systems. Most early systems were based on pure swapping where the entire process image was moved between the swap device and main memory. Pure swapping introduced high I/O overheads and increased latencies when processes had to move from swapped sleeping state to in-memory ready state. Later, when virtual memory was introduced, swapping was used for moving smaller segments or pages of processes between the swap device and main memory. This considerably brought down I/O overheads and latency. In today's scenario of memory management, the virtual memory system is based on pages, and paging and swapping are used interchangeably to mean demand paged memory management.

12.5 SEGMENTATION

Segmentation is a memory allocation technique that requires low overhead. This completely eliminates internal fragmentation and reduces the problem of external fragmentation as well. Memory is easily protected by associating some protection bits in the segment table.

A program may be considered as consisting of many segments of codes, one for each function or routine. Alternatively, we can also consider the whole code as one segment, data as another segment and stack as a third segment. Hence, instead of allocating a single contiguous block of memory, we can allocate the total memory as more than one block of appropriate size (exactly equal to the sizes of the segments) to load the program. The virtual or logical address now consists of segment number and offset within that segment as shown in Figure 12.8.

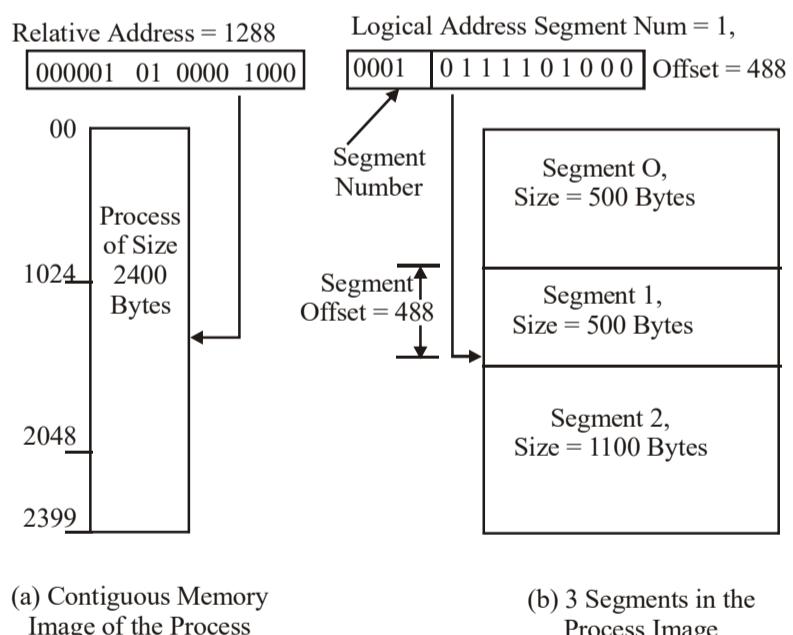


Fig. 12.8 Logical Addresses in Segmentation

NOTES

With each segment, there is a set of access permissions such as execute only, shared, read only or read/write. Each segment has a length, the number of bytes or words that it occupies in memory. References within the segment are permitted only for the associated access permissions of the segment, and for the offset addresses that are less than the segment length. Any violations in access permissions and the limits of offset addresses will be caught by raising appropriate interrupt exceptions to abort process execution.

A contiguous memory image of size 2400 bytes of a process is shown in Figure 12.8(a) and the three segments of the same process in the segmentation technique are shown in Figure 12.8(b). The sizes of segments 0, 1 and 2 are 800 bytes, 500 bytes and 1100 bytes, respectively totalling to 2400 bytes. A relative address 1288 in the linear address (relative to 0) space belongs to offset 488 in segment 1.

Segmentation requires a segment table to indicate where the segments are loaded in memory and the address translation hardware (or memory management unit) to map the virtual segment number and offset to physical memory address as shown in Figure 12.9. The segment table in Figure 12.9 has three entries, each indicating the base address of the segment loaded in physical memory and the length of the segment. In the Figure 12.9, the segments 0, 1 and 2 are shown loaded starting at addresses 13056 (binary 0010 0010 0000 0000), 8992 (binary 0011 0011 1000 0000), and 17408 (binary 0100 0100 0000 0000) respectively. The lengths of the segments are 800, 500 and 1100 respectively shown in binary. 4 bits are used for segment numbers and 12 bits are used for offsets within segments. So, there can be a maximum of 16 segments and collectively the segment sizes can be up to 4096 bytes. When references are made to the segment using offset addresses, the offsets are checked by the address translation unit to ensure that they are within the segment length. If the offset is less than the length of the segment, it is added to the base address of the segment to give the corresponding physical address; otherwise an illegal memory reference exception is raised to abort process execution. An offset of 488 (binary 0001 1110 1000) in segment 1 is translated into the physical address $8992 + 488 = 9480$.

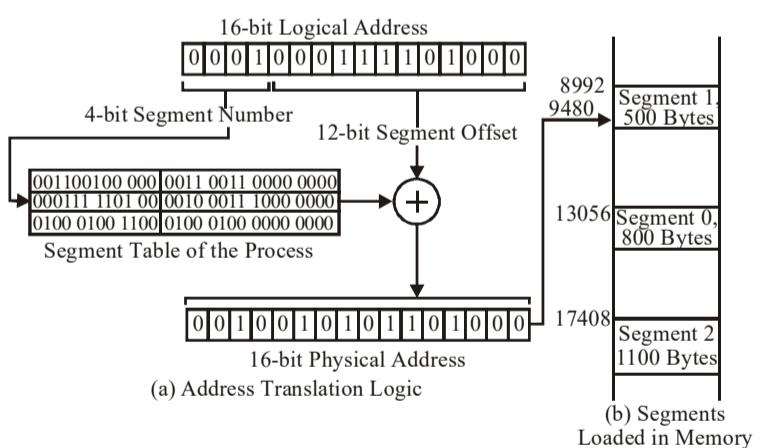


Fig. 12.9 Logical to Physical Address Translation in Segmentation

NOTES

Segmentation can be also implemented with paging, where the segments are divided into fixed pages and a segment offset is viewed as a page number and an offset within that page. Each segment uses a page table to map this virtual page and offset within the segment to the physical memory frame and offset within it.

To support virtual memory, a segment may have a present-or-absent bit to indicate whether the segment is in memory or on secondary backing store. When the segment is not loaded into the memory, any attempt to access the segment will raise a segment missing interrupt exception. This triggers execution of the OS handler to load the segment from secondary store and update the present-or-absent bit of the segment. Any attempt to access the segment will then proceed normally.

12.6 VIRTUAL MEMORY

The implementation of virtual memory is done using demand paging or demand segmentation. The computer systems, such as Burroughs use demand segmentation. Most of the operating systems use demand paging as demand segmentation involves complex segment replacement algorithms. We will look at demand paging.

Demand Paging

All the processes before starting their execution are present in the secondary storage. When the process is to be executed, it is brought into the memory. The process is divided into fixed-size partitions called pages. Rather than loading the entire process into the memory, the system brings only those pages that are needed by the user. The pager is used to bring the pages into the memory. The swapper is used to load the entire process into the memory.

The pager estimates which pages of the process may be used. This improves the optimal usage of the memory as unused pages are not brought into the memory. This decreases the page swap time as well as the memory required by the process. Demand paging needs some support from the hardware to differentiate the pages that are present in the memory and the pages that are present in the secondary storage. Valid–Invalid bit is used to decide whether the reference to the page is valid or not. This bit indicates that the page is present in the memory or not. If it is set to valid, then the page is in the memory. Any reference to the page whose Valid–Invalid bit is set to valid is legal.

Page Fault

The performance of a virtual memory management system depends on the total number of page faults with the help of following factors:

- **Page Replacement Algorithm:** This algorithm is used to select which frames are to be replaced when a page fault occurs.
- **Frame Allocation Algorithm:** This algorithm is used to decide how many frames are allocated to each process.

NOTES

- **Page Replacement:** When there is a page fault, the referenced page must be loaded. If there is no available frame in memory, one page is selected for replacement and if the selected page has been modified, it must be copied back to disk (swapped out).

A page replacement algorithm is used to satisfy the inclusion property or is called a stack algorithm if the set of pages in a k -frame memory is always a subset of the pages in a $(k + 1)$ -frame memory. Figure 12.10 shows the segmented virtual memory with demand paging which includes Page Table Base Processor (PTBR) to create page map table.

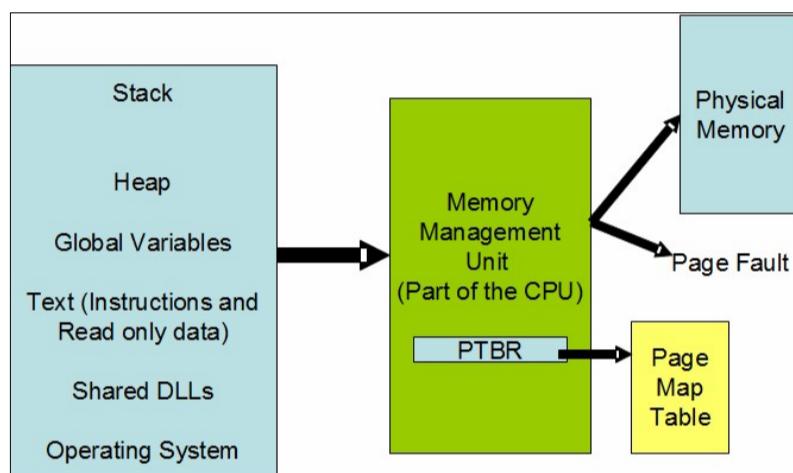


Fig. 12.10 Segmented Virtual Memory with Demand Paging

Pure Demand Paging

We can start the execution of the process with no pages in the memory. When the program counter points to the instruction, we bring the page containing the instruction. This scheme brings a page when it is required; in other words, it never brings in a page until it is required. This concept is called pure demand paging.

When the running process is analysed, an interesting fact is observed. The process tends to have locality of reference. The pages of the same process are faulted.

Performance of Demand Paging

Demand paging affects the performance of the system. The effective access time in demand paging is calculated using the following formula:

$$\text{Effective Access Time (EAT)} = (1 - p) \times \text{Memory Access} \\ + p (\text{Page Fault Overhead} \\ + \text{Swap Page Out} \\ + \text{Swap Page In} \\ + \text{Restart Overhead})$$

NOTES

Where

p is the probability of the page fault and $0 \leq p \leq 1$.

We want p to be equal to 0, i.e., the system has few page faults.

When a page fault occurs, the following happens:

- A trap is sent to the operating system.
- The state of the process and its register contents is stored.
- The operating system determines whether the trap is due to the page fault.
- The operating system checks whether the referenced page is legal and the actual location of the page on the disk is determined.
- Find a free frame and issue a page transfer request from the disk.
 - (i) The request is sent to the device queue until it gets the chance.
 - (ii) The system waits for the seek time and rotation latency.
 - (iii) Transferring of the page is done.
- While the system is finding a free frame to issue a page transfer request from the disk, the CPU can be allocated to the other processes.
- When the disk transfer of the page is complete, the IO interrupt is generated.
- The state of the other process is stored.
- The interrupt generated by the disk results in interrupt routine to be executed.
- The page table entries are updated and the bit is set to valid.
- Let the scheduler allocate the process to the CPU.
- The state of the process is restored and the execution is resumed.

All the above steps occur when a page fault occurs. A lot of time is wasted when the page fault occurs. If the time taken to service the page fault is 25 ms and the memory access time is 100 ms, then the effective access time is as follows:

$$\begin{aligned}\text{Effective Access Time} &= (1 - p) \times (100) + p (25 \text{ ms}) \\ &= (1 - p) \times (100) + p \times 25,000,000 \\ &= 100 + 24,999,900 \times p\end{aligned}$$

The effective access time is directly proportional to the page fault rate.

12.6.1 Basic Concepts

Programs must be loaded into the memory for execution. During this period, the program is called a process. In a multiprogramming environment, several processes are executed in the system. The memory for all the processes must

be allocated. If the memory size is 1000 KB and we have 5 processes each of size 200 KB, then the memory for all the processes can be allocated without any difficulty. If the processes size is greater than the total memory size, then we cannot execute those processes.

Using the virtual memory technique, we can execute a process which is larger than the physical memory. Using virtual memory, the physical memory can be assumed to be an extremely large memory. When the virtual memory technique is used, the programmer need not worry about the physical memory, which means you can execute programs that are larger than the size of the physical memory. Virtual memory decreases the performance of the system and is difficult to implement. The main advantage of virtual memory is that it uses the existing physical memory effectively.

If we closely inspect the program execution, the entire program is not needed inside the memory. If the function `Sqrt` is to be executed, then it must be present in the memory. Sometimes, the programs have the error code that is executed when the error occurs. If the error does not occur in the program execution, then the error handling code is not needed to be loaded in the memory. Arrays may be declared to be of size 500 and only then some of the elements may be used. Some of the functions in the program are never called; so they are not required to be in the memory.

In some of the cases, the entire program is needed but the CPU executes only one instruction at a time if that instruction is present in the memory.

The virtual memory separates the logical memory from the physical memory. This separation results in large virtual memory for the users. Virtual memory allows sharing of files and memory by different processes. Here the pages are shared, which improves the performance at the time of the process creation.

Check Your Progress

5. What is assumed for creating executable images of programs?
6. Name the access permissions that are attached to every segment?
7. Where is segmentation implemented in paging?
8. How is virtual memory implemented?
9. What fact is observed when the running process is analysed?
10. What does the virtual memory separate?

12.7 LOGICAL VERSUS PHYSICAL ADDRESS SPACE

Logical or virtual addresses are those used while generating executable files by the linker. Logical address space is the set of addresses used in the program when the code is assembled by the linker; that is, from zero to the size of the program.

NOTES

NOTES

Physical addresses are the actual addresses in the main memory where the program is loaded, and the physical address space is the set of addresses used by the program when loaded into the main memory.

The linker assumes that the program will be loaded starting from address zero onwards. So, all the addresses (absolute addresses) used with memory reference instructions like load and store, are relative to the starting zero address. When the program is to be loaded for execution, it may be loaded to some non-zero starting physical memory address. So, all the absolute addresses are to be adjusted to suit the new starting address of the program. This is called relocation.

The actual representation of logical/virtual addresses depends on the type of memory allocation technique employed.

If the entire program is to be loaded into the memory as a single contiguous segment, the logical addresses are from zero to the size of the program.

If a program is to be loaded into different non-contiguous memory segments (segmentation technique is explained in the subsequent sections), the logical addresses will contain a segment number and the addresses (offsets) from the beginning of the segments.

If paged memory allocation (paging is given in subsequent sections) is employed, then the logical addresses include a page number and the addresses (offsets) from the beginning of the pages.

12.8 PAGING

Paging is a storage mechanism used in Operating Systems to retrieve processes in the form of pages from the secondary storage into the main memory.

The primary principle behind the paging process is to isolate each framework in the form of pages. In the form of frames, the main memory will also be split.

One page of the operation must be stored in one of the frames of the memory. The pages can be located at different memory locations, but it is always the goal to find the neighbouring frames or gaps. Only if necessary otherwise they remain in the secondary storage are pages brought into the main memory of the operation.

In today's scenario of memory management, people use swapping and paging interchangeably. Historically, swapping is totally different from paging. These are two different activities of memory management techniques.

Paging is a technique of memory management where small fixed-length pages (memory page frames) are allocated instead of a single large variable-length contiguous block in the case of the dynamic allocation technique. Paging was introduced primarily to eliminate the severe external fragmentation problem of single large variable-length contiguous block memory allocation schemes. Later when virtual memory was introduced, the demand paging technique was developed

NOTES

to execute programs whose sizes were larger than available physical memory with the help of a swap device for swapping pages between secondary store and main memory. The development of paging was found to be very useful for swapping out (writing out to secondary store) a part of a process instead of the whole memory image of a process to find memory following a request for loading new pages of a process. This also helped to reduce I/O overheads and process latency due to swapping as the pages that are swapped (using the Least Recently Used (LRU) page replacement algorithm) will be the ones that are not required immediately for execution. Today, paging means demand paged memory management without the need for virtual memory or the use of any swap devices.

12.8.1 Page Size and Page Realease

We have shown that an additional overhead is induced by the larger page table size because we have to split the table into the pages and then store it in the main memory.

Our concern must be about processes not running on the page table execution. The page table offers support for method execution. The wider the table for the page, the greater the overhead.

Page Table Size = number of page entries in page table X The size of a single page entry in page table X

Let's take an example.

Space for Virtual Address = 2 GB = 2×2^{30} Bytes

The Size of the Page = 2 KB = 2×2^{10} Bytes

Number of Pages in the Table of Pages = $(2 \times 2^{30}) / (2 \times 2^{10}) = 1 M$ pages

1 million pages will be available, which is quite a large amount. Try making the page size bigger, say 2 MB, though.

Then, in the page table, the number of pages = $(2 \times 2^{30}) / (2 \times 2^{20}) = 1 K$ pages.

We will figure out that the page table size is anti-proportional to the page size by comparing the two scenarios.

There's still excess on the final page at Paging. If the virtual address space is not a page size multiple, so there will be some bytes left and we have to allocate that many bytes to a complete page. This is essentially an overhead project.

Let's consider

Size of page = 2 KB

Room of Virtual Address = 17 KB

Number of pages then = $17 \text{ KB} / 2 \text{ KB} =$

The number of pages will be 9, but only 1 byte will be used on the 9th page and the rest of the page will be wasted.

NOTES

In general,

If size of page = p bytes

Size of entry = e bytes

Room for Virtual Address = S bytes

Overhead $O = (S/p) \times e + (p/2)$ then,

On average, the amount of pages lost in a virtual space is $p/2$ (the half of total number of pages).

The minimum overhead, for

$$O/p = 0$$

$$-S/(p^2) + 1D 2 = 0$$

$$P == \text{bytes } (2.S.e) \text{ bytes}$$

Therefore, the overhead would be negligible if the page size is equal to $(2.S.e)$ bytes.

12.9 PAGE TABLE AND ITS ENTRIES

As the page table entries are quite large in number, typically millions, the page table itself needs to be organized so that it becomes easy to search a page in the page table. The logical address space is typically 2^{32} to 2^{64} and if the page size is 4 KB 2^{12} , then the page table consists of one million entries ($2^{32}/2^{12}$). To solve the problem, the page table is divided into many smaller parts using several approaches. These approaches are as follows:

Two-Level Paging

The page table itself is paged. The logical address is divided into three parts—one for indexing the outer page table, the second to index the inner page table and the third for the displacement. Figure 12.11 shows two-level paging.

Outer Page	Inner Page	Offset
p_1	p_2	d
42	10	12

Fig. 12.11 Two-Level Paging

The Virtual Address eXtension (VAX) architecture also supports two-level paging.

In Figure 12.12, the outer page table has pointers to the parts of the page table. If the user wants to search the page, first the outer page table is searched for the portion of the page table. Then, the inner page table is searched for the exact page.

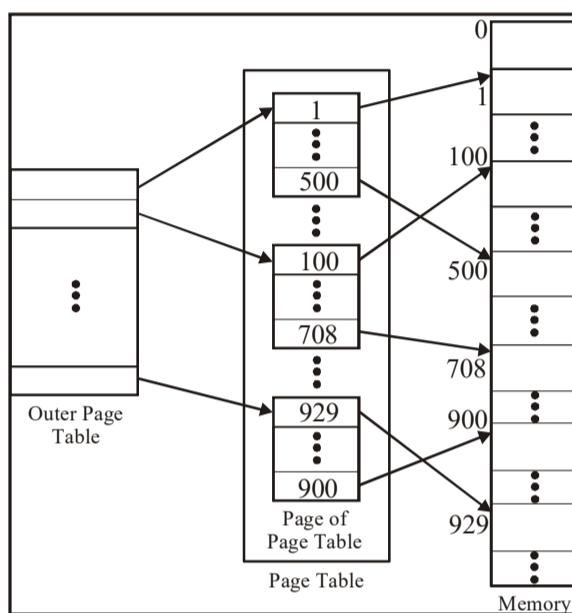
NOTES

Fig. 12.12 Searching Page Table

For the very large page table, the outer page can further be divided into parts as follows:

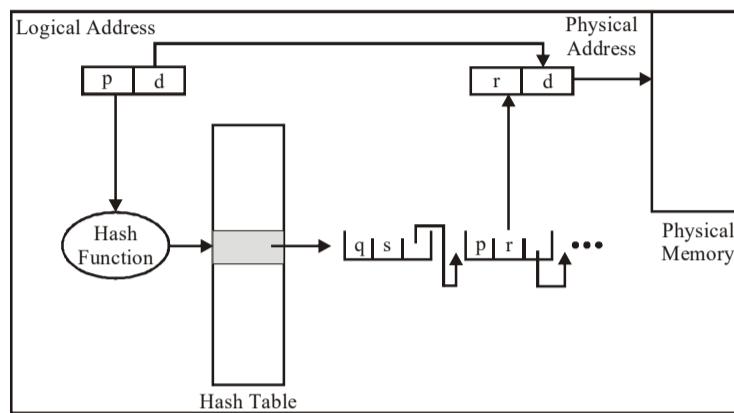
2nd Outer Page	Outer Page	Inner Page	Offset
p_1	p_2	p_3	d
32	10	10	12

Hashed Page Table

To handle the address space of a size larger than 32 bits, hashed page tables are used. Virtual page number is used as hash value. Initially, the virtual page number is hashed and the values are placed in the hash table. The hash table contains the hash value of the virtual page number and the corresponding frame number. The virtual page number generated by the CPU is hashed to get the hashed page number. The hashed page number is mapped to the frame number in the hashed page table. The hashed page table has the search time constant; this means that to search any entry, the hash table needs constant time. If two virtual page numbers produce the same hash value, then there is a collision. A linked list is used to store the values. If there is a match, the corresponding page frame is resolved.

NOTES

Figure 12.13 depicts the hashed page table scheme.

**Fig. 12.13** Hashed Page Table Scheme

The logical address contains the page number. This is also called virtual page. The virtual page number is hashed using the hash function to produce a value. If the two or more virtual page numbers produce the same hash value, then they are stored in the linked list. In Figure 12.13, the linked list contains pages q with frame number s and page p with frame number r. Both the pages q and p produce the same hash value. When collision occurs, then the linked list is searched for the desired page and the corresponding frame number is retrieved. The frame number along with displacement is the physical address.

Inverted Page Table

Every process has its own page table. A page table has one entry for each frame. The operating system has to first find out where the page table is present in the memory and then use the page table to search the actual physical address of the page. Each page table consists of millions of entries and may consume a large amount of physical memory. What processes are present in the memory and where each process page table is stored in the memory is difficult to find. The drawback of this approach is managing large page tables. To solve the problem, an inverted page table is used. The inverted page table consists of entries where each entry has the process id and the page number. This scheme has only one page table. This scheme needs the process id and the page number in that process to find the frame number. The virtual address consists of <process-id, page-number, offset>. The inverted page table entry is the pair process-id and the page number. When the memory is referenced, the virtual address consisting of process-id and the page number is given to the memory management unit. The memory management subsystem searches for the mapping. If a match is found at location, say k, then the physical address <k, offset> is produced. If there is no mapping found, then it is treated as illegal address. The disadvantage of this

NOTES

approach is that it increases the search time as the inverted page table is sorted by the physical address; sometimes, we need to search the entire page table. This would take more time. To solve this, we can use the hashing technique to decrease the search time.

Shared Pages

In paging, a common code can be shared. This approach minimizes memory requirements. This is useful in time-sharing systems. Consider a system with five users and every one using a text editor application. If a text editor application has 100 KB of code and 100 KB of data section, then we need a total of 1000 KB for five users. The code section is common for all the users; hence, it can be shared among the five users. Here we need 600KB in the sharing mode. Sharing is possible if the code is re-entrant. Re-entrant code is not changed during execution so more than one process can share the re-entrant code. In sharing the pages, each process must have its own storage area for storing the register contents because the two processes have different data but the same code section. In our text editor application with five users, the code section of the text editor can be shared but each user has his own data section. Only one copy of the editor is required in the memory. The programs that are heavily used by the users are shared, which optimizes the memory use. The systems that use the inverted page tables have difficulty in implementing shared pages.

12.10 PAGE FAULT

A page fault occurs when the process wants to access the page whose Valid–Invalid bit is set to invalid. This indicates the page is not in the memory. If the process wants it in the memory, then it issues a page fault. The system must bring the page requested into the memory. When a page fault occurs, the system must handle the page fault (Refer Figure 12.14).

The following are the steps for handling a page fault:

- First the Valid–Invalid bit is checked to know whether the reference is invalid or valid.
- If the bit is set valid and the page is not present in the memory, then find a free frame.
- Issue the disk transfer operation to bring the desired page into the memory in the newly allocated frame.
- When the desired page is brought into the memory, then the internal page table is updated with the page number and the frame is allocated.
- The instruction execution is resumed.

NOTES

Figure 12.14 shows the handling mechanism when a page fault occurs.

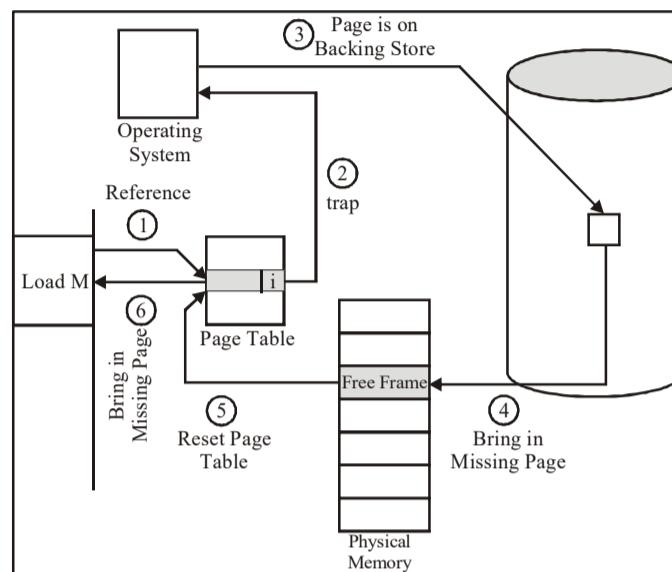


Fig. 12.14 Handling Mechanism during Page Fault

We need to store the registers contents, program counter and the other state of the process before the page fault. After the page is brought into the memory, the process can resume without any difficulty.

12.10.1 Demand Paging

The simple paging technique described earlier requires that all pages of a process are loaded into physical memory before the start of execution. However, it is possible to start the execution if the instructions and data pertaining to the current execution is available in the main memory. That is, at any point in the execution sequence, if the pages that are referenced in the execution are available in the memory, the execution can proceed. In other words, when a process tries to reference a virtual address that is not in the physical memory, it should be brought into the physical memory before execution. So, if an arrangement is made to bring a page into physical memory when it is attempted for reference for the first time, the execution can start even before loading any page into memory. This is the case of pure demand paging. In such systems, the operating system responds to the failed attempts of the page references and brings the pages from secondary store. When the process accesses a page not in physical memory, a page fault interrupt exception transfers control to the operating system page fault interrupt handler. This OS handler brings the page from secondary store to the physical memory and updates the page table entry of the page with the new frame number where the page is loaded. So, in pure demand paging, a process may begin with no pages loaded in the physical memory, and page faults bring pages into memory until all the pages in the working set are loaded.

However, in most systems, a minimum number of pages are loaded before the start of execution in order to reduce the number of page faults during execution.

Virtual Memory Management

Advantages

Demand paging has the following advantages when compared to pure paging where all pages are loaded before the start of execution:

- Pages that are not accessed will not be loaded into main memory. This saves memory space which can be used for loading other programs to increase the degree of multiprogramming.
- Initial program start-up is faster as only a few essential pages are loaded to start execution.
- Initial disk overhead is reduced as fewer pages are loaded to start execution.
- No additional hardware support is required other than what is needed for simple paging. The protection page fault interrupt handler is employed to bring pages not loaded, from secondary store when they are referenced for the first time.
- The program size can be larger than the available physical memory.

NOTES

Demand Paging System Advantages Over Partitioned Non-Virtual System:

The program size can be larger than the available physical memory. There is no additional work for the programmer to run the program using virtual memory unlike the case of the old manual overlay, where the programmer has to design the overlay.

Disadvantages

- Program execution will be slower due to the latency in loading pages at run time. A technique to reduce the latency and hence to increase the speed of execution is to use prepaging. In prepaging, the pages needed at all points in the previous execution sequence are remembered and these pages are pre-loaded by the paging system before accessing them. This reduces the normal page faults.
- It is difficult to provide this sophisticated memory management technique to embedded computers which are normally low-cost and low-power systems.
- The demand paging technique incorporating page replacement algorithms is somewhat complex.

12.10.2 Multilevel Page Table

One way to reduce space in the page table is to avoid allocating space for parts of the address space that are not in memory. Multilevel page tables are a good approach to do this. The page identifier is broken down into sub-indices and each of which corresponds to a sub-page table. The more levels of division, the more precise the allocation but there is a limit because of overhead. The more levels of

NOTES

division and the more memory references are needed to resolve an address. Multilevel page tables effectively address the space concerns of page tables. The biggest drawback of single level page tables is their size. They must contain one entry for each virtual address. This is an enormous number of entries especially considering that the virtual address space by design should be far larger than is ever required. Consider the large hole between the stack and the heap in which there is an entry for every unused page.

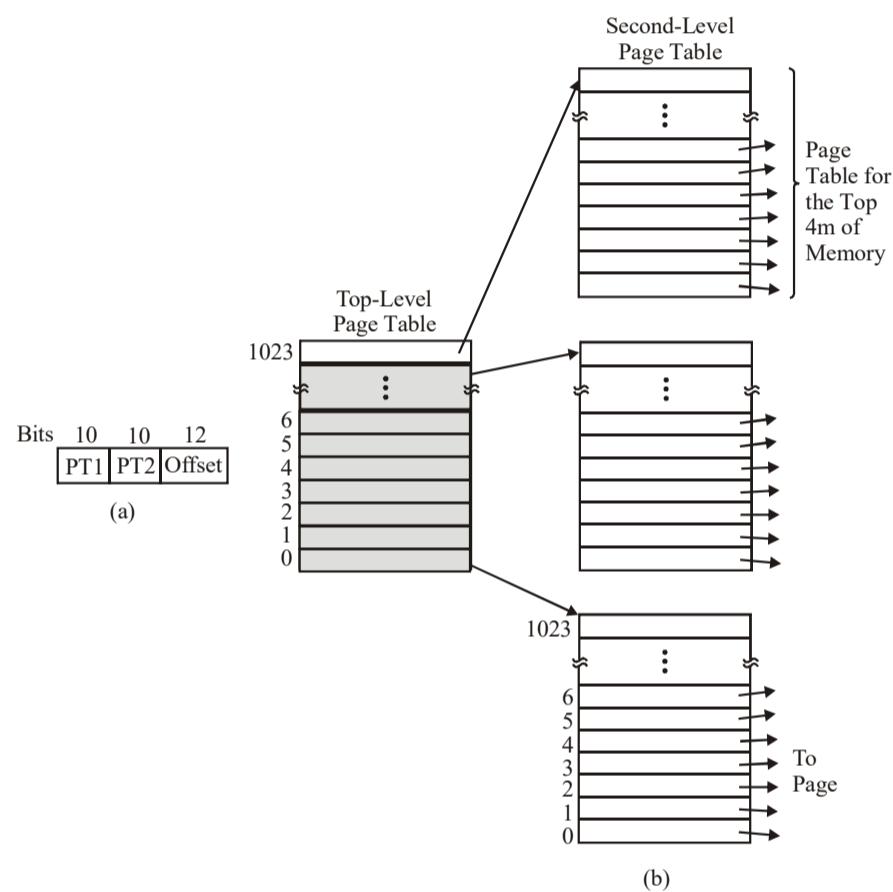


Fig. 12.15 (a) A 32-Bit Address with Two Page Table Fields (b) Two-Level Page Tables

The function of the multilevel page table method is to avoid keeping all the page tables in memory all the time. In particular, those that are not needed should not be kept around. For example, a process needs 12 MB, the bottom 4MB of memory for program text, the next 4 megabytes for data, and the top 4MB for the stack. In between the top of the data and the bottom of the stack is a gigantic hole that is not used. In Figure 12.15(b), we have the top-level page table with 1024 entries corresponding to the 10-bit PT1 field. When a virtual address is presented to the MMU, it first extracts the PT1 field and uses this value as an index into the top-level page table. Each of these 1024 entries represents 4M because the entire 4 GB, i.e., 32-bit virtual address space has been segmented into memory chunks of

NOTES

1024 bytes (Refer Figure 12.15(a)). The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table. Entry 0 of the top-level page table points to the page table for the program text, entry 1 points to the page table for the data and entry 1023 points to the page table for the stack. The other (shaded) entries are not used in top-level page table. The PT2 field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

As an example, consider the 32-bit virtual address 0x00403004 (4,206,596 decimal) which is 12,292 bytes into the data. This virtual address corresponds to PT1 = 1, PT2 = 2 and Offset = 4. The address space contains over a million pages but only four page tables are needed. The second-level tables use memory stack ranges from 0 to 4M, 4M to 8M and above 8M. The literal 'M' represents 'Modify' operation to memory which is generally restricted for 1024 entries. Page offset d is a displacement combined with base address to define the physical memory address that is sent to the memory unit. With 4KB pages, the last 12 bits are the offset into the page. That leaves 20 bits for a 32-bit address. These 20 bits are used as an index into the page tables not an offset. By splitting them evenly between the two levels, both tables are of the same size. The first table specifies the address of the second table and the second table specifies the physical address of the page. 10 bits for each index means 1024 entries in each table. This approach is used frequently because each table exactly contains 1 page while using 32-bit entries. The Memory Management Unit (MMU) first uses PT1 to index into the top-level page table and obtain entry 1 which corresponds to addresses 4M to 8M. It then uses PT2 to index into the second-level page table just found and extract entry 3, which corresponds to addresses 12288 to 16383 within its 4M chunk, i.e., absolute addresses 4,206,592 to 4,210,687. This entry contains the page frame number of the page containing virtual address 0x00403004. If that page is not in memory, the Present/Absent bit in the page table entry will be zero causing a page fault. If the page is in memory, the page frame number taken from the second-level page table is combined with the offset (4) to construct a physical address. This address is put on the bus and sent to memory. The Present/Absent bits in 1021 entries of the top-level page table are set to 0, forcing a page fault if they are ever accessed. The two level page table system can be expanded to three, four or even more levels. Additional levels give more flexibility but it is doubtful that the additional complexity is worth it beyond three levels.

Structure of a Page Table Entry

Let us now see the structure of the page tables in the large to the details of a single page table entry. The exact layout of an entry is machine dependent but the kind of information present is roughly the same from machine to machine. The size varies from computer to computer, but 32 bits is a common size. The most important

NOTES

field is the Page Frame Number. After all, the goal of the page mapping is to locate this value. Next to it we have the Present/Absent bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault. Figure 12.16 displays the structure of page table entry.

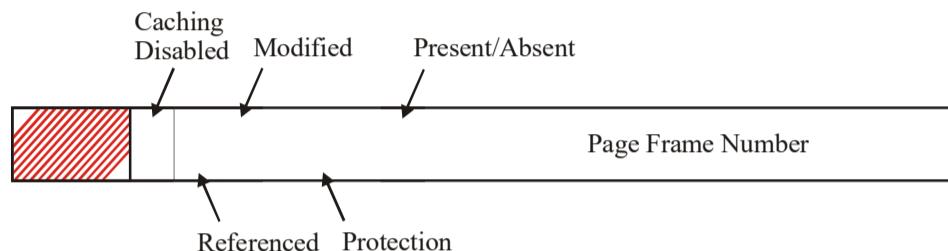


Fig. 12.16 Structure of Page Table Entry

The protection bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement is having 3 bits in which 1-bit is used for enabling reading, writing and executing the page. The Modified and Referenced bits keep track of page usage. When a page is written to, the hardware automatically sets the Modified bit. This bit is of value when the operating system decides to reclaim a page frame. If the page in it has been modified, i.e., dirty and it must be written back to the disk. If it has not been modified, i.e., clean and it can just be abandoned since the disk copy is still valid. The bit is sometimes called the dirty bit since it reflects the page's state. The Referenced bit is set whenever a page is referenced, either for reading or writing. Its value is to help the operating system used choose a page to when a page fault occurs. Pages that are not being used are better candidates than pages that are and this bit plays an important role in several of the page replacement algorithms.

Finally, the last bit allows caching to be disabled for the page. This feature is important for pages that map onto device registers rather than memory. If the operating system is sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is essential that the hardware keep fetching the word from the device and not use an old cached copy. With this bit, caching can be turned off. Machines that have a separate I/O space and do not use memory mapped I/O do not need this bit. The disk address used to hold the page when it is not in memory is not part of the page table. The reason is simple. The page table holds only that information which the hardware needs to translate a virtual address to a physical address. Information of the operating system needs to handle page faults is kept in software tables inside the operating system. The hardware does not need it. Following steps are required to know the concept of multilevel page table.

- Prepare one big page table.

NOTES

- Call it the second-level page table and cut it into pieces into each size of a page. You can get many page table entries in one page so you will have few pages than page table entries.
- Construct a first-level page table containing page table entries that point to these pages.
- This first-level page table is small enough to store in memory.
- Since, the second-level page table already exist then this level page tables refer to unused memory.

For a two level page table, the virtual address is divided into three pieces which can be represented as follows:

+-----+	+-----+	+-----+
P#1	P#2	Offset
+-----+	+-----+	+-----+

In the above example, P#1 gives the index into the first level page table. Then the next pointer in the corresponding PTE is used to reach the frame containing the relevant second-level page table. P#2 gives the index into this second-level page table. Then, the next pointer in the corresponding page table entry is used to reach the frame containing the (originally) requested frame. Offset gives the offset in this frame where the requested word is located. VAX uses a second-level page table structure. There is no need to stop the operations at this level. Scalable Processor ARChitecture (SPARC) has three levels and the Motorola 68030 has four levels but the number of bits of virtual address used for P#1, P#2, P#3 and P#4 can be varied. Virtual memory allows each process to have its own private address space. The page table for the process maps memory references from this virtual private address space onto page frames in real memory. Because a process can subsequently access any memory location in its address space, the page table has to be big enough to cover every page within that space whether that page is actually mapped to real memory or not. A 32-bit system with 4KB pages means that 20-bit pages are used in page table for 20-bit indexing. It means that the page table needs to have 1,048,576 (2^{20}) entries. If extra flags are skipped then each entry is about 24 bits through which the page table consumes 3MB per process with a 64-bit address space and each page table will consume approximately 28,672 TeraByte (TB).

The vast bulk of processes use only a very small part of their address space although the regions that are used can be segmented for other processing. For example, the stack may be at a different end of memory than the text and heap. It is useless to maintain entire page tables since large regions of it are unused and empty. To optimize memory use for a sparse page table, you can arrange the page table in a hierarchy. The virtual address is divided into three parts. The high order bits index into a top level page table which refers to index table. This table contains the base address of a partial page table, i.e., the pages that contain those high order bits. The middle bits are an offset into this partial page table. This partial page table contains page table entries. If a large enough region of memory is not used, then the partial page tables considered as last bits used for those regions

NOTES

which do not need to be created and the corresponding entries in the index table can indicate that there is no valid mapping.

Figure 12.17 illustrates the frame address used in page tables. The virtual address is divided into three fields known as the directory # (page table #1), the page number (page table #2) and the offset. Directory entries are created to span the virtual address space. Fewer entries can cover the entire address space. Page tables are created (page table #2) for those directory entries that are actually used. Processes with sparse memory usage save space. But, it also costs time. An additional access is used in memory to do a page table lookup. It is also important to note that multilevel page tables consume more memory than single level page tables if most of the address space is actually used and the difference is lost in the overhead of the additional structure.

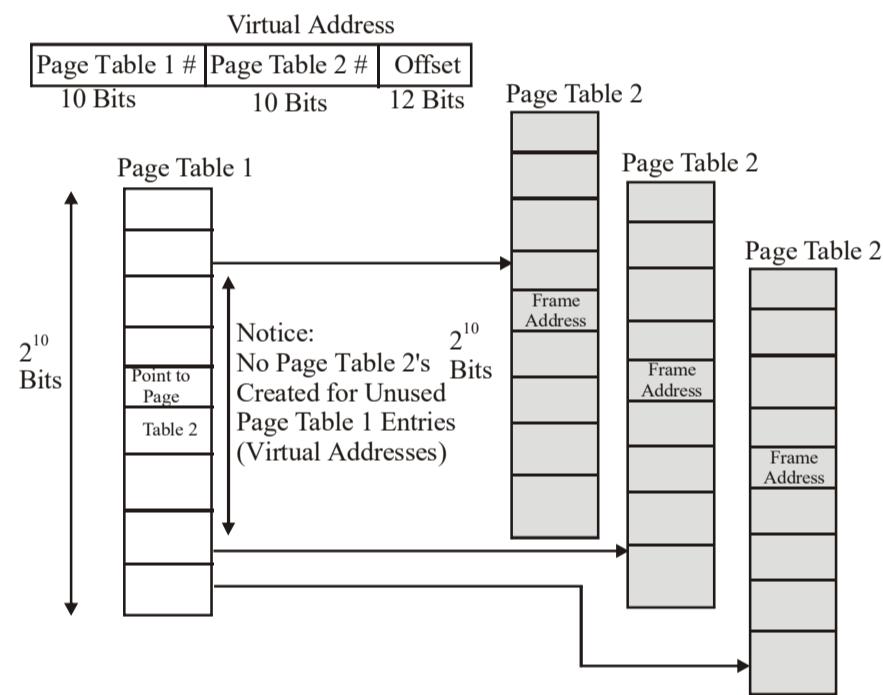


Fig. 12.17 Frame Address used in Page Tables

Most of the virtual memory is the unused space between the data and stack regions. However, with demand paging this space does not waste real memory. But, the single large page table does waste real memory. The concept of multilevel page tables refers to a similar idea used in UNIX inode based file systems which is to add a level of indirection and have a page table containing pointers to page tables. Let us see how the virtual memory is managed. When a user executes a program, the operating system creates an address space for it to run in. This address space will include the instructions for the program itself as well as any data it requires. In a system with memory protection, each process is restricted by the operating system to accessing only the memory in its own address space. However, the combined program memory requirements

NOTES

often exceed the system's amount of physical memory installed on a computer. So, modern operating systems use a portion of the hard disk called a swap file to extend the amount of available memory. This technique, called virtual memory, treats physical memory as a cache of the most recently used data. In a virtual memory system, memory is divided into units called pages. A page is typically 4-8 KB in size. The set of addresses that identify locations in virtual memory is called the virtual address space. Each process is allocated a virtual address space. The virtual address space can range from 0-4 GB on a 32-bit architecture. Figure 12.18 shows the virtual memory addressing in the memory space.

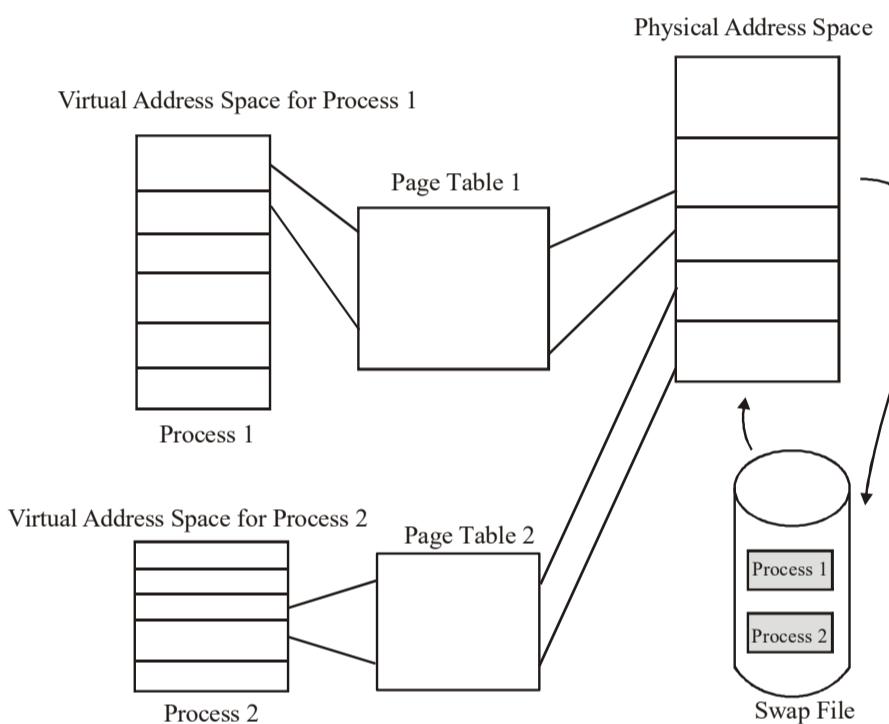


Fig. 12.18 Virtual Memory Addressing

In Figure 12.18, virtual address space for Process 1 refers to the physical address space whereas virtual address space for Process 2 refers to the swap file that includes the two processes, namely Process 1 and Process 2. A process's address space contains the set of instructions and data that is mapped into virtual memory when a program is executed. Virtual memory addresses are translated into physical memory addresses through the use of a look-up table called a page table. In addition to mapping, the entire process is set into virtual memory in which a subset of pages is also mapped into physical memory. As each instruction in the process executes, it is either found in physical memory or is not found called a page fault. When a page fault occurs, the page that is needed must be moved from the hard disk into physical memory before the instruction can be executed. To make room (stack) for the new page, the operating system may need to decide which page is

NOTES

used to move out from physical memory. This process is known as swapping. A page fault is time consuming because retrieving data from the hard disk is much slower than obtaining it directly from the physical memory. Operating systems attempt to minimize the number of page faults by swapping multiple pages at once. This becomes a trade-off between operating system overhead and the time saved by minimizing page faults which is affected by the size of each process's working set. The operating system takes care of swapping and translating from virtual address space to physical address space. This means that developers have a flat address space at their disposal. With a virtual memory scheme, the amount of memory available is seemingly limited only by the amount of hard drive space. Virtual memory has more memory available than physical memory because it removes the restriction that an entire process must be loaded into memory at one time.

12.11 PAGE REPLACEMENT

As stated earlier, when page fault occurs, page fault routine locates for a free page frame in memory and allocates it to the process. However, there is a possibility that the memory is full, that is, no free frame is available for allocation. In that case, the operating system has to evict a page from the memory to make space for the desired page to be swapped in. The page to be evicted will be written on the disk depending on whether it has been modified or not. If the page has been modified while in the memory, it is rewritten to the disk; otherwise no rewrite is needed.

To keep track whether the page has been modified, a **modified (M)** bit (also known as **dirty** bit) is added to each page table entry. This bit indicates whether the page has been modified. When a page is first loaded into the memory, this bit is cleared. It is set by the hardware when any word or byte in the page is written into. At the time of page replacement, if dirty bit for a selected page is cleared, it implies that the page has not been modified since it was loaded into the memory. The page frame is written back to the swap space only if dirty bit is set.

The system can select a page frame at random and replace it by the new page. However, if the replaced page is frequently accessed, then another page fault would occur when the replaced page is accessed again resulting in degradation of system performance. Thus, there must be some policy to select a page to be evicted. For this, there are various page replacement algorithms. These replacement algorithms can be evaluated by determining the number of page faults using a reference string. A **reference string** is an ordered list of memory references made by a process. It can be generated by a random-number generator or recording the actual memory references made by an executing program. To illustrate the page replacement algorithms, consider the reference string as shown in Figure 7.5

for the memory with three page frames. For simplicity, instead of actual memory references, we have considered only the page numbers.

5	0	5	3	5	2	5	0	1	0	7	3
---	---	---	---	---	---	---	---	---	---	---	---

Fig 7.5 Structure of the Reference String

First-In First-Out Page Replacement

The first-in, first-out (FIFO) is the simplest page replacement algorithm. As the name suggests, the first page loaded into the memory is the first page to be replaced. That is, the page is replaced in the order in which it is loaded into the memory.

To illustrate the FIFO replacement algorithm, consider our example reference string shown in Fig 7.5. Assuming that initially, all the three frames are empty, the first two references made to page 5 and 0 and cause page faults. As a result, they are swapped in memory. The third reference made to page 5 does not cause page fault as it is already in memory. The next reference made to page 3 causes a page fault and that page is brought in memory. The reference to page 2 causes a page fault which results in the replacement of page 5 as it is the oldest page. Now, the oldest page is 0, so reference made to page 5 will replace page 0. This process continues until all the pages of reference string are accessed. It is clear from Figure 7.6 that there are nine page faults.

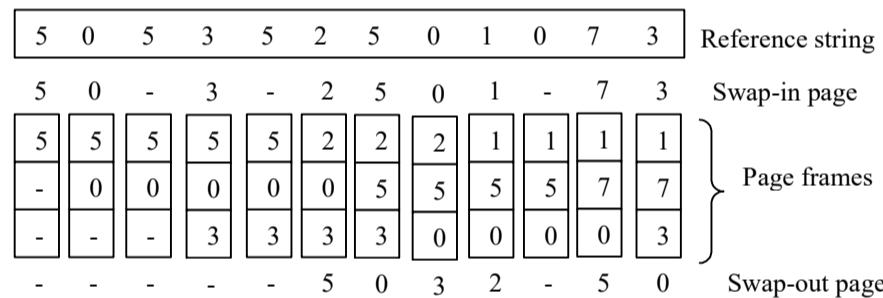


Fig 7.6 FIFO Replacement Algorithm

To implement this algorithm, each page table entry includes the time (called **swap-in** time) when the page was swapped in the memory. When a page is to be replaced, the page with the earliest swap-in time is replaced. Alternatively, a FIFO queue can be created to keep track of all the pages in the memory with the earliest one at the front and the recent at the rear of the queue. At the time of page fault, the page at the front of the queue is removed and the newly arrived page is added to the rear of the queue.

The FIFO page replacement algorithm is easier to implement as compared to all other replacement algorithms. However, it is rarely used as it is not very efficient. Since it does not consider the pattern of the usage of a page, a frequently

NOTES

NOTES

used page frame may be replaced resulting in more page faults. Moreover, it suffers from **Belady's anomaly**—a situation in which increasing the number of page frames would result in more page faults. To illustrate this, consider the reference string containing five pages, numbered from 2 to 6 (see Fig 7.7). From this figure, it is clear that with three page frames, a total of nine page faults occur. On the other hand, with four page frames, a total of ten page faults occur.

2	3	4	5	2	3	6	2	3	4	5	6
Reference string											
2	3	4	5	2	3	6	-	-	4	5	-
2	2	2	5	5	5	6	6	6	6	6	6
-	3	3	3	2	2	2	2	2	4	4	4
-	-	4	4	4	3	3	3	3	3	5	5
-	-	-	2	3	4	5	-	-	2	3	-
Swap-in page											
Page frames											
Swap-out page											

(a) FIFO with three page frames

2	3	4	5	2	3	6	2	3	4	5	6
Reference string											
2	3	4	5	-	-	6	2	3	4	5	6
2	2	2	2	2	2	6	6	6	6	5	5
-	3	3	3	3	3	3	2	2	2	2	6
-	-	4	4	4	4	4	4	3	3	3	3
-	-	-	5	5	5	5	5	4	4	4	4
-	-	-	-	-	-	-	2	3	4	5	6
Swap-in page											
Page frames											
Swap-out page											

(b) FIFO with four page frames

Fig. 7.7 Belady's Anomaly

Optimal Page Replacement

The optimal page replacement (OPT) algorithm is the best possible page replacement algorithm. The basic idea behind this algorithm is that whenever a page fault occurs, some pages are in memory; out of these pages, one will be referenced at the next instruction while other pages may not be referenced until the execution of certain number of instructions. In case of page replacement, the page that is referenced in last will be replaced. That is, the page to be referenced in the most distant future is replaced. For this, each page can be labelled in the memory with the number of instructions to be executed before that page is referenced for the first time. The page with the highest label is replaced from the memory.

NOTES

To illustrate this algorithm, consider our example reference string (see Figure 7.5). Like FIFO, the first two references made to pages 5 and 0 cause page faults. As a result, they are swapped into the memory. The third reference made to page 5 does not cause page fault as it is already in memory. The reference made to page 3 causes a page fault and thus is swapped into memory. However, the reference made to page 2 replaces page 3 because page 3 is required at the last instruction whereas pages 5 and 0 are required at the next instruction. The page faults and the pages swapped-in and swapped-out for all the page references are shown in Figure 7.8. This algorithm causes seven page faults.

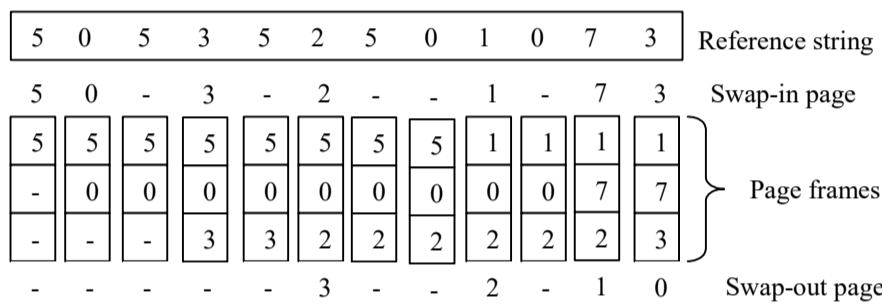


Fig 7.8 Optimal Page Replacement Algorithm

The advantage of this algorithm is that it causes the lowest number of page faults as compared to other algorithms. The disadvantage of this algorithm is that its implementation requires prior knowledge of which page will be referenced next. Though this algorithm is not used in systems practically, it is used as the basis for comparing performance of other algorithms.

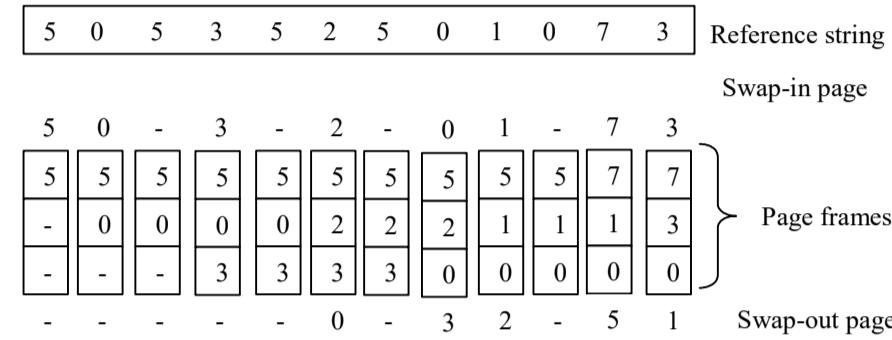
Note: To implement OPT, a program can be executed on a simulator and all the page references are recorded. Using the page reference records obtained during first run, it can be implemented at second run.

Least Recently Used Page Replacement

The least recently used (LRU) algorithm is an approximation to the optimal algorithm. Unlike optimal algorithm, it uses the recent past behaviour of the program to predict the near future. It is based on the assumption that the page that has been used in the last few instructions will probably be referenced in the next few instructions. Thus, it replaces the page that has not been referenced for the longest time.

Consider our example reference string (see Figure 7.5). As a result of this algorithm, the page faults and the pages swapped-in and swapped-out for all the page references are shown in Figure 7.9. Upto five references, page faults are same as that of optimal algorithm. When a reference is made to page 2, page 0 is replaced as it was least recently used. However, after page 5, it is being used again leading to a page fault. Regardless of this, the number of page faults is eight which is less than in case of FIFO.

NOTES

**Fig 7.9 LRU Page Replacement Algorithm**

One way to implement LRU is maintaining a linked list of all the pages in the memory; the most recently used page is at the head and the least recently used page is at the tail of the list. Whenever a page is to be replaced, it is deleted from the tail of the linked list and the new page is inserted at the head of the linked list. The problem with this implementation is that it requires updating the list at every page reference despite of whether page fault occurs or not. It is because whenever a page in memory is referenced, being the most recent page, it is removed from its current position and inserted at the head of the linked list. This results in extra overhead.

Alternatively, the hardware can be equipped with a counter. This counter is incremented by one after each instruction. The page table has a field to store the value of the counter. Whenever a page is referenced, the current value of the counter is copied to that field in the page table entry for that page. Whenever a page is to be replaced, this algorithm searches the page table for the entry having the lowest counter value (means the least recently used page) and replaces that page.

Clearly, it has less page faults as compared to FIFO algorithm. Moreover, it does not suffer from the Belady's anomaly. Thus, it is better than FIFO algorithm and is used in many systems. The disadvantage of this algorithm is that it is time consuming when implemented using linked list. Otherwise, it needs extra hardware support for its implementation.

Note: Both the optimal and LRU algorithms belong to a special class of page replacement algorithms called **stack algorithms** that never exhibit Belady's anomaly.

The Second Chance Page Replacement

The second chance page replacement algorithm (sometimes also referred to as **clock** algorithm) is a refinement over FIFO algorithm. It replaces the page that is both the oldest as well as unused instead of the oldest page that may be heavily used. To keep track of the usage of the page, it uses the **reference bit (R)** which

is associated with each page. This bit indicates whether the reference has been made to the page while it is in memory. It is set whenever a page is accessed for either reading or writing. If this bit is clear for a page that means this page is not being used.

Whenever, a page is to be replaced, this algorithm uses the FIFO algorithm to find the oldest page and inspects its reference bit. If this bit is clear, the page is both the oldest and unused and thus, replaced. Otherwise, the second chance is given to this page and the reference bit of this page is cleared and its load time is set to the current time. Then the algorithm moves to the next oldest page using FIFO algorithm. This process continues until a page is found whose reference bit is clear. If the reference bit of all the pages is set (that is, all the pages are referenced), then this algorithm will proceed as pure FIFO.

This algorithm is implemented using a circular linked list and a pointer that points to the next victim page. Whenever a page is to be replaced, the list is traversed until a page whose reference bit is clear is found. While traversing, the reference bit of each examined page is cleared. When a page whose reference bit is clear is found, that page is replaced with the new page and pointer is advanced to the next page. For example, consider our example reference string with reference bits shown in Figure 7.10. The algorithm starts with the page 5, say at time $t = 18$. Since, the reference bit of this page is set, its reference bit is cleared and time is reset to the current system time as though it has just arrived in the memory. The pointer is advanced to the next page that is page 0. The reference bit of this page is clear, so it is replaced by the new page. The pointer is advanced to the page 3 which will be the starting point for next invocation of this algorithm.

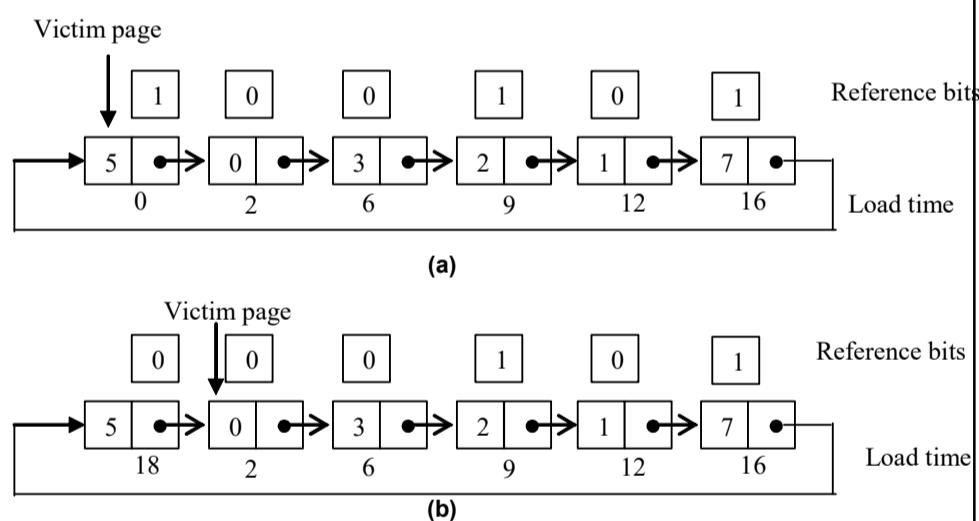


Fig 7.10 The Second Chance (Clock) Page Replacement Algorithm

NOTES

NOTES

Counting-Based Page Replacement Algorithm

Other than the page replacement algorithms discussed earlier, there are several other algorithms. Some of them keep record of how often each page has been referenced by associating a counter with each page. Initially, the value of this counter is 0, which is incremented every time when a reference to that page is made. That is, the counter counts the number of references that have been made to each page. A page with the highest value of the counter is heavily used while for a page with the lowest value of counter, there are following interpretations:

- That page is least frequently used
- That page has been just brought in and is yet to be used

The algorithm based on the first interpretation is known as **least frequently used (LFU)** page-replacement algorithm. In this algorithm, when a page is to be replaced, the page with lowest value of counter is chosen for replacement. Clearly, the page that is heavily used is not replaced. The problem with this algorithm arises when there is a page that was used heavily initially, but afterwards never used again. For example, in a multipass compiler, some pages are used heavily during pass 1; after that pass, they may not be required. Still, these pages will not be replaced as they have high value of counter. Thus, this algorithm may replace useful pages instead of pages that are not in use. The algorithm that is based on the second interpretation is called the **most frequently used (MFU)** page-replacement algorithm.

Both these algorithms are not commonly used, as their implementation is expensive. Moreover, they do not approximate the OPT page replacement algorithm.

Check Your Progress

11. What does the actual representation of virtual address depend on?
12. Define paging.
13. What are the different parts of logical address?
14. What does inverted page table consist of?
15. When does the page fault interrupt exception transfer control to the operating system page fault interrupt handler?
16. What is the drawback of the single level page tables?
17. How can the memory of the sparse page table be optimized?

12.12 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Various techniques of memory allocation are available for multiprogramming are,
 - (i) Equal size fixed partitions.
 - (ii) Variable size fixed partitions.
 - (iii) Dynamic partitioning.
2. If the job size is smaller, a second job cannot be loaded into a partition simultaneously with the first one, resulting in unutilized memory space. This is called internal fragmentation loss, because it is internal to the allocated partition.
3. Both the Best Fit and First Fit techniques produce small pieces of memory blocks that are smaller than the normal size of processes.
4. When programs terminate, memory is deallocated and this memory is reallocated for new programs.
5. Executable images of programs are created assuming that the programs will be loaded into the memory address starting from zero.
6. With each segment, there is a set of access permissions such as execute only, shared, read only or read/write.
7. Segmentation can be implemented with paging where the segments are divided into fixed pages and a segment offset is viewed as a page number and an offset within that page.
8. The implementation of virtual memory is done using demand paging or demand segmentation.
9. When the running process is analysed, an interesting fact is observed. The process tends to have locality of reference. The pages of the same process are faulted.
10. The virtual memory separates the logical memory from the physical memory.
11. The actual representation of virtual address depends on the type of memory allocation technique employed.
12. Paging is a technique of memory management where small fixed-length pages (memory page frames) are allocated instead of a single large variable-length contiguous block in the case of the dynamic allocation technique.
13. The logical address is divided into three parts—one for indexing the outer page table, the second to index the inner page table and the third for the displacement.

NOTES

NOTES

14. The inverted page table consists of entries where each entry has the process id and the page number.
15. When the process accesses a page not in physical memory, a page fault interrupt exception transfers control to the operating system page fault interrupt handler.
16. The biggest drawback of single level page tables is their size.
17. To optimize memory use for a sparse page table, you can arrange the page table in a hierarchy.

12.13 SUMMARY

- Memory management is the act of managing computer memory. In its simpler forms, this involves providing ways to allocate portions of memory to programs at their request and freeing it for reuse when no longer needed. The management of main memory is critical to the computer system.
- The simplest memory allocation is to have several fixed memory partitions and allocate a process to each one.
- Different sized partitions can be allocated to accommodate different processes memory needs. Unless the workload is very well understood, fixed partitions are seldom used.
- Address binding is a mapping from one address space to another. Protection means that a process should be prevented from accessing the physical address space of the operating system or the address space of other processes without permission.
- Swapping and segmentation are memory allocation techniques. Segmentation requires low overhead which completely eliminates internal fragmentation and reduces the problem of external fragmentation as well.
- Virtual memory is a feature of an operating system that enables a process to use a memory (RAM) address space that is independent of other processes running in the same system, and use a space that is larger than the actual amount of RAM present, temporarily relegating some contents from RAM to a disk, with little or no overhead.
- The set of all logical addresses generated by a program is known as logical address space, whereas the set of all physical addresses corresponding to these logical addresses is physical address space.
- Paging serves a twofold purpose - memory protection and virtual memory (the two being almost inextricably interlinked). A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses.

When a process does something the memory management unit does not like, a page fault interrupt is thrown.

- The simple paging technique described earlier requires that all pages of a process are loaded into physical memory before the start of execution. However, it is possible to start the execution if the instructions and data pertaining to the current execution is available in the main memory.
- One way to reduce space in the page table is to avoid allocating space for parts of the address space that are not in memory. Multilevel page tables are a good approach to do this.

Virtual Memory Management

NOTES

12.14 KEY WORDS

- **Memory management:** It is the act of managing computer memory involving ways for allocating portions of memory to programs at their request and freeing it for reuse when no longer needed.
- **External fragmentation:** It is the situation wherein the total amount of free memory may be greater than that required for loading a program.
- **Relocation:** It is the process of adjusting the absolute memory reference addresses in the executable program.
- **Virtual address space:** It is the set of addresses that identify locations in virtual memory.
- **Swap space:** This is the secondary storage space that is meant for use by the memory manager to maintain memory images of executable programs.

12.15 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answers Questions

1. What are the methods of memory allocation for multiprogramming?
2. What is the feature of the Worst Fit technique?
3. What is the process of keeping track on allocated and free memory blocks?
4. Why is protection required by a process?
5. Define swap.
6. How is memory protected in segmentation?
7. How does a segment support virtual memory?
8. What is a page fault?

NOTES

9. What is the use of virtual memory?
10. Define logical address space.
11. How is paging correlated to swapping?
12. What does the virtual address extension support?
13. What is the use of the linked list?
14. What are the advantages of demand paging system over partitioned non-virtual systems?
15. Where is a referenced bit set in a page table entry?
16. Define virtual memory addressing.

Long-Answers Questions

1. Explain the importance of memory allocation in an operating system.
2. Describe the various methods of memory allocation.
3. Briefly describe external and internal fragmentation.
4. Describe the concept of Buddy system with the help of examples.
5. What is address binding? Explain its importance in multiprogramming.
6. How is swapping different from segmentation? Elaborate.
7. Explain the use of logical addresses in segmentation?
8. Virtual memory involves both paging and segmentation. Explain.
9. Differentiate between logical and physical address spaces.
10. What is the role of paging in an OS? Explain with the help of examples.
11. Explain the various differences in hashed page table and inverted page table.
12. Describe the various steps used for handling page faults.
13. Explain the advantages and disadvantages of demand paging.
14. Discuss the structure of page table entry with the help of illustrations.
15. Explain the use of frame address in page tables.

12.16 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.

Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.

Virtual Memory Management

Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.

Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.

Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.

Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.

Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

NOTES

BLOCK - V
DISK PERFORMANCE AND FILE,
DATABASE SYSTEMS

UNIT 13 DISK SCHEDULING

Structure

- 13.0 Introduction
- 13.1 Objectives
- 13.2 Concept of Disk Management
- 13.3 Disk Structure
- 13.4 Disk Scheduling
- 13.5 Disk Scheduling Algorithms
 - 13.5.1 FCFS Scheduling
 - 13.5.2 SSTF Scheduling
 - 13.5.3 SCAN Scheduling
 - 13.5.4 C-SCAN Scheduling
 - 13.5.5 LOOK Scheduling
- 13.6 Rotaional Optimization
- 13.7 Answers to Check Your Progress Questions
- 13.8 Summary
- 13.9 Key Words
- 13.10 Self Assessment Questions and Exercises
- 13.11 Further Readings

13.0 INTRODUCTION

Physical memory is not sufficient to accommodate all the needs of a computer. It is limited in size and is not permanent. Secondary storage is a collection of disks and magnetic tapes that can store large amounts of data and data stored on disks are permanent. Hard disks and tape drives are examples of secondary storage media. They are more efficient and cheaper. When a process needs to perform I/O to or from a disk, it invokes a system call to the operating system. The operating system checks to see whether the desired disk drive is available; if it is available, the request can be serviced immediately; otherwise, the request is placed in a queue. Once an I/O operation is completed, the operating system selects the pending request to be serviced.

Disks come in various shapes and sizes. The most common types of disks are floppy disks, hard disks and diskettes. Floppy disks and diskettes consist of a single disk of magnetic material, while a hard disk is normally a stacked set of disks placed on top of one another. Each disk has a separate read/write head. In

hard disks, there is a limit on the density of information, which can be stored on it. A stepper motor is used to control the heads, which moves across each surface, which has a fixed number of tracks on it. All the tracks on all the surfaces of the disks are aligned. A cylinder is the sum of all the tracks that are at a fixed distance from one end of the disk. Hard disks require protection from dust. As hard disks have become more efficient and cheaper, floppy disks are no longer used.

A disk is the permanent storage media. It is divided into a number of cylinders; each cylinder is divided into a number of tracks; each track is divided into a number of sectors and each sector has several blocks. Addressing of disk drive is done as large one-dimensional arrays of logical blocks. A logical block is the smallest unit of transfer. The size of a logical block is generally 512 bytes. The one-dimensional array of logical blocks is mapped into the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from the outermost to the innermost.

Disk scheduling is performed by operating systems to coordinate the arrival of I/O requests for the disc. Disk scheduling is referred to as I/O scheduling as well. Disk scheduling is important because: Multiple I/O requests can come through various processes and the disc controller can serve only one I/O request at a time.

In this unit, you will study about the disk, disk scheduling strategies and rotational optimization.

13.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand how an operating system manages disks
- Analyse the disk structure
- Explain disk management, structure of a disk and disk scheduling
- Compare various disk scheduling algorithms
- Know about the Rotational optimization

13.2 CONCEPT OF DISK MANAGEMENT

Disks are used to store bulk data as secondary storage for modern computers. A hard disk has a device controller, which translates numerical addresses into head movements. The device controller gets the instructions for execution from the operating system.

A Small Computer System Interface (SCSI) drive is the most popular drive used for large personal computers and workstations. An SCSI is of four types—

Disk Scheduling

NOTES

NOTES

SCSI 1, SCSI 2, fast SCSI 2 and SCSI 3. An SCSI drive is coupled with a bus and identifies an SCSI address. Each SCSI controller can address up to seven units.

Each track is usually divided into sectors; a sector is a fixed-size region, which is present along the track. When writing to a disk, data is written in units of a whole number of sectors. On a disk, the number of sectors per track is not constant, nor is the number of tracks. If a sector is damaged, it must be patched up by some repair program. Usually, data in that sector is lost. Disks must be checked for damages. When a disk is formatted, the damaged sectors are not used for storing data. The SCSI drive keeps a defect list that consists of all bad sectors. A new disk contains a list of sectors and is updated as more defects occur. Formatting is a process by which the sectors of a disk are created along the tracks; each sector is labelled with an address, so that the disk controller can find the correct sector.

On simple disks, formatting is done manually. On complex disks like SCSI drives, a low-level formatting on the disk is done by the manufacturer. High-level formatting is not necessary when a file system is able to manage the hardware sectors.

Data consistency is checked by writing data to the disk and reading back the data. An error occurs if there is any disagreement. Some device controllers detect bad sectors and move data to a good sector if there is an error. Another way of checking data consistency is to calculate a number for each sector; the number is calculated on the basis of what data is present in the sector. The calculated number is stored in the sector. When data is read back, the number is recalculated and if there is disagreement, then an error is generated. This is called Cyclic Redundancy Check (CRC) or error correcting code.

The disk drive rotates at the speed 60 to 200 rotations per second. **Transfer rate** is the rate at which data flows between a drive and a computer. **Positioning time (random-access time)** is the time for moving the disk arm to the desired cylinder and is also called **seek time**. The time for the desired sector to rotate under the disk head is called **rotational latency**. **Head crash** results from the disk head making contact with the disk surface. **Host controller** in computer uses bus to talk to **disk controller** built into the drive or storage array.

13.3 DISK STRUCTURE

A disk is the permanent storage media. It is divided into a number of cylinders; each cylinder is divided into a number of tracks; each track is divided into a number of sectors and each sector has several blocks, as shown in Figure 13.1. Addressing of disk drive is done as large one-dimensional arrays of logical blocks. A logical block is the smallest unit of transfer. The size of a logical block is generally

512 bytes. The one-dimensional array of logical blocks is mapped into the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from the outermost to the innermost.

To access the disk faster, each track is usually divided into sectors; a sector is a fixed-size region, which is present along the track. When writing to a disk, data is written in units of a whole number of sectors. Because the heads of the disk move simultaneously on all the surfaces of the disks, we can increase read/write efficiency by allocating blocks related to one file in parallel across all surfaces.

The physical properties of a disk are number of tracks, sectors per track, speed of revolution and the number of heads. An operating system must access all the different types of disks without any difficulty. On a disk, the number of sectors per track is not constant, nor is the number of tracks.

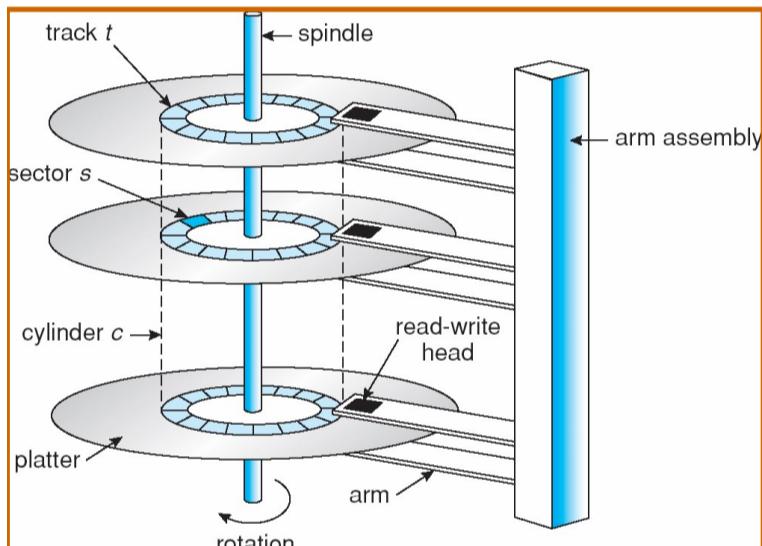


Fig. 13.1 Physical Properties of a Disk

Disk Scheduling

NOTES

13.4 DISK SCHEDULING

The operating system is responsible for the efficient use of the disk drives; this means having fast access time and a high bandwidth for the disks. The disk header moves only in the forward or backward direction. The access time has the following three major components:

- Seek time is the time required for the disk to move the heads to the cylinder containing the desired sector.
- Rotational latency is the time required for the disk to rotate the desired sector.

NOTES

- Transfer time is the time taken to transfer the data present in the blocks in a sector.

The transfer time is so fast that it can be negligible. The rotation latency is normally 7200 rpm, i.e., 120 rotations per sec. Even this value is negligible. The efficiency of the operating system is directly dependent on the seek time. The objective of the operating system is to minimize seek time.

Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Disk access time for any request must be as fast as possible. Scheduling is done to improve the average disk access time. The speed of access depends on seek time. Each disk drive has a queue of pending requests. A request can be the input request (write request) or output (read request). A disk address consists of disk number, cylinder number, surface number, sector number and the number of blocks to transfer.

Data in the blocks is transferred into the main memory address. Block numbers indicate the amount of information to be transferred, i.e., the byte count.

Check Your Progress

- Define sector, track and cylinder.
- What is the disk bandwidth?
- How is information on a disk referenced?
- Describe seek time.
- Define latency time.
- What characteristics determine the disk access speed?

13.5 DISK SCHEDULING ALGORITHMS

Servicing of disk I/O requests can be done by using disk scheduling algorithms.

We illustrate disk scheduling algorithms with an example.

Consider a disk with 200 tracks from 0 to 199. Let the request queue be

98, 183, 37, 122, 14, 124, 65, 67

and let the head point to 53 track.

In the above example, the disk has 200 tracks. Request queue has 98 as its first request. Here the disk must be moved from 53 track to 98 track as the disk header is positioned at 53 track initially.

13.5.1 FCFS Scheduling

Disk Scheduling

First Come First Serve (FCFS) scheduling is the simplest form of disk scheduling. It is a fair algorithm and may not provide the best possible service.

For the ordered disk queue with requests on tracks

98; 183; 37; 122; 14; 124; 65; 67

Read/write head initially at track 53

From 53 track the header moves to 98 track

From 98 track the header moves to 183 track

From 183 track the header moves to 37 track

From 37 track the header moves to 122 track

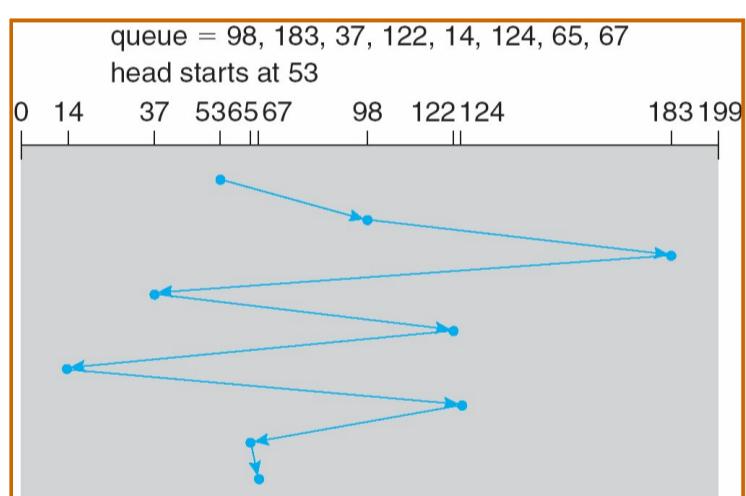
From 122 track the header moves to 14 track

From 14 track the header moves to 124 track

From 124 track the header moves to 65 track

From 65 track the header moves to 67 track

NOTES



Total head movement = 640 tracks

And there is a wild swing from 122 to 14 and back to 124. Wild swings occur because the requests do not always come from the same process; they are inter-leaved with requests from other processes.

13.5.2 SSTF Scheduling

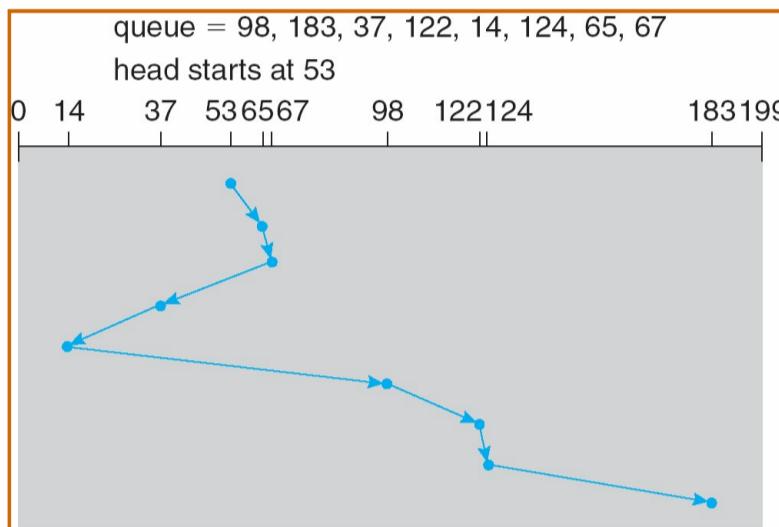
Shortest Seek Time First (SSTF) scheduling selects the request with the minimum seek time from the current head position. SSTF scheduling is a form of SJF scheduling and it may cause starvation of some requests. This algorithm services

all requests close to the current head position before moving the head far away i.e., moves the head to the closest track in the service queue.

Example: A service queue can be serviced as

NOTES

53; 65; 67; 37; 14; 98; 122; 124; 183



53; 65; 67; 37; 14; 98; 122; 124; 183

From 53 track the header moves to 65 track

From 65 track the header moves to 67 track

From 67 track the header moves to 37 track

From 37 track the header moves to 14 track

From 14 track the header moves to 98 track

From 98 track the header moves to 122 track

From 122 track the header moves to 124 track

From 124 track the header moves to 183 track

Total head movement = 263 tracks.

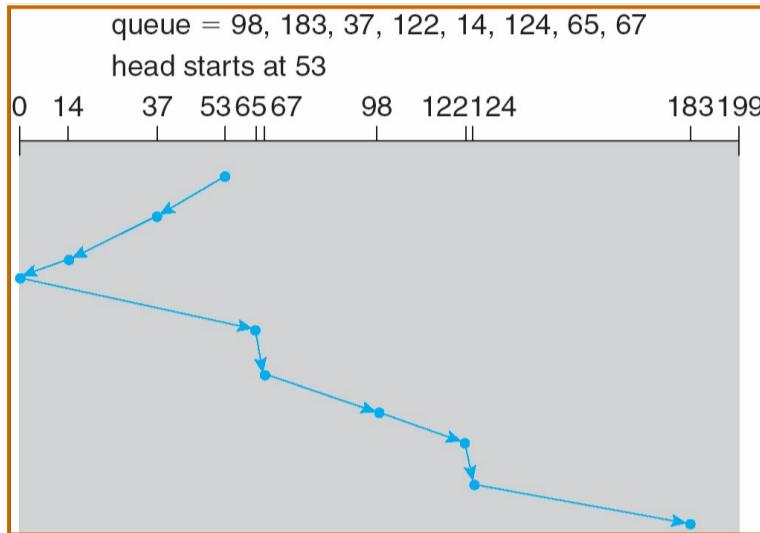
13.5.3 SCAN Scheduling

The disk arm starts at the one end of the disk and moves toward the other end, servicing requests until it gets to the other end of the disk where the head movement is reversed and servicing continues.

This is also termed as elevator algorithm due to its similarity with building elevators. The head constantly scans the disk from one end to the other. For example, the read/write head begins at one end of the disk and proceeds towards the other end, servicing all requests as it reaches each track. At the other end, the

direction of head movement is reversed and servicing continues. For example, assume a head moving towards 0 in a queue 53; 37; 14; 0; 65; 67; 98; 122; 124; 183.

Disk Scheduling



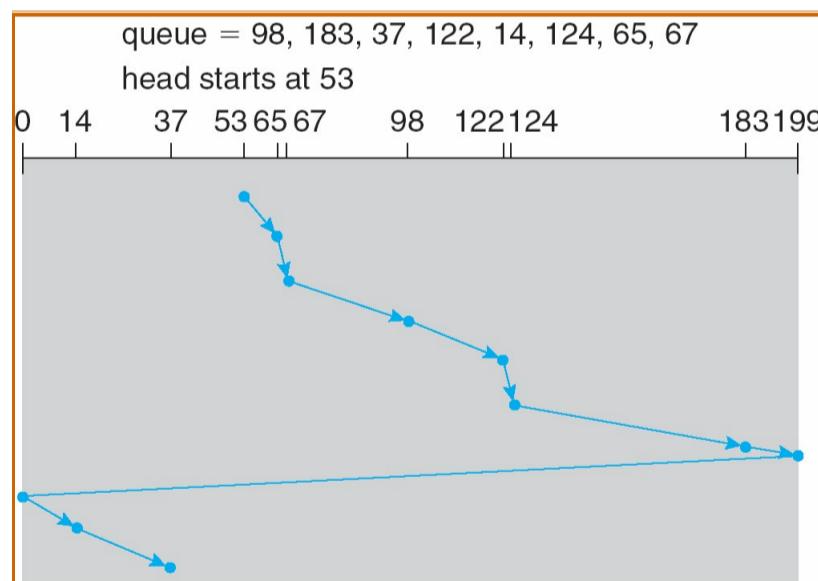
NOTES

Total head movement = 236 track.

When head reverses, we have only a few requests. Heaviest density of requests is at the other end.

13.5.4 C-SCAN Scheduling

Circular SCAN refers to a variation of SCAN scheduling. It moves the head from one end to the other. When it reaches the other end, it immediately come back to the first end without servicing any requests on the way.



NOTES**13.5.5 LOOK Scheduling**

LOOK scheduling moves the head only till the last request in that direction. No more requests in the current direction; reverses the head movement. Before you move in that direction, you need to look for a request.

Placing directories halfway between the inner and outer tracks of a disk reduce the head movement.

SSTF is common and has a natural appeal.

SCAN and C-SCAN perform better for systems that place a heavy load on the disk.

Performance depends on the numbers and types of requests.

Requests for disk service can be influenced by the file allocation method.

A disk scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm, if necessary.

Either SSTF or LOOK is a reasonable choice for the default algorithm.

When a user wants to open a file, an operating system needs to locate the file in the directory structure. The operating system searches the directory structure frequently. If the directory structure is present in the first track and the file data is present on the last track, then the head must be moved from the first track to the last track. This decreases the performance. To improve the performance, the directory structure is cached for faster access. The disk scheduling algorithms are complex to implement; hence, they are written in separate module of the operating system.

Disk Scheduling Algorithms

As discussed, accessing data from a disk requires seek, rotational delay and data transfer. Among these three, seek time is the one that dominates the entire access time. We may recall that it is the time taken for the read/write heads to position themselves on a specific track on the disk platter. Whenever a disk access request arrives, the head is moved into place on the specific track. In case only one request comes in at one point of time, all the requests are serviced as and when they arrive; nothing can be done to reduce seek time. However, there is always the possibility that new requests will arrive while the system is servicing another one. In this case, new requests are placed in the queue of pending requests. Thus, after completing the current request, the operating system has to choose which request from the queue to service next. Several algorithms have been developed that serve this purpose for operating system; some of them are discussed here. We will see that selecting requests in an appropriate order can reduce the seek time significantly.

First-Come, First-Served (FCFS) Algorithm

Disk Scheduling

In this algorithm, the request at the front of queue is always selected to be serviced next. That is, the requests are served on First-Come-First-Served basis. To understand the concept, consider a disk with 100 cylinders and a queue of pending requests to access blocks at cylinders

15, 96, 35, 27, 73, 42, 39, 55

Further, suppose that the head is resting at cylinder 40 when the requests arrive. First, the head is moved to cylinder 15, since it is at the front of the queue. After servicing the request at cylinder 15, the head is moved to 96, then on 35, 27, 73, and so on (see Figure 13.2). It is clear that servicing all the requests results in total head movement of 271 cylinders.

NOTES

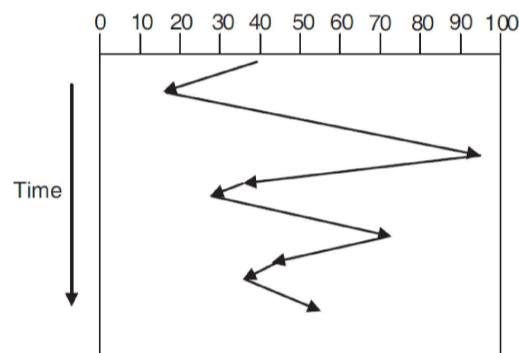


Fig. 13.2 FCFS Algorithm

Though it uses the simplest algorithm to select a request from the queue, it does not optimize the disk performance. It is clear that head movement from cylinder 15 to cylinder 96 and then back again to cylinder 35 constitutes the major portion of the total head movement of 271 cylinders. If the request for cylinder 96 could be scheduled after the requests for cylinders 35 and 27 then, the total head movement could be reduced significantly, from 271 to 179 cylinders.

Shortest Seek Time First (SSTF) Algorithm

As just discussed, scheduling request for cylinders that are far away from the current head position after the request for closest cylinder can reduce the total head movement significantly. This is what SSTF algorithm attempts to do. This algorithm suggests operating system to select the request for cylinder which is closest to the current head position. To understand this, consider once again the above pending-request queue with head initially at cylinder 40. The request for cylinder 39 is closest to the current head position, so it will move to cylinder 39. After servicing request at cylinder 39, the head will move to cylinder 42 and then to service requests at cylinders 35, 27 and 15. Now, the request for cylinder 55 is

closest, so head will move to cylinder 55. Next closest request is for cylinder 73, so the head will move to cylinder 73 and, finally, to cylinder 96 (see Figure 13.3).

NOTES

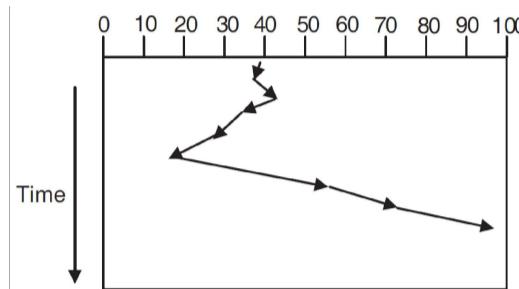


Fig. 13.3 SSTF Algorithm

This algorithm requires a total head movement of 112 cylinders, a major improvement over FCFS. However, this algorithm can cause some requests to wait indefinitely—a problem called starvation. Suppose, when the request for cylinder 15 is being serviced, a new request arrives for cylinder 17. Clearly, this algorithm makes the head to move to cylinder 17. Further suppose that while the request at cylinder 17 is being serviced, new requests arrive for cylinders 14 and 22. In fact, if requests close to the current head position arrive continuously, requests for cylinders that are far away will have to keep waiting indefinitely.

SCAN Algorithm

In this algorithm, the head starts at one end of the disk and moves toward the other end, with servicing requests at cylinders along the way. Upon reaching the other end, the head reverses its movement and continues servicing requests along the way. This process of moving the head back and forth across the disk continues.

To understand this concept, consider our example again. Here, in addition to request queue and current head position, we must know the direction in which the head is moving. Suppose the head is moving toward the cylinder 100; it will service the requests at cylinders 42, 55, 73, and 96 in the same order. Then, upon reaching the end, that is, at cylinder 100, the head reverses direction and services the requests at cylinders 39, 35, 27, and 15 (see Figure 13.4).

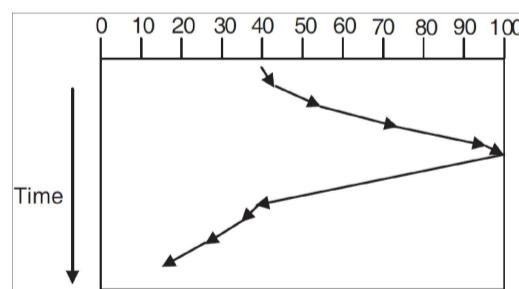


Fig. 13.4 SCAN Algorithm

NOTES

The algorithm is simple and almost obviates the starvation problem. However, all the requests that are behind the head will have to wait (no matter how close they are to the head) until the head reaches the end of disk, reverses direction, and comes back towards them. However, a new request that is right in front of the head will be serviced almost immediately (no matter when it enters the queue).

LOOK Algorithm

A minor modification of SCAN algorithm is LOOK algorithm. In this algorithm, the head starts at one end and scans toward the other end, servicing the requests along the way, just as it does in the SCAN algorithm. However, here, the head does not necessarily reach the end of disk; instead, when there are no more requests in the direction in which the head is moving, it reverses direction. Figure 13.5 illustrates LOOK algorithm for our example queue of pending requests. In this example, the head reverses its direction after servicing the request for cylinder 96.

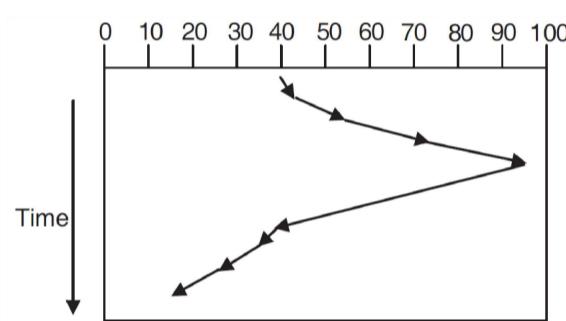


Fig. 13.5 LOOK Algorithm

C-SCAN and C-LOOK Algorithms

The C-SCAN (circular SCAN) and C-LOOK (circular LOOK) are variants of SCAN and LOOK algorithms respectively, and are designed to provide a more uniform wait time. In these algorithms, though the head scans through the disk in both directions, it services requests in one direction only. That is, when the head reaches the other hand, it promptly returns to the starting end without servicing any requests. Figure 13.6 illustrates the C-SCAN and C-LOOK algorithm for our example queue of requests.

NOTES

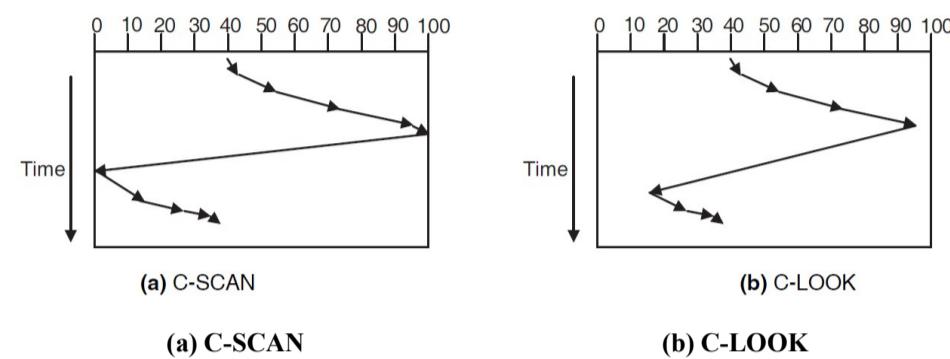
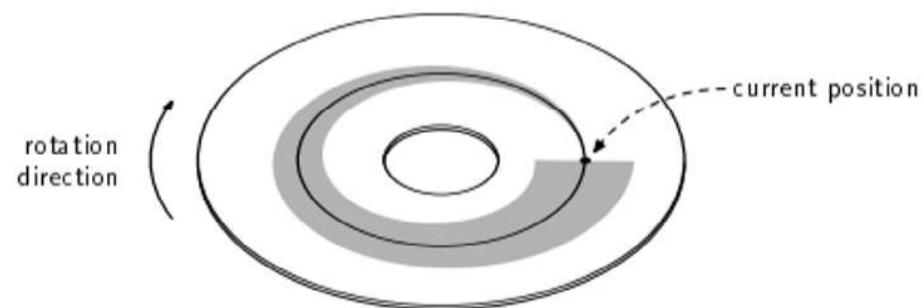


Fig. 13.6 C-SCAN and C-LOOK Algorithms

13.6 ROTAIONAL OPTIMIZATION

Rotational location optimization disc scheduling algorithms use seek distance versus rotational distance information implemented as rpo tables (arrays) which are stored in flash memory within each disc drive. We consider a novel representation scheme for this information reducing the necessary flash memory by a factor of more than thirty thereby reducing the manufacturing cost per drive.

It is a greedy algorithm based on access time that relies on the necessary search and rotational latencies. Under the head to carry the desired physical barrier. The access time for each command during the selection process, it decides the queue and selects the command with the lowest time.



In practise, specifically identifying the boundary of the accessible area is difficult and a rigorous strategy for deciding the relative position of the border is necessary for the measurement of access time. An extra rotation, referred to as a miss, would be the expense of making the wrong decision, and such performance can be much worse than choosing the next command at random.

NOTES

form factor	3.5 inch
capacity	18.3 GB
platters	5
heads	10
rotation rate	10,000 rpm
servo identifiers per platter	90
average seek	4.9 ms
number of cylinders	11,712
number of zones	15
media data rate	23.3 … 44.3 MB/s

The IBM Ultrastar 18LZX drives use probabilistic data about the edges of the reachable area since the reachable area boundary is fuzzy around the edge. Instead of calculating the access time inside the queue for each command, the estimated access time (EAT) for each command is determined. The estimated access time is the time required, on average, to perform a specific search. The estimate, together with the internal tabular representation of the required data, of the typical planned access time. Time is represented in sides; as noted in the table above, each side represents $6/90 \text{ ms} = 44.44 \text{ s}$. For a given search distance, Figure 13.7 shows the non-miss probability curve where the x-axis represents the rotational latency in sides from the current position and the y-axis represents the likelihood of finding the target cylinder without a miss within the rotational latency; these probabilities are referred to as non-miss probabilities.

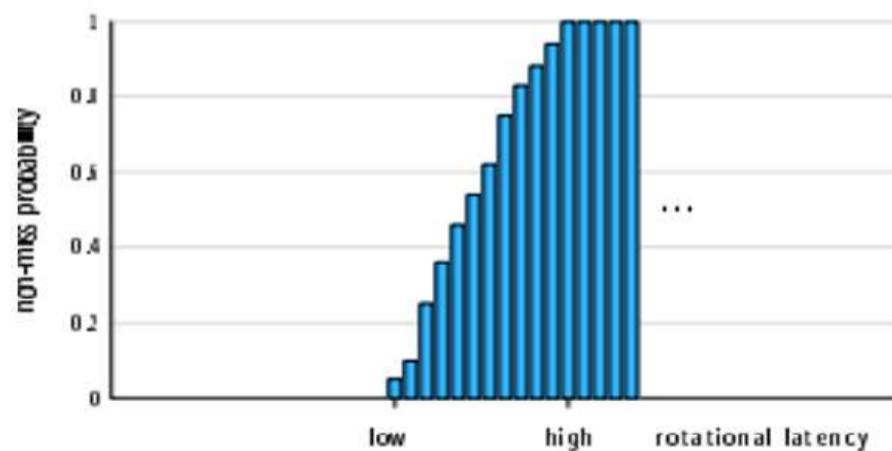


Fig. 13.7 Typical Non-miss Probability Curve for a Particular Seek Distance

NOTES

There are two low d and high d integral values for a given search distance, which specify a portion of one edge of the reachable field. As shown in Figure 13.2, the likelihood of a non-miss is one for any rotational latency rl of at least as large as high d and the probability is zero for rl of less than low d . A standard table would contain the estimated access time evaluated as low d to high rl for each rotational latency rl , from low d to high d .

$$EAT = (1 p) (rl + 90) + p - r$$

Check Your Progress

7. What is the problem with FCFS scheduling with disks?
8. Write a short note on SSTF scheduling.
9. Describe the SCAN algorithm.
10. How does the C-SCAN method vary from the SCAN method?
11. Where should disk directories be placed?

13.7 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. A sector is the smallest block that can be read or written on a disk. A track is a collection of sectors on the same circumference on a single surface. A cylinder is a collection of all tracks of the same radius on a disk.
2. The total number of bytes transferred divided by the total time between the first request for service and the completion on the last transfer.
3. Information on a disk is referenced by the drive number, surface, track, and sector.
4. Seek time is the time for the read/write head to find the desired cylinder.
5. Latency time is the time for the disk to rotate to the start of the desired sector.
6. The characteristics determining the disk access speed are as follows:
 - **Seek time:** Time for head to reach specified track.
 - **Latency time:** Determined by the rate of rotation.
 - **Transfer time:** Determined by the rate of rotation, amount of data to be transferred and the density of the data on the disk.
7. Head may swing wildly from track to track, temporarily skipping some tracks that need to be used. Very inefficient.

8. Shortest Seek Time First: It processes I/O in tracks closest to the current position of the head.
9. Scan algorithm starts with the lowest track with an I/O request and processes requests until the highest is found; then proceeds in the reverse order.
10. After reaching the highest track number request, it returns to the lowest track number request, without processing any en route.
11. Near the middle tracks.

Disk Scheduling

NOTES

13.8 SUMMARY

- Disks are used to store bulk data as secondary storage for modern computers.
- A hard disk has a device controller, which translates numerical addresses into head movements. The device controller gets the instructions for execution from the operating system.
- A Small Computer System Interface (SCSI) drive is the most popular drive used for large personal computers and workstations.
- On simple disks, formatting is done manually. On complex disks like SCSI drives, a low-level formatting on the disk is done by the manufacturer. High-level formatting is not necessary when a file system is able to manage the hardware sectors.
- Data consistency is checked by writing data to the disk and reading back the data. An error occurs if there is any disagreement. Some device controllers detect bad sectors and move data to a good sector if there is an error.
- Another way of checking data consistency is to calculate a number for each sector; the number is calculated on the basis of what data is present in the sector. The calculated number is stored in the sector.
- When data is read back, the number is recalculated and if there is disagreement, then an error is generated. This is called cyclic redundancy check (CRC) or error correcting code.
- The disk drive rotates at the speed 60 to 200 rotations per second. Transfer rate is the rate at which data flows between a drive and a computer. Positioning time (random-access time) is the time for moving the disk arm to the desired cylinder and is also called seek time.
- The time for the desired sector to rotate under the disk head is called rotational latency. Head crash results from the disk head making contact with the disk surface. Host controller in computer uses bus to talk to disk controller built into the drive or storage array.

NOTES

- The operating system is responsible for the efficient use of the disk drives; this means having fast access time and a high bandwidth for the disks. The disk header moves only in the forward or backward direction.
- Physical memory is limited in size and is not permanent. We need a secondary storage, which is a collection of disks and magnetic tapes that can permanently store large amounts of data. Hard disks and tape drives are basic secondary storage media. These are efficient and cheaper.
- The most common types of disks are floppy disks, hard disks and diskette.
- As hard disk units become more efficient and cheaper, floppy disks are no longer used. A hard disk is composed of several disks arranged as a stack of disks one on top of the other. Each disk has a separate read/write head.
- Stepper motor is used to control the heads, which moves across each surface, which has a fixed number of tracks on each surface. All the tracks on all the surfaces of the disks are aligned.
- A cylinder is the sum of all the tracks that are at a fixed distance from one end of the disk.
- A hard disk has a device controller, which translates numerical addresses into head movements.
- The physical properties of the disk are number of tracks, sectors per track, speed of revolution, and number of heads.
- The operating system is responsible for the efficient use of disk drives. The disk header moves only in the forward or backward direction.
- Access time has three major components. Seek time is the time required for the disk to move the heads to the cylinder containing the desired sector. Rotational latency is the time required for the disk to rotate the desired sector.
- Transfer time is the time taken to transfer the data present in the blocks in a sector.
- The efficiency of an operating system directly depends on the seek time. The main objective of an operating system is to minimize seek time. Scheduling is done to improve the average disk access time. Speed of access depends on Seek time. Each disk drive has a queue of pending requests.
- A request can be an input request (write request) or output (read request). Disk address consists of disk number, cylinder number, surface number, sector number and the number of blocks to transfer.
- The data in the blocks is transferred into the main memory address. Block numbers indicate the amount of information to be transferred, i.e. the byte count.

- The servicing of disk I/O requests can be done by using disk scheduling algorithms.
- First Come First Serve scheduling is the simplest and fair but may not provide the best possible service.
- Shortest Seek Time First scheduling selects the request with the minimum seek time from the current head position. It services all requests close to the current head position before moving the head far away. It may cause starvation of some requests and it is not optimal.
- In SCAN scheduling, the disk arm starts at the one end of the disk and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Circular SCAN or C-SCAN is a variation of SCAN scheduling. It moves the head from one end to the other. Upon reaching the other end, it immediately comes back to the first end without servicing any requests on the way.
- LOOK scheduling looks for a request before moving in that direction and moves the head only as far as the last request in that direction. If no more requests are pending in the current direction, it reverses the head movement.
The performance of scheduling algorithms is heavily dependent on numbers and types of requests. Placing directories halfway between the inner and outer tracks of disk reduces head movement.
- SSTF is common and has a natural appeal. SCAN and C-SCAN perform better for systems that place a heavy load on the disk. Requests for disk service can be influenced by the file allocation method. Either SSTF or LOOK is a reasonable choice for the default algorithm.
- To improve performance, the directory structure is cached for faster access. The disk scheduling algorithms are complex to implement; hence, they are written in a separate module of the operating system.

Disk Scheduling

NOTES

13.9 KEY WORDS

- **Swap space management:** A high-priority process is swapped into the main memory and a low-priority process is swapped out of the main memory. The swapped-out process is stored into the secondary (backing) storage. The space in the storage where the swapped-out processes are stored is called swap space. Techniques used to manage the swap space are known as swap space management.
- **Cyclic redundancy check:** Data consistency is checked by writing data to the disk and reading back the data. An error occurs if there is any

NOTES

disagreement. Some device controllers detect bad sectors and move data to a good sector if there is an error. Another way of checking data consistency is to calculate a number for each sector; the number is calculated on the basis of what data is present in the sector. The calculated number is stored in the sector. When data is read back, the number is recalculated and if there is disagreement, then an error is generated. This is called cyclic redundancy check (CRC) or error correcting code.

- **Transfer rate:** The rate at which data flows between a drive and a computer.
- **Positioning time:** The time for moving the disk arm to the desired cylinder and is also called seek time.
- **Rotational latency:** The time for the desired sector to rotate under the disk head is called rotational latency.
- **Disk bandwidth:** The total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.
- **Shortest Seek Time First scheduling:** Shortest Seek Time First (SSTF) scheduling selects the request with the minimum seek time from the current head position. SSTF scheduling is a form of SJF scheduling and it may cause starvation of some requests.
- **SCAN:** In SCAN scheduling, the disk arm starts at the one end of the disk and moves toward the other end, servicing requests until it gets to the other end of the disk where the head movement is reversed and servicing continues. This is also called elevator algorithm because of similarity with building elevators.
- **Circular SCAN (C-SCAN):** Circular SCAN is a variation of SCAN scheduling. It moves the head from one end to the other. Upon reaching the other end, immediately come back to the first end without servicing any requests on the way.
- **LOOK scheduling:** In this scheduling, the head moves only as far as the last request in that direction. No more requests in the current direction; reverses the head movement. Look for a request before moving in that direction.

13.10 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Briefly describe the structure of a disk.
2. Explain disk arm scheduling algorithm.

3. What are the various scheduling criteria for CPU scheduling?
4. Which of the algorithms tends to minimize the process flow time?
5. Write a short note on:
 - Transfer rate
 - Positioning time

Disk Scheduling

NOTES

Long-Answer Questions

1. Suppose that a disk drive has 5000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests in the FIFO order is 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130
Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk scheduling algorithms?
 - a. FCFS
 - b. SSTF
 - c. SCAN
 - d. LOOK
 - e. C-SCAN
2. Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer. What are the different advantages and disadvantages of layered architecture?
3. What is RR scheduling algorithm?
4. Difference between FCFS and C-SCAN disk scheduling algorithm.
5. Elaborate the concept of rotational optimization.

13.11 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.
- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.

NOTES

- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.
- Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.
- Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.
- Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.

UNIT 14 FILE AND DATABASE SYSTEM

NOTES

Structure

- 14.0 Introduction
- 14.1 Objectives
- 14.2 File Concept
 - 14.2.1 File Attributes
 - 14.2.2 File Operations
 - 14.2.3 File Types
 - 14.2.4 File Structure
- 14.3 Access Methods
 - 14.3.1 Sequential Access
 - 14.3.2 Direct Access
- 14.4 Data Hierarchy
 - 14.4.1 Purpose of the Data Hierarchy
 - 14.4.2 Components of the Data Hierarchy
- 14.5 Directory Structure
 - 14.5.1 Single-Level Directory
 - 14.5.2 Two-Level Directory
 - 14.5.3 Hierarchical Directory
- 14.6 Protection
 - 14.6.1 Types of Access
 - 14.6.2 Access Control
- 14.7 File System Structure
 - 14.7.1 File System Implementation
- 14.8 Directory Implementation
 - 14.8.1 Linear List
 - 14.8.2 Hash Table
- 14.9 File Optimization
 - 14.9.1 7 Things You Should know about File Optimization
 - 14.9.2 How File Optimization Works
- 14.10 Allocation Methods
 - 14.10.1 Contiguous Allocation
 - 14.10.2 Linked Allocation
 - 14.10.3 Indexed Allocation
- 14.11 Free Space Management
 - 14.11.1 Bit Vector
 - 14.11.2 Linked List
 - 14.11.3 Grouping
 - 14.11.4 Counting
- 14.12 Efficiency and Performance
 - 14.12.1 Efficiency
 - 14.12.2 Performance
- 14.13 Recovery
 - 14.13.1 Backup and Restore
 - 14.13.2 Consistency Checking

NOTES

- 14.14 File Access Control
 - 14.14.1 Setting Permissions
- 14.15 Answers to Check Your Progress Questions
- 14.16 Summary
- 14.17 Key Words
- 14.18 Self Assessment Questions and Exercises
- 14.19 Further Readings

14.0 INTRODUCTION

A file is a set of related data that is contained in a secondary storage device. Or file is a list of entities that are connected logically. A file is the smallest allotment of logical secondary storage from the user's perspective.

Computer applications require large amounts of data to be stored that can be used later as and when required. For this, secondary storage devices, such as floppy disk, magnetic disk, magnetic tape and hard disk are used. The storage of data on secondary storage devices makes the data persistent; that is, the data is permanently stored and can survive system failures and reboots.

To store and retrieve files on the disk, an operating system provides a mechanism called the file system. This system is a part of the operating system that is primarily responsible for the management and organization of various files in a system.

A directory structure is the way an operating system's file system and its files are displayed to the user. Files are typically displayed in a hierarchical tree structure.

The control of file access operates just like a bank. A vault with safety deposit boxes is inside your local bank where you can store your valuables, such as the deed to your house, understanding that without keys to the vault and the key to your safety deposit box, no one can access the deed. In a similar way, critical computer files can be secured by the file access control function of the operating system. We'll discuss how an operating system can secure files and directories or allow access to them in this lesson.

The programme that manages and organises the files in a storage medium is a file system. It controls how it stores and retrieves information. A software framework is a DBMS or a Database Management System. It is used for database accessing, developing, and handling.

If a computer's file system is irrevocably lost, whether due to hardware or software restoring all the information will be difficult, time consuming and in many cases impossible. So it is advised to always have file-system backups.

In this unit, you will study about the file and database system, data hierarchy, files, file systems, file optimization, file allocation, free space management, and file access control.

14.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the concept of files
- Discuss the aspects related to file systems
- Explain the different file access methods
- Analyse the concept of data hierarchy
- Describe the various types of directory structures
- Explain the protection mechanism
- Describe the file system structure
- Know about the concept of file optimization
- Elaborate basic concepts of file system implementation
- Explain how directory implementation is done
- Describe the various disk allocation methods
- Analyse the issues related to system efficiency and performance
- Know how to activate the recovery of a system in the event of a failure
- Understand the file access control

NOTES

14.2 FILE CONCEPT

As stated in the introduction, a system stores data on various storage devices. The operating system, however, for the convenience of use of data on these devices provides a uniform logical view of the data storage to the users. The operating system abstracts from the physical properties of its storage devices and defines a logical storage unit known as a **file**. This allows user to directly access the data (on physical devices) without knowing where the data is actually stored.

A file is a collection of related data stored as a named unit on the secondary storage. It can store different types of data, like text, graphic, database, executable code, sound, videos, etc. and on the basis of the data, a file can be categorized as a data file, graphic file, database file, executable file, sound file, video file, etc.

Moreover, the structure of a file is based on the type of the file. For example, a graphic file is an organized collection of pixels, a database file is a collection of tables and records, and a batch file is a collection of commands.

Note: From a users' view, it is not possible to write data directly to a storage device until it is within a file.

14.2.1 File Attributes

A file in a system is identified by its name. The file name helps a user to locate a specific file in the system. Different operating systems follow different file naming

NOTES

conventions. However, most operating systems accept a file name as a string of characters, or numbers or some special symbols as well. For instance, names, such as `alice`, `tom`, `3546`, `!hello` and `table2-1` are all valid file names. Note that some operating systems distinguish the upper and lower case characters in the file names. For instance, in UNIX the file names `Alice`, `alice`, `ALICE` refer to three different files whereas, in DOS and Windows they refer to the same file.

Apart from the file name, some additional information (also known as **file attributes**) is also associated with each file. This information helps the file system to manage a file within the system. The file attributes related to a file may vary in different operating systems. Some of the common file attributes are as follows.

- **Name:** Helps to identify and locate a file in the system.
- **Size:** Stores information about the current size of the file (in bytes, words, or blocks).
- **Type:** Helps the operating system to recognize and use the recommended program to open a particular file type. For instance, to open an mpeg (multimedia) file, operating system uses a media player.
- **Identifier:** A unique tag, usually a number that helps the file system to recognize the file within the file system.
- **Location:** A pointer that stores location information of the device and location of the file on that device.
- **Date and Time:** Stores information related to a file, such as, creation, last modification and last use. Such information may be useful in case of protection, security and monitoring, etc.
- **Protection:** Stores information about the access permissions (read, write, execute) of different users. For example, it may specify who can access the file and which operations can be performed on a file by a user.

Figure 14.1 shows the list of some attributes that MS DOS attaches to a file.

Date	Time	Size	Name and Type
01/30/2009	03:33 PM	323	EXC.CPP
12/18/2008	01:40 PM	22,058,104	antivir_workstation_winu_en_h.exe
02/03/2009	01:21 PM	3	src.txt

Fig. 14.1 File Attributes of MS DOS

The information related to a file is stored as a directory entry in the directory structure. The directory entry includes the file's name and the unique identifier. The identifier in turn locates the other file attributes.

14.2.2 File Operations

File operations are the functions that can be performed on a file. An operating system handles the file operations through the use of system calls. The various operations that can be performed on a file are create, write, read, seek, delete, open, append, rename and close a file.

NOTES

- **Create a File:** To bring a file into existence, the `create` system call is used. When this system call is used, the operating system searches for the free space in the file system and allocates it to the file. In addition, the operating system makes a directory entry to record the name, location and other information about the file.
- **Open-File:** To open a file, the `open` system call is used which accepts the file name and the access-mode (read, write, execute) as parameters and returns a pointer to the entry in the **open-file table** (a table in the main memory that stores information about the files that are opened at a particular time). The operating system searches the directory entry table for the file name and checks if the access permission in directory entry matches the request. If that access-mode is allowed, it then copies the directory entry of the file to the open-file table.
- **Write to a File:** To store data into a file, the `write` system call is used which accepts the file name and the data to be written to the file as parameters. The operating system searches the directory entry to locate the file and writes data to the specified position in the file and also updates the write pointer to the location where next write operation is to take place.
- **Read a File:** To retrieve data from a file, the `read` system call is used which accepts the file name, amount of data to be read and a read pointer to point to the position from where the data is to be read as parameters. The operating system searches the specified file using the directory entry, performs the read operation and updates the pointer to the new location. Note that since a process may be only reading or writing a file at a time, a single pointer called **current position pointer** can be used for both reading and writing. Every time a read or write operation is performed, this pointer must be updated.
- **Seek File:** To position the pointer to a specific position in a file, the `seek` system call is used. Once the pointer is positioned, data can be read from and written to that position.
- **Close File:** When all the operations on a file are completed, it must be closed using the `close` system call. The operating system searches and erases the file entry from the open-file table to make space for the new file entries. Some systems automatically close a file when the process that has opened the file terminates.
- **Delete File:** When a file is not required, the `delete` system call is used. The operating system searches the file name in the directory listing. Having found the associated entry, it releases all space allocated to the file (that can be reused by other files) by erasing its corresponding directory entry.
- **Append File:** To add data at the end of an existing file, `append` system call is used. This system call works similar to the `write` system call, except that it positions the pointer to the end of file and then performs the write operation.
- **Rename File:** To change the name of an existing file, `rename` system call is used. This system call changes the existing entry for the file name in the directory to a new file name.

NOTES

14.2.3 File Types

As stated in the above sections, files can be of different types. The operating system can handle a file in a reasonable way only if it recognizes and supports that file type. A user request to open an executable file with a text editor will produce garbage if the operating system has not been told that it is an executable file.

The most common technique to implement a file type is by providing extension to a file. The file name is divided into two parts, with the two parts separated by a period ('.') symbol, where the first part is the **name** and the second part after the period is the **file extension**. A file extension is generally one to three characters long, it indicates the *type* of the file and the operations (read, write, execute) that can be performed on that file. For example, in the file name `Itlesl.doc`, `Itlesl` is the name and `.doc` is the file extension. The extension `.doc` indicates that `Itlesl.doc` is a document file and should be opened with an editor. Similarly, a file with `.exe` or `.com` extension is an executable file. Table 14.1 lists various file types, extension and their meaning.

Table 14.1 File Types and Extensions

File type	Extension	Meaning
Archive	<code>arc, zip, tar</code>	Related files compressed and grouped together into single file for storage
Batch	<code>bat, sh</code>	An executable file stores a series of commands that can be executed with a single command
Backup file	<code>bak, bkf</code>	Stores a copy of the data on the disk, used for recovering system crash
Executable	<code>exe, com, bin</code>	Used to run various programs on a computer
Library	<code>lib, a, so, dll,</code>	Stores libraries of routines for programmers
Image	<code>bmp, jpeg, gif, jfif, dib</code>	Stores images and graphics
Multimedia	<code>mpeg, mp2, mpa, mpe</code>	Stores audio and video information
Object	<code>obj, o</code>	Machine language file, precompiled, used for generating output
System file	<code>inf, ini,drv</code>	Stores system information for loading and managing different applications
Text	<code>txt, doc</code>	Stores textual data, documents
Word processor	<code>wp, rrf, doc</code>	Stores various word processor formats

File extensions help the operating system to know about the application program that has created the file. For instance, the file with `.txt` extension will be opened with a text editor and the file with `.mp3` extension will be opened with a music player supporting the `.mp3` files. Note that the operating system automatically opens the application program (for the known file types) whenever a user double clicks the file icon.

NOTES

Some operating systems, such as UNIX, support the use of double extension to a file name. For example, the file name `file1.c.z` is a valid file name, where `.c` means that `file1` is a C language file and `.z` means the file is compressed using some zip program. A file extension can be system defined or user defined.

Another way to implement the file type is the use of **Magic Number**. A magic number is a sequence of bits, placed at the starting of a file to indicate roughly the type of file. The UNIX system makes use of magic number to recognize the file type. However, not all its files have magic numbers. UNIX system allows file-name-extension hints to help its user determine the type of contents of the file, it.

14.2.4 File Structure

The file structure refers to the internal structure of the file, that is, how a file is internally stored in the system. The most common file structures recognized and enforced by different operating systems are as follows:

- **Byte Sequence:** In this file structure, each file is made up of a sequence of 8-bit bytes (see Figure 14.2(a)) having no fixed structure. The operating system does not attach any meaning to the file. It is the responsibility of the application program to include code to interpret the input file into an appropriate structure. This type of file structure provides flexibility to the user programs as they can store any type of data in the files and name these files in any way as per their convenience. UNIX operating systems support this type of file structure.
- **Record Sequence:** In this file structure, a file consists of a sequence of fixed-length records where arbitrary number of records can be read from or written to a file. The records cannot be inserted or deleted in the middle of a file. In this system, the read operation returns one record and the write operation appends or overwrites one record. CP/M operating system supports this type of scheme.

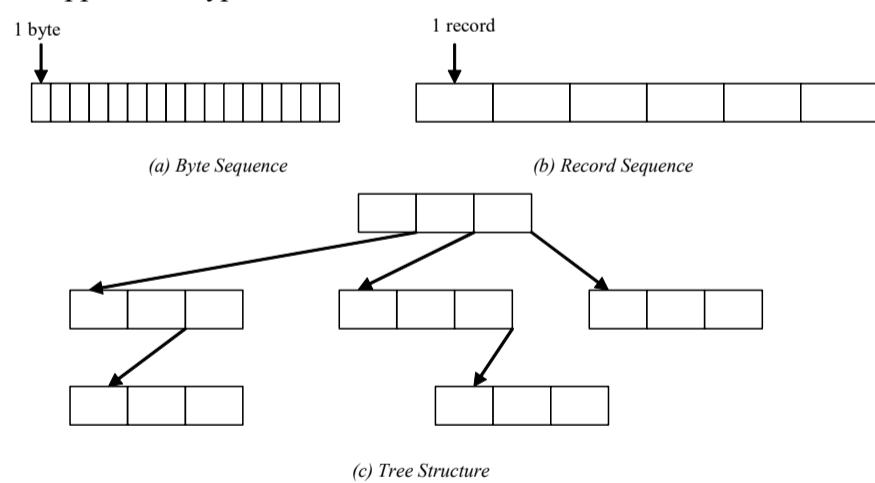


Fig. 14.2 File Structures

NOTES

- **Tree Structure:** In this file structure, a file consists of a tree of disk blocks where each block holds a number of records of varied lengths. Each record contains a key field at a fixed position. The records are searched on key value and new records can be inserted anywhere in the file structure. This type of file structure is used on mainframe system where, it is called **ISAM (Indexed Sequential Access Method)**.

Regardless of the file structure used, all disk I/O take place in terms of blocks (physical records) where all blocks are of equal sizes and the size of a block is generally determined by the size of the sector. Since the disk space to a file is allocated in a number of blocks, some portion of the last block in a file is generally wasted. For instance, if each block is of 512 bytes, then a file of 3150 bytes would be allocated seven blocks, and the last 434 bytes will be wasted. The wastage of bytes to keep everything in units of blocks (instead of bytes) is internal fragmentation. Note that all file systems face internal fragmentation and with larger block sizes, there is more internal fragmentation.

14.3 ACCESS METHODS

The information stored in the file can be accessed in one of the two ways: sequential access or direct access.

14.3.1 Sequential Access

When the information in the file is accessed in order, one record after the other, it is called **sequential access**. It is the easiest file access method. Compilers, multimedia applications, sound files and editors are the most common examples of the programs using sequential access.

The most frequent and common operations performed on a file are read and write operations. In case of read operation, the record at the location pointed by the file pointer is read and the file pointer is then advanced to the next record. Similarly, in case of write operation, the record is written to the end of the file and pointer is advanced to the end of new record.

14.3.2 Direct Access

With the advent of disks as a storage media, large amount of data can be stored on it. Sequential access of this data would be very lengthy and slow process. To overcome this problem, the data on the disk is stored as blocks of data with index numbers which helps to read and write data on the disk in any order (known as **random or direct access**).

Under direct access, a file is viewed as a sequence of blocks (or records) which are numbered. The records of a file can be read or written in any order using this number. For instance, it is possible to read block 20, then write block 4,

NOTES

and then read block 13. The block number is a number given by the user. This number is relative to the beginning of the file. This relative number internally has an actual absolute disk address. For example, the record number 10 can have the actual address 12546 and block number 11 can have the actual address 3450. The relative address is internally mapped to the absolute disk address by the file system. The user gives relative block number for accessing the data without knowing the actual disk address. Depending on the system, this relative number starts with either 0 or 1 for a file.

In direct access, the system calls for read and write operations are modified to include the block number as a parameter. For instance, to perform the read or write operation on a file, user gives `read n` or `write n` (`n` is the block number) rather than `read next` or `write next` system calls used in sequential access.

Most applications with large databases require direct access method for immediate access to large amounts of information. For example, in a railway reservation system, if a customer requests to check the status for reservation of the ticket, the system must be able to access the record of that customer directly without having the need to access all other customers' records.

Note that an operating system may support either sequential access or direct access, or both for accessing the files. Some systems require a file to be defined as sequential or direct when it is created, so that it can be accessed in the way it is declared.

14.4 DATA HIERARCHY

Data Hierarchy refers to, often in a hierarchical form, the formal arrangement of data. Lands, databases, files and so on are involved in data organisation. A data field contains an entity's single fact or attribute. “September 19, 2004”¹⁹ September 2004. This can be treated as a single date field, or as 3 fields, i.e. month, month, month, day, and year. A set of similar fields is a record. An Employee Record can contain a field of name, field of address, field of birthdate, etc. A file is a set of documents that are linked. If there are 100 employees, each employee will have a record and a file would represent the compilation of 100 such records. In a folder, files are combined. A Database Management System is used to do this. If we want to capture other aspects of employee data, then other files could also be generated, such as the Employee Training History file and the Employee Job History file. The above is a view of data that a computer user has used. In the hierarchical model, which is one way of organising data in a database, the structure above can be seen.

14.4.1 Purpose of the Data Hierarchy

“Data hierarchy” is a basic concept in the theory of data and databases and helps to demonstrate the relationships in a database or data file between smaller and

NOTES

larger components. It is used to provide a better sense of knowledge about the data components and how they are connected.

In databases with referential integrity, third normal type or perfect key, this is particularly important. “Data hierarchy” is the product of correct arrangement without redundancy of the data. Ultimately, preventing duplication leads to a proper “Data hierarchy” reflecting the data relationship and exposing its relational structure.

14.4.2 Components of the Data Hierarchy

The data hierarchy components are described below.

A **data field** contains an entity’s single fact or attribute. Take into account the date area, e.g. ‘19 September 2004.’ This can be treated as a single date field (e.g. date of birth) or three fields, namely, a month, month, and year date.

A set of similar fields is a **record**. A name field(s), address field, birthdate field, and so on can be included in an employee record.

A **file** is a set of documents that are linked. If there are 100 workers, each employee will have a record (e.g. named Employee Personal Information record) and a file would be a list of 100 such records (in this case, called Employee Personal Details file).

In a folder, files are combined. A **Database** Management System is used to do this. If we want to capture other aspects of employee data, then other files could also be generated, such as the Employee Training History file and the Employee Job History file.

Check Your Progress

1. The additional information that helps the operating system to manage a file within the file system is called _____.
2. In Windows, the file system distinguishes upper case and lower case characters in a file name, it treats ‘comp’ and ‘COMP’ as two different files. (True/False)
3. Which system supports double extensions to a file name?
4. The data on a disk is kept as blocks of data with an _____ to access data directly in random order.
5. A directory is a flat file that stores information about files and sub-directories. (True/False)
6. What do you mean by data hierarchy?
7. What is data field?
8. What is the difference between absolute path name and relative path name?

NOTES

14.5 DIRECTORY STRUCTURE

As stated earlier, a computer stores numerous data on disk. To manage this data, the disk is divided into one or more partitions (also known as **volumes**) and each partition contains information about the files stored in it. This information is stored in a **directory** (also known as **device directory**). Figure 14.3 shows different file-system organization.

Different operations that can be performed on a directory are as follows.

- **Create a File:** New files can be created and added to a directory by adding a directory entry in it.
- **Search a File:** Whenever a file is required to be searched, its corresponding entry is searched in the directory.
- **List a Directory:** All the files along with their contents in the directory entry are listed.
- **Rename a File:** A file can be renamed. A user might need to rename the file with the change in its content. When a file is renamed its position within the directory may also change.
- **Delete a File:** When a file is no longer required, it can be deleted from the directory.
- **Traverse the File System:** Every directory and every file within a directory structure can be accessed.

Note: A directory is a flat file that stores information about files and subdirectories.

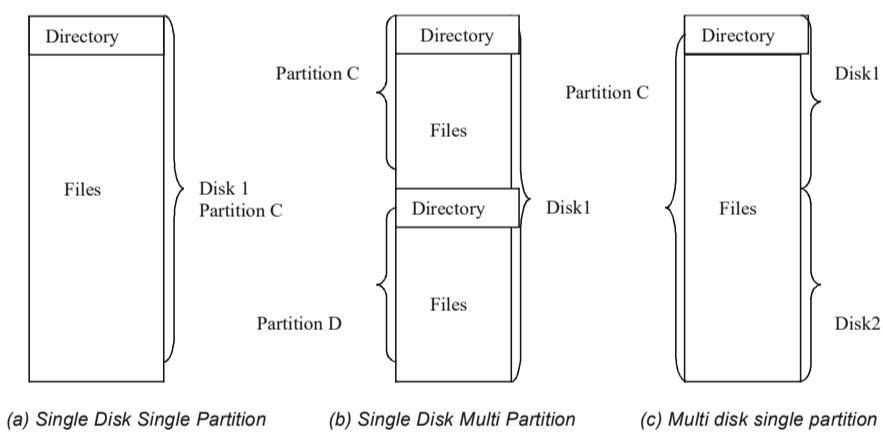


Fig. 14.3 Various File System Organization Schemes

Note: It is possible to have more than one operating system on a single disk, where a user can boot any of the operating system according to the need.

There are various schemes to define the structure of a directory. The most commonly used schemes are as follows.

NOTES

- Single-level directory
- Two-level directory
- Hierarchical directory

All these schemes are discussed in subsequent section.

14.5.1 Single-Level Directory

Single-level directory is the simplest directory structure. There is only one directory that holds all the files. Sometimes this directory is referred to as **root directory**. Figure 14.4 shows a single-level directory structure having five files. In this figure, box represents directory and circles represent files.

The main drawback of this system is that no two files can have the same name. For instance, if one user (say, jojo) creates a file with name `file1` and then another user (say, abc) also creates a file with the same name, the file created by the user abc will overwrite the file created by the user jojo. Thus, all the files must have unique names in a single-level directory structure. With the increase in the number of files and users on a system, it becomes very difficult to have unique names for all the files.

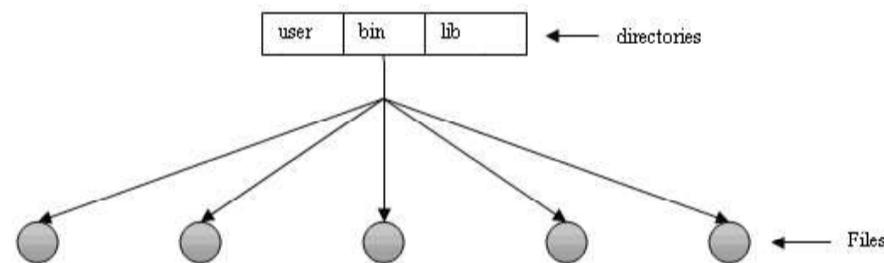


Fig. 14.4 Single-Level Directory Structure

14.4.2 Two-Level Directory

In a two-level directory structure, a separate directory known as **User File Directory (UFD)** is created for each user. Whenever, a new UFD is created, an entry is added to the **Master File Directory (MFD)** which is at the highest level in this structure (see Figure 14.5). When a user refers to a particular file, first, the MFD is searched for the UFD entry of that user and then the file is searched in the UFD.

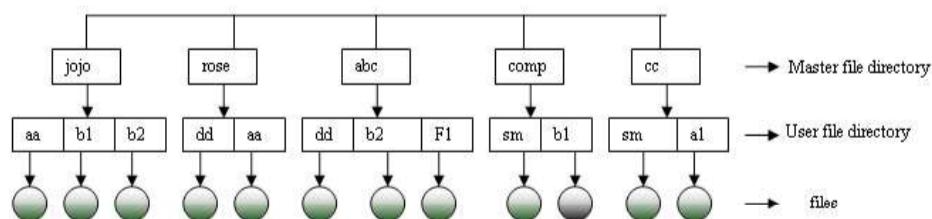


Fig. 14.5 Two-Level Directory Structure

NOTES

Unlike, single-level directory structure, only the file names should be unique in a two-level directory. That is, there may be files with same name in different directories. Thus, there will not be the problem of name-collision in this directory structure but there is one disadvantage that the users in this directory structure are not allowed to access files of other users. If a user wants to access a file of other user, he needs special permissions from the administrator. In addition, to access other users' file, the user must know the correct **path name** (which includes the user name and the file name). Note that different systems use different syntax for file naming in directories. For instance, in MS DOS, to access the file in the `sm` directory, the user gives `//comp/sm`, where `//` refers to root, `comp` is the user name, `sm` is the directory.

In some situations, a user might need to access files other than their own file. One such situation might occur with system files. The user might want to use system programs like compilers, assemblers, loaders, or other utility programs. In such a case, to copy all the files in every user directory would require a lot of space and thus, would not be feasible. One possible solution to this is to make a special user directory and copy system files into it. Now, whenever a filename is given, it is first searched in the local UFD, and if not found there then the file is searched in the special user directory that contains system files.

14.4.3 Hierarchical Directory

The hierarchical directory, also known as **tree of directory** or **tree-structured directory**, allows users to have subdirectories under their directories, thus making the file system more logical and organized for the user. For instance, a user may have directory `furniture`, which stores files related to types of furniture say `wooden`, `steel`, `cane`, etc. Further, he wants to define a subdirectory which states the kind of furniture available under each type say `sofa`, `bed`, `table`, `chair`, etc. Under this system, the user has the flexibility to define, group and organize directories and sub-directories according to his requirements.

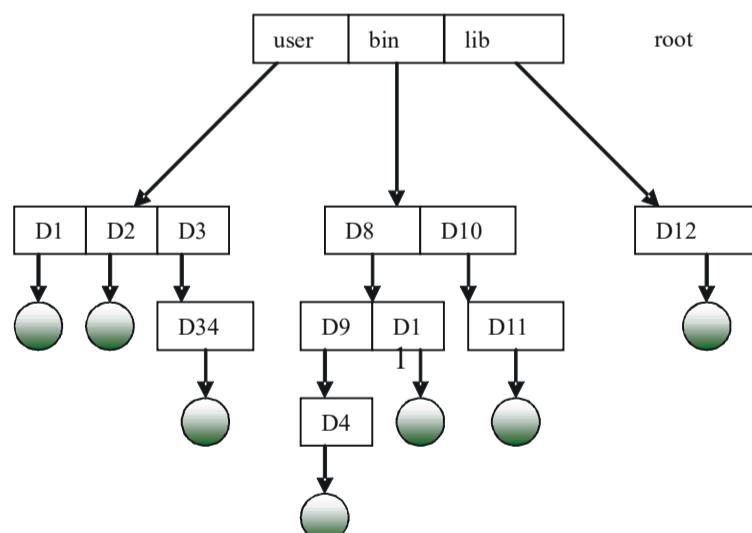


Fig. 14.6 Hierarchical Directory Structure

NOTES

The hierarchical directory structure has the root directory at the highest level, which is the **parent** directory for all directories and subdirectories. The root directory generally consists of system library files. All files or directories at the lower levels are called **child** directories and a directory with no files or subdirectory is called a **leaf**. Every file in the system has a unique path name. A path name is the path from the root, through all the sub-directories, to a specified file. Figure 14.6 shows the hierarchical directory structure having different levels of directories, subdirectories and related files.

The user under hierarchical directory system can access files of other users in addition to his own files. To access the files the user can specify either absolute path name or relative path name. The **absolute path name** begins at the root and follows a path down to the specified file or using the **relative path name** that defines a path from the current working directory. For instance, to access a file under directory D1, using absolute path name, the user will give the path \\bin\\D8\\D1\\filename. On the other hand, if the user's current working directory is \\bin\\D8, the relative path name will be D1\\filename.

In this structure, the major concern is the deletion of the files. If a directory is empty it can be deleted simply; however, if the directory contains subdirectories and files, they need to be handled first. Some systems, for example, MS DOS requires a directory to be completely empty before the delete operation can be performed on it. The user will have to delete all the files, subdirectories, files in subdirectories before performing the delete operation on a directory. Whereas, other systems, for example, UNIX is flexible as it allows user to delete a complete directory structure containing files and subdirectory with a single `rm` command. Though it is easy for a user to handle delete operation on directory under the UNIX system, but it increases the chances of accidental deletion of files.

Note: MS DOS, WINDOWS, and UNIX are some of the examples of systems using hierarchical directory structure.

14.6 PROTECTION

The information stored in a system requires to be protected from the physical damage and unauthorized access. A file system can be damaged due to various reasons such as, a system breakdown, theft, fire, lightning or any other extreme condition that is unavoidable and uncertain. It is very difficult to restore the data back in such conditions. In some cases, when the physical damage is irreversible, the data can be lost permanently. Though, physical damage to a system is unavoidable, measures can be taken to safeguard and protect the data.

In a single-user system, protection can be provided by storing a copy of information on the disk to the disk itself, or to some other removable storage media such as, magnetic tapes and compact disc. If the original data on the disk is accidentally erased or overwritten, or becomes inaccessible because of its malfunctioning, the backup copy can be used to restore the lost or damaged data.

Apart, from protecting the files from physical damage, the files in a system also need a protection mechanism to control improper access.

14.6.1 Types of Access

In a single-user system or in a system where users are not allowed to access the files of other users, there is no need for a protection mechanism. However, in a multi-user system where one user can access files of other users and the system is prone to improper access, a protection mechanism is a necessity. The access rights define the type of operation that a user is allowed to perform on a file. The different access rights that can be assigned to a particular user for a particular file are as follows.

- **Read:** Allow only reading the file.
- **Write:** Allow writing or rewriting the file.
- **Execute:** Allow running the program or application.
- **Append:** Allow writing new information at the end of the file.
- **Copy:** Allow creating a new copy of the file.
- **Rename:** Allow renaming a file.
- **Edit:** Allow adding and deleting information from the file.
- **Delete:** Allow deleting the file and releasing the space.

There are many protection mechanisms, each having some advantages and disadvantages. However, the kind of protection mechanism used depends on the need and size of the organization. A smaller organization needs a protection mechanism different from the protection mechanism of a larger organization with large number of people accessing the files.

NOTES

14.6.2 Access Control

To protect the files from improper accesses, the access control mechanism can follow either of the two approaches.

- **Password:** A password can be assigned to each file and only a user knowing the password can access the file. This scheme protects the file from unauthorized access. The main drawback of this approach is the large number of passwords which are practically very difficult to remember (for each file separately). However, if only one password is used for accessing all the files, then all the files become easily accessible if somebody knows the password. To balance the number of passwords in a system, some systems follow a scheme, where a user can associate a password with a subdirectory. This scheme allows a user to access all the files under a subdirectory with a single password. This scheme is also not very much safe. To overcome the drawbacks of these schemes, the protection must be provided at a more detailed level by using multiple passwords.

NOTES

- **Access Control List:** In this approach, access to a file is provided on the basis of identity of the user. An **Access-Control List (ACL)** is associated with each file and directory, it stores user names and the type of access allowed to each user. When a user tries to access a file, the ACL is searched for that particular file. If that user is listed for the requested access, the access is allowed. Otherwise, the user is denied access to the file. This system of access control is effective but, in case when all the users want to read a file, the ACL for that particular file should list all users with read permission. The main drawback of this system is that, making such a list would be a tedious job when number of users is not known. Moreover, the list need to be dynamic in nature as the number of users will keep on changing, thus resulting in complicated space management.

To resolve the problems associated with ACL, a restricted version of the access list can be used in which the length of the access list is shortened by classifying the users of the system into the following three categories:

- **Owner:** The user who created the file.
- **Group:** A set of users who need similar access permission for sharing the file is a group, or work group.
- **Universe:** All the other users in the system form the universe.

Based on the category of a user, access permissions are assigned. The owner of the file has full access to a file, and can perform all file operations (read, write and execute) whereas, a group user can read and write a file but cannot execute or delete a file. However, the member of the universe group can only read a file and is not allowed to perform any other operations on a file.

The above method of classifying users in groups will not work, when one user wants to access file of other user (for performing a specific file operation). For example, say, a user comp wants to access the file abc of other user comp1, for reading its content. To provide file-specific permissions to a user, in addition to the user groups, an access-control list is attached to a file. This list stores the user names and permissions in a specific format.

The UNIX operating system uses this method of access control, where the users are divided into three groups, and access permissions for each file are set with the help of three fields. Each field is a collection of bits, where three bits are used for setting protection information and an additional bit is kept for the file owner, for the file's group and for all other users. The bits are set as `-rwx`, where `r` controls read access, `w` controls write access and `x` controls execution. When all three bits are set to `-rwx`, it means a user has full permission on a file whereas, if only `-r-` field is set, it means a user can only read from a file and when `-rw-` bits are set, it means user can read and write but cannot execute a file. The scheme requires total nine bits, to store the protection information. The permissions for a file can be set either by an administrator or a file owner.

NOTES

14.7 FILE SYSTEM STRUCTURE

Every operating system imposes a file system that helps to organize, manage and retrieve data on the disk. The file system resides permanently on the disk. The design of the file system involves two key issues. The first issue includes defining a file and its attributes, operations that can be performed on a file, and the directory structure. The second issue includes creating data structures and algorithms for mapping the logical file system onto the secondary storage devices.

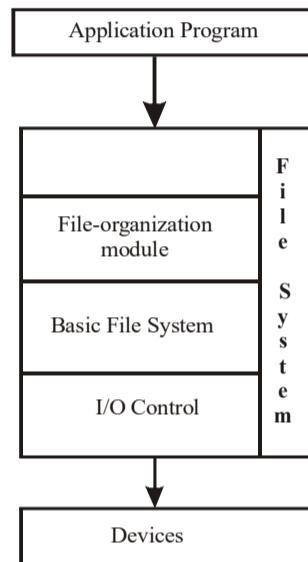


Fig. 14.7 File System Layers

Figure 14.7 shows that the file system is made up of different layers, where each layer represents a level. Each level uses the features of the lower levels to create new features that are used by higher levels. When the user interacts with the file system (through system commands), the I/O request occurs on a device. The **device driver** in turn generates an interrupt to transfer information between the disk drive and the main memory. High-level commands make the input to device driver, and its output is low-level hardware specific instructions. The device driver also writes specific bit pattern to special locations in the I/O controller's memory which tells the controller about the device location to act upon and the actions to be taken.

The next component in the file system is the **basic file system** that issues generic (general) commands to the appropriate device driver to read and write physical blocks on the disk. The physical blocks are referred to by the numeric disk address (for example, drive 1, cylinder 43, track 4, sector 16).

Next level component of the file system is the **file-organization module** which organizes the files. It knows the physical block address (the actual address) and logical block address (the relative address), allocation method, and location of a file. Using this information, it translates the logical address into physical address

NOTES

and helps basic file system to transfer files. The other function of the file organization module is to keep track of the free space and provide this space for allocation when needed.

The **logical file system** at the next level manages all the information about a file except the actual data (content of the file). It manages the directory structure and provides the necessary information to the file-organization module when the file name is given. It maintains file structure using the **File Control Block (FCB)** that stores information about a file such as ownership, permissions, and location of the file content. Protection and security is also taken care by the logical file system.

Apart from the physical disk drive, there are other removable devices also, such as CD-ROM, floppy disk, pen drives and other storage devices attached to a system. Each of these devices has a standard file system structure imposed by its manufacturers. For instance, most CD-ROMs are written in *High Sierra* format, which is a standard format agreed upon by CD-ROM manufacturers. The standard file system for the removable media makes them interoperable and portable for use on different systems. Apart from the file systems for removable devices each operating system has one (or more) disk-based file system. UNIX system uses the UNIX File System (UFS) as a base. Windows NT supports disk file system formats, such as, FAT, FAT32, and NTFS (or Windows NT file system), along with CD-ROM, DVD, and floppy-disk file system formats.

14.7.1 File System Implementation

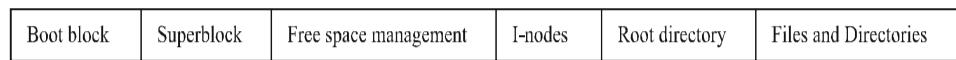
There are several **on-disk** and **in-memory** structures that are used to implement a file system. Depending on the operating system and the file system, these may vary, but the general principles remain the same. Many of the structures that are used by most of the operating system are discussed here. The on-disk structures include:

- **Boot Control Block:** It contains enough information that the system needs to boot the operating system from that partition. Though, not all the partitions in a disk contain a bootable operating system, every partition starts with a boot block. Having one block in each partition reserved for a boot block is a good idea because any partition can have operating system in the future. If the partition does not contain any operating system this block can be empty. In UNIX File System (UFS), this block is called the **boot block** and in Windows (NTFS), it is called the **partition boot sector**.
- **Partition Control Block:** It is a table that stores key information related to partition, number and size of blocks in the partition, free block count and free block pointers, and FCB pointers and free FCB count. In UFS this is called **superblock**, in NTFS, it is **Master File Table**.

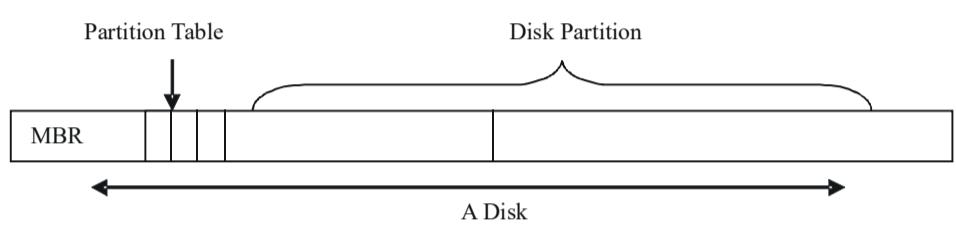
Further, each partition has a directory structure, with root directory at the top. The directory structure helps to manage and organize the files in the file system. A new FCB is allocated when creating a new file that stores information such as file permissions, ownership, size, and location of the data blocks. In UFS this is called the **i-node** (an array of data structures, one for each file). In NTFS, this

NOTES

information is kept within the Master File Table, which uses a relational database structure, where each row stores information about a file. Figure 14.8 shows an example of file system layout based on UNIX file system.



(b) Structure of a Partition



(a) Structure of a Disk

Fig. 14.8 File System Layout

The in-memory structure helps in improving performance of the file system. The in-memory structures include:

- **In-Memory Partition Table:** It stores information about each mounted partition.
- **In-Memory Directory Structure:** It stores the information about the recently accessed directories.
- **System-Wide Open-File Table:** It contains a copy of the FCB of each open file and a count of the number of processes that have the file open.
- **Per-Process Open-File Table:** It contains a pointer to the corresponding entry in the system-wide open-file table along with some related information.

When the open call passes a file name to the file system, the directory structure is searched for the given file name. When the file is found, its FCB is copied into the system-wide open-file table, and the count is incremented.

Once the system-wide open-file table is updated, an entry is made in the per-process open-file table. This entry includes a pointer to the appropriate entry in the system-wide open-file table, a pointer to the position in the file where the next read or write will occur, and the mode in which the file is open. The open call returns a pointer to the appropriate entry in the per-process file system table. This pointer is then used to perform all the operations as long as the file is open. It means that until a file is closed, all the operations are carried out on the open-file table.

When a process closes the file, the corresponding entry from the per-process open-file table is removed and the open count of the system-wide entry is decremented. The value in the count indicates the number of users who have opened the file currently. Thus, when this value becomes 0, the updated file information is copied back to the disk-based structures and the entry is removed from the system-wide open-file table.

NOTES

Check Your Progress

9. The information in a computer needs to be protected from _____ and _____.
10. The _____ and _____ are two most commonly used directory-management algorithms.
11. What is the use of an access control list?
12. Name the components of the file system that is responsible for transfer of information between the disk drive and the main memory.
13. A user interacts with the system by using the system calls. (True/False)
14. A _____ stores all the information related to a file.
15. What is file optimization?

14.8 DIRECTORY IMPLEMENTATION

The efficiency, performance and reliability of a file system are directly related to the directory-management and directory-allocation algorithms selected for a file system. The most commonly used directory-management algorithms are linear list and hash table.

14.8.1 Linear List

The linear list method organizes a directory as a collection of fixed size entries, where each entry contains a (fixed-length) file name, a fixed structure to store the file attributes, and pointers to the data blocks (see Figure 14.9). The linear list method stores all file attributes compiled as a single directory entry and uses a linear search to search a directory entry from the list of entries. It means to search an entry in the list of entries; each entry (starting from the first entry in the directory) is examined one by one until the desired entry is found. This is simple to program; however, with extremely large directories the search becomes very slow, which is a major disadvantage of this method. MS DOS and Windows operating system use this approach for implementing directories.

Directory	
Itles1	attributes
language	attributes
magazine	attributes
sports	attributes
computer	attributes

Fig 14.9 Example of Linear List Directory Entry (in MS DOS and Windows)

NOTES

Some systems follow a variation of linear list method for recording the file information. For example, in UNIX, the directory entry consists of two fields: the file name and the **I-node (index-node)** number. The i-node number contains the disk address of the I-node structure that stores the file attributes and the address of the file's data blocks (see Figure 14.10). With this approach, the size of the directory entry is very small. This approach has certain advantages over the linear list of directory.

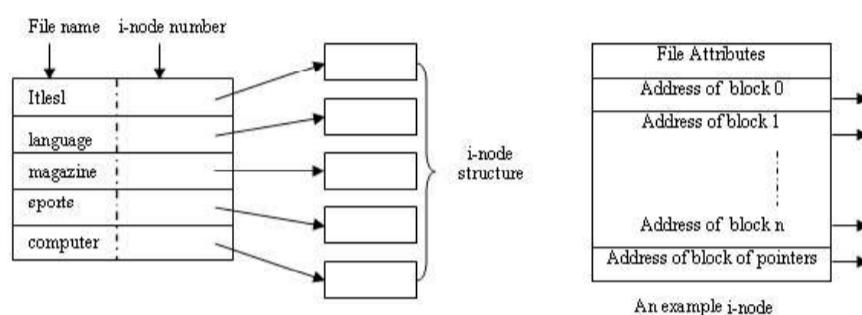


Fig. 14.10 Example of Linear List Directory Entry (in UNIX)

In both cases, when the user sends a request to create a new file, the directory is searched to check whether any other file has the same name or not. If no other file has the same name, the memory will be allocated and an entry for the same would be added at the end of the directory. To delete a file, the directory is searched for the file name and if the file is found, the space allocated to it is released. The delete operation results in free space that can be reused. To reuse this space, it can be marked with a used–unused bit, a special name can be assigned to it, such as all-zeros or it can be linked to a list of free directory entries.

When performing the file operations, one thing is common that is searching the directory for a particular file. The search technique applied greatly influences the time taken to make the search and in turn the performance and efficiency of the file system. As discussed, with long directories, a linear search becomes very slow and takes $O(n)$ comparisons to locate a given entry, where n is the number of all entries in a directory. To decrease the search time, the list can be sorted and a binary search can be applied. Applying binary search reduces the average search time but keeping the list sorted is a bit difficult and time-consuming, as directory entries have to be moved with every creation and deletion of file.

14.8.2 Hash Table

While implementing directories, the search time required to locate a directory entry corresponding to a file is always a major concern. To considerably reduce the search time, a more complex data structure known as a **hash table** along with the linear list of directory entry is used.

A **hash table** is a data structure, with 0 to $n-1$ table entries, where n is the total number of entries in the table (see Figure 14.11). It uses a hash function to compute a hash value (a number between 0 to $n-1$) based on the file name. For

NOTES

instance, file name is converted into integers from 0 to $n-1$ and this number is divided by n to get the remainder. Then, the table entry corresponding to the hash value (value of remainder) is checked. If the entry space is unused, then a pointer to the file entry (in the directory) is placed there. However, if the entry is already in use, we can say a **collision** has occurred. In such a situation, a linked list of entries that hash to the same value is created and the table entry is made to point to the header of the linked list.

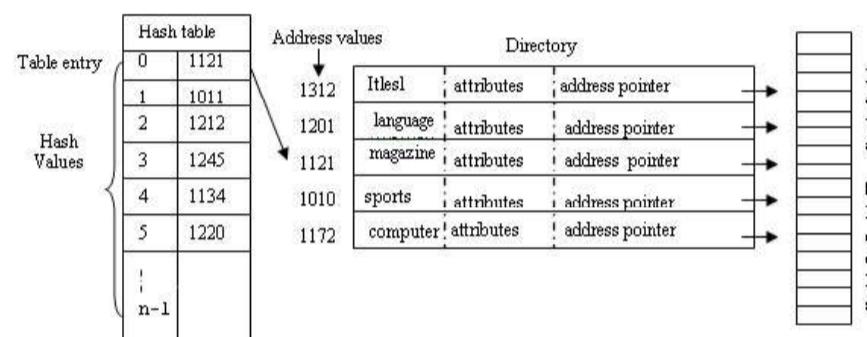


Fig. 14.11 Hash Table Linked to a Directory

We follow the same process to search a file. The file name is hashed to locate the hash table entry and then all the entries on the chain are checked to see if the file name exists. If the name is not found in the chain, the file is not present in the directory.

Despite all the advantages the main disadvantage of this approach is the creation of long overflow chains if the hash function is not distributing the values uniformly. Now, in order to search an entry, a linear search in the long overflow chain may be required, which increases the access time. In addition, the administration of hash table becomes a complex process. Some system copies the directory entries for frequently accessed files in the cache memory. This saves the time required to re-read information from the disk, thus, making the file access fast.

14.9 FILE OPTIMIZATION

The average worker is generating much larger digital documents than ever before, and in much larger volumes. The outcome is an overwhelming rise in the storage needed to house the user-generated files of an organisation. In spite of decreasing storage costs and a host of other storage management technologies, Sysadmins still have an eye on the potential to be more effective and keep ahead of their growth in storage.

14.9.1 7 things you Should know about File Optimization

- Optimization of files is extremely efficient at reducing Image-heavy document size, such as PDF, Word & PowerPoint files that are not well reduced with the use of other technologies for compression. Reason for the compression

NOTES

- never works since these files are predominantly contain data which has already been only compressed.
2. This greatly decreases the file size of standalone JPEGs - without their resolution being revised. This is what it does. Use the latest resources in the JPEG format to do so. Using more effective image decoding and re-encoding, Configuration. The resolution of the picture remains the same, but Consumes less disc space.
 3. The method is fully clear - it is simply transparent -Targets and extracts excess data from files, leaving excess data Smaller but similar in all other respects to the Oh, originals. This guarantees the file owners are not going to be aware of any updates to their files.
 4. There is a permanent reduction of optimised files - so Optimized files do not inflate to their original size if they are not optimised. They are interpreted or edited afterward.
 5. Optimization is a type of compression that is lossy, so the decline cannot be rolled back, but all the programme for Neuxpower file optimization has fair defaults and offers versatile configurations for handling the reduction level and to stop discarding something substantial.
 6. Smaller files that are configured do not have to be decompressed, meaning they open quicker, faster, and without the need to provide additional software.
 7. It is complementary to other types of compression -in addition to compression or compression, it can be used easily.to get even more savings with deduplication.

14.9.2 How File Optimization Works

File Optimization takes a distinct approach to tackling massive files compared with conventional compression. The exceptional output stems from the fact that many large files, such as PDFs and Microsoft Office documents, contain substantial hidden bloat that can be eliminated or ‘optimised’ to minimise the size of the file without altering the format of the file or impacting the file’s visual integrity. Within a text, File Optimization discusses individual components, determining how to minimise each component in turn. It will choose the most efficient method of reduction by evaluating each variable in isolation instead of applying general settings in the entire file. This method produces far smaller files that stay in the same format, so it does not need any readers or decompression software.



NOTES

14.10 ALLOCATION METHODS

An important function of the file system is to manage the space on the secondary storage. It includes keeping track of the number of disk blocks allocated to files and the free blocks available for allocation.

The two main issues related to disk space allocation are:

- Optimum utilization of the available disk space.
- Fast accessing of files.

The widely used methods for allocation of disk space are, contiguous, linked, and indexed. For discussing these different allocation strategies, a file is considered to be a sequence of blocks and all I/O operations on a disk occur in terms of blocks.

14.10.1 Contiguous Allocation

In contiguous allocation, each file is allocated contiguous blocks on the disk, that is, one after the other (see Figure 14.12). Assuming only one job is accessing the disk, once the first block, say b, is accessed, accessing block b+1 requires no head movement normally. Head movement is required only when the head is currently at the last sector of a cylinder and moves to the first sector of the next cylinder; the head movement is only one track. Therefore, number of seeks, and thus, seek time in accessing contiguously allocated files is minimal. This improves the overall file system performance.

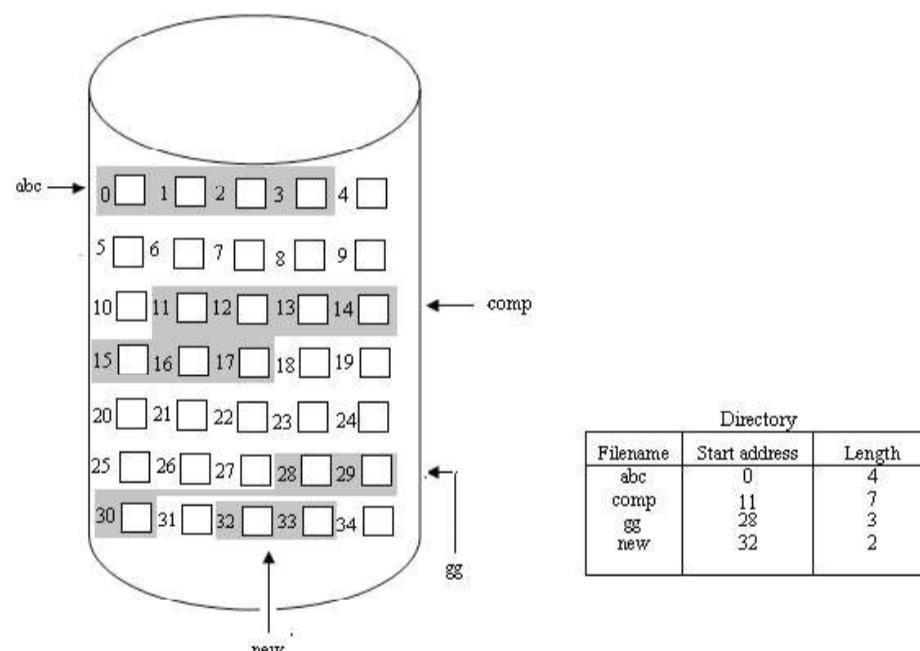


Fig. 14.12 Example of Contiguous Allocation

It is relatively simple to implement the file system using contiguous allocation method. The directory entry for each file contains the file name, the disk address of the first block, and the total size of the file.

Contiguous allocation supports both sequential and direct access to a file. For sequential access, the file system remembers the disk address of the last block referenced, and when required, reads the next block. For direct access to block b of a file that starts at location L , the block $L+b$ can be accessed immediately.

Contiguous allocation has a significant problem of external fragmentation. Initially, the disk is free, and each new file can be allocated contiguous blocks starting from the block where the previous file ends. When a file is deleted, it leaves behind some free blocks in the disk. This is not a problem until we have contiguous blocks to allocate to a new file at the end of disk. However, with time, the disk will become full, and at that time the free blocks are fragmented throughout the disk. One solution to this problem is **compaction**, which involves moving the blocks on the disk to make all free space into one contiguous space. Compaction is quite time consuming as it may take hours to compact a large hard disk that uses contiguous allocation. Moreover, normal operations are not permitted generally during compaction.

An alternative to expensive compaction is to reuse the space. For this, we need to maintain a list of holes (an unallocated segment of contiguous blocks). In addition, we must know the final size of a file at the time of its creation so that a sufficiently large hole can be allocated to it. However, determining the file size in advance is generally difficult and allocating either too little or too more space to a file is a problem. If we allocate more space than what it needs, we end up wasting precious memory. On the other hand, if we allocate too little space than what is needed, we may not extend the file, since the blocks on both sides of the file may be allocated to some other files. One possibility to extend the space is to terminate the user program and then the user must restart it with more space. However, restarting the user program repeatedly may again be expensive. Alternatively, system may find the larger hole, copy the contents of the file to the new space, and then release the previous space. This can be done repeatedly until required space is available contiguously in the disk. Moreover, the user program need not be restarted and the user is also not informed about this. However, the task is again time-consuming.

14.10.2 Linked Allocation

The file size generally tends to change (grow and shrink) over time. The contiguous allocation of such files results in the several problems. Linked allocation method overcomes all the problems of contiguous allocation method.

NOTES

NOTES

In the linked allocation method, each file is stored as a linked list of disk blocks. The disk blocks are generally scattered throughout the disk, and each disk block stores the address of the next block. The directory entry contains the file name and the address of the first and last blocks of the file (see Figure 14.13).

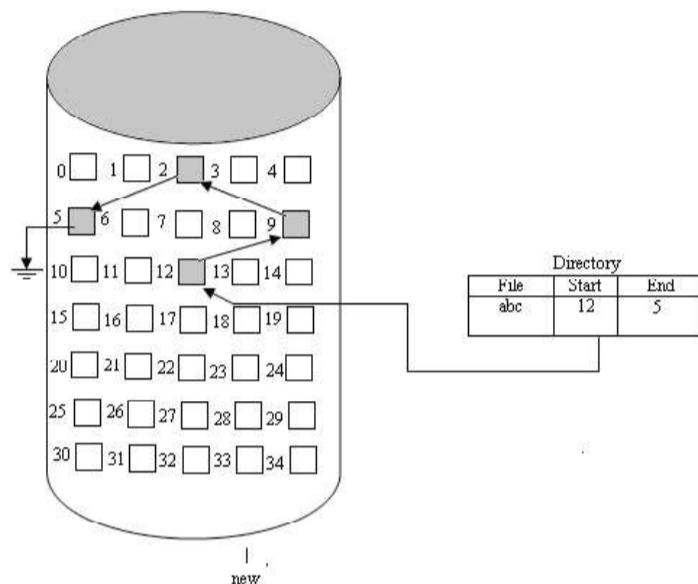


Fig. 14.13 Example of Linked Allocation

Figure 14.13 shows the linked allocation for a file. A total of four disk blocks are allocated to the file. The directory entry indicates that the file starts at block 12. It then continues at block 9, block 2, and finally ends at block 5.

The simplicity and straightforwardness of this method makes it easy to implement. The linked allocation results in optimum utilization of disk space as even a single free block between the used blocks can be linked and allocated to a file. This method does not come across with the problem of external fragmentation, thus, compaction is never required.

The main disadvantages of using linked allocation are the slow access speed, disk space utilization by pointers, and low-reliability of the system. As this method provides only sequential access to files, therefore, to find out the n th block of a file, the search starts at the beginning of the file and follows the pointer until the n th block is found. For a very large file, the average turn around time is high.

In linked allocation, maintaining pointers in each block requires some disk space. The total disk space required by all the pointers in a file becomes substantial, which results in more space required by each file. The space required to store pointers can otherwise be used to store the information. To overcome this problem, contiguous blocks are grouped together as a **cluster**, and allocation to files takes place as clusters rather than blocks. Clusters allocated to a file are then linked together. Having a pointer per cluster rather than per block reduces the total space

NOTES

needed by all the pointers. This approach also improves the disk throughput as fewer disk seeks are required. However, this approach may increase internal fragmentation because having a partially full cluster wastes more space than having a partially full block.

The linked allocation is also not very reliable. Since disk blocks are linked together by pointers, a single damaged pointer may prevent us from accessing the file blocks that follow the damaged link. Some operating systems deal with this problem by creating special files for storing redundant copies of pointers. One copy of file is placed in main memory to provide faster access to disk blocks. Other redundant pointers files help in safer recovery.

14.10.3 Indexed Allocation

There is one thing common to both linked and indexed allocation, that is, non-contiguous allocation of disk blocks to the files. However, they follow different approaches to access the information on the disk. Linked allocation supports sequential access, whereas, indexed allocation supports sequential as well as direct access.

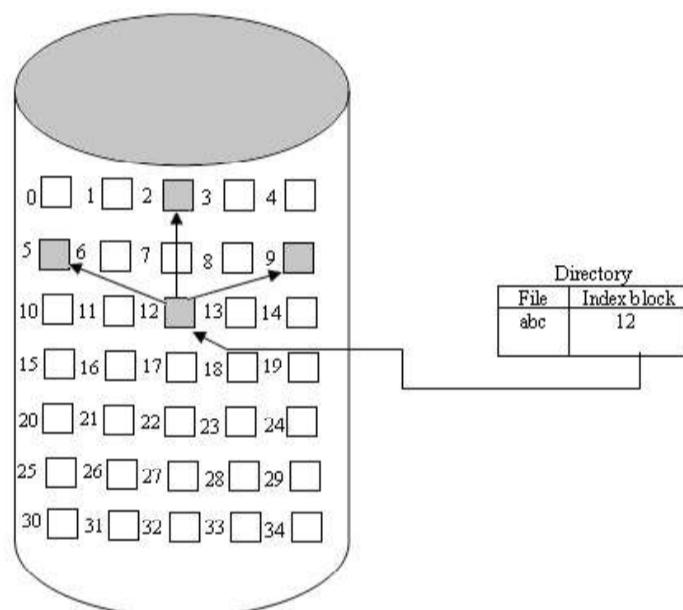


Fig. 14.14 Example of Indexed Allocation

In indexed allocation, the blocks of a file are scattered all over the disk in the same manner as they are in linked allocation. However, here the pointers to the blocks are brought together at one location known as the **index block**. Each file has an index block (see Figure 14.14), which is an array of disk-block pointers (addresses). The k th entry in the index block points to the k th disk block of the file. To read the k th disk block of a file, the pointer in the k th index block entry is

NOTES

used to find and read the desired block. The index block serves the same purpose as a page map table does in the paged memory systems.

The main advantage of indexed allocation is the absence of external fragmentation, since, any free blocks on the disk may be allocated to fulfill a demand for more space. Moreover, the index can be used to access the blocks in a random manner.

When compared to linked allocation, the pointer overhead in indexed allocation is comparatively more. This is because with linked allocation, a file of only two blocks uses a total of 8 bytes for storing pointers (assuming each pointer require 4 bytes of space). However, with indexed allocation, the system must allocate one block (512 bytes) of disk space for storing pointers. This results in wastage of 504 bytes of the index block as only 8 bytes are used for storing the two pointers.

Clearly, deciding the size of index block is a major issue because too large block may result in wastage of memory and too small index block limits the size of largest file in the system. If 4 bytes are used to store a pointer to a block, then a block of size 512 bytes can store up to 128 pointers, thus, the largest file in that system can have 65536 bytes (512×128) of information. However, we may have a file which exceeds the size limit of 65536 bytes. To solve this problem, **multi-level indexes**, with two, three, or four levels of indexes may be used. The two level indexes, with 128×128 addressing is capable of supporting file sizes up to 8 MB and the three level indexes with $128 \times 128 \times 128$ addressing can support file size of up to 1 GB.

The performance of the file system can be greatly enhanced by placing the frequently accessed index blocks in cache memory. This reduces the number of disk accesses required to retrieve the address of the target block.

14.11 FREE SPACE MANAGEMENT

Whenever a new file is created, it is allocated some space from the available free space on the disk. The free space can be either the space on the disk that is never used for allocation or the space left by the deleted files. The file system maintains a **free-space list** that indicates the free blocks on the disk. To create a file, the free-space list is searched for the required amount of space, and the space is then allocated to the new file. The newly allocated space is removed from the free-space list. Similarly, when a file is deleted, its space is added to the free-space list. The various methods used to implement free-space list are bit vector, linked list, grouping and counting.

NOTES**14.11.1 Bit Vector**

Bit vector also known as **bit map** is widely used to keep track of the free blocks on a disk. To track all the free and used blocks on a disk with total n blocks, a bit map having n bits is required. Each bit in a bit map represents a disk block where, a 0 in a bit represents an allocated block and a 1 in a bit represent a free block. Figure 14.15 shows bit map representation of a disk.

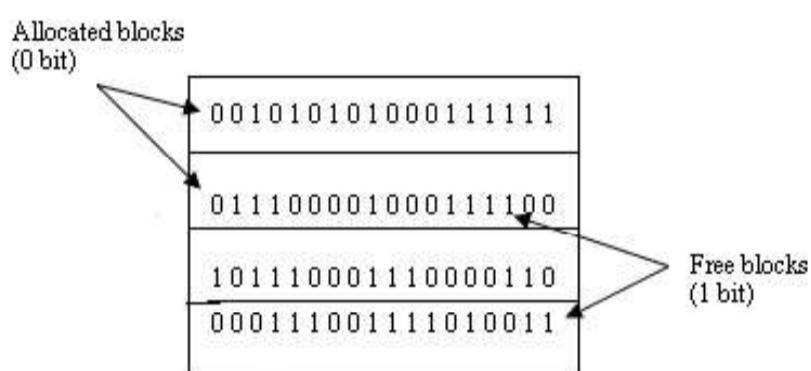


Fig. 14.15 Bit Map Representation

The bit map method for free-space list implementation is very simple. For instance, if a file requires four free blocks using contiguous allocation method, free blocks 12, 13, 14, and 15 (the first four free blocks on the disk that are adjacent to each other) may be allocated. However, for the same file using linked or indexed allocation, the file system may use free blocks 2, 4, 6, and 8 for allocation to the file.

The bit map is usually kept in main memory to optimize the search for free blocks. However, for systems with larger disks, keeping the complete bit map in main memory becomes difficult. For a 2 GB disk with 512-byte blocks, a bit map of 512 KB would be needed.

14.11.2 Linked List

The linked list method for free-space management creates a linked list of all the free blocks on the disk. A pointer to the first free block is kept in a special location on the disk and is cached in the memory. This first block contains a pointer to next free block, which contains a pointer to next free block, and so on. Figure 14.16 shows the linked list implementation of free blocks, where block 2 is the first free block on the disk, which points to block 4, which points to block 5, which points to block 8, which points to block 9, and so on.

NOTES

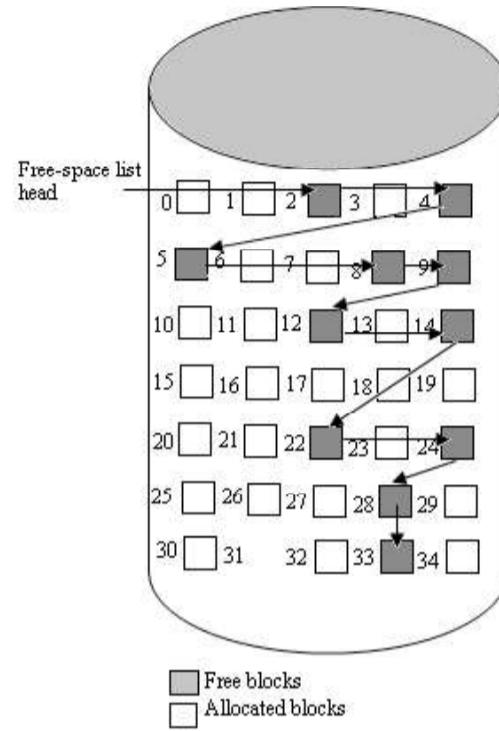


Fig. 14.16 Free Space Management through Linked List

Linked list implementation for managing free-space list requires additional space. This is because a single entry in linked list requires more disk space to store a pointer as compared to 1 bit in bit map method. In addition, traversing the free-list requires substantial I/O operations as we have to read each and every block, which takes a lot of time.

14.11.3 Grouping

Grouping is a modification to the free-list approach in the sense that instead of having a pointer in each free block to the next free block, we have pointers for first n free blocks in the first free block. The first $n-1$ blocks are then actually free. The n th block contains the address of next n free blocks, and so on. A major advantage of this approach is that the addresses of many free disk blocks can be found with only one disk access.

14.11.4 Counting

When contiguous or clustering approach is used, creation or deletion of a file allocates or de-allocates multiple contiguous blocks. Therefore, instead of having addresses of all the free blocks, as in grouping, we can have a pointer to the first free block and a count of contiguous free blocks that follow the first free block. With this approach, the size of each entry in the free-space list increases because

an entry now consists of a disk address and a count, rather than just a disk address. However, the overall list will be shorter, as count is usually greater than 1.

14.12 EFFICIENCY AND PERFORMANCE

The allocation methods and directory-management techniques discussed so far greatly affects the disk performance and efficiency.

14.12.1 Efficiency

The optimum utilization of disk space to store the data in an organized manner defines the efficiency of a file system. A careful selection of the disk-allocation and directory-management algorithms is most important to improve the efficiency of a disk.

To improve the disk efficiency, the UNIX file system allocates in advance)the i-nodes on a disk partition. The i-nodes are spread throughout the partition. The allocation and free-space algorithms then attempt to keep file's data blocks near it's i-node block. This reduces the seek time to access file's data and thus improves the disk as well as system's performance.

UNIX also makes use of clustering to improve its file system performance. The size of the clusters depends on the file size. For large files, large clusters are used, and for small files, small clusters are used. This reduces the internal fragmentation that otherwise occurs when normal clustering takes place.

The amount and nature of information kept in the file's directory (i-node) influences the efficiency of the file system. A file's directory that stores detailed information about a file is informative but at the same time it requires more disk read/write for keeping the information up to date. Therefore, while designing the file system, due consideration must be given to the data that should be kept in the directory.

Other consideration that must be kept in mind while designing the file system is determining the size of the pointers (to access data from files). Most systems use either 16-bit or 32-bit pointers. These pointer sizes limit the file sizes to either 2^{16} (64 KB) or 2^{32} bytes (4 GB). A system that requires larger files to store data can implement a 64-bit pointer. This pointer size supports file of 2^{64} bytes. However, the greater the size of the pointer, the more is the disk space required to store it. This in turn makes allocation and free space management algorithms (linked list, indexes, and so on) to use more disk space.

For better efficiency and performance of a system, the various factors, such as pointer size, length of directory entry, table size, and so on, need to be considered while designing the operating system.

NOTES

NOTES

14.12.2 Performance

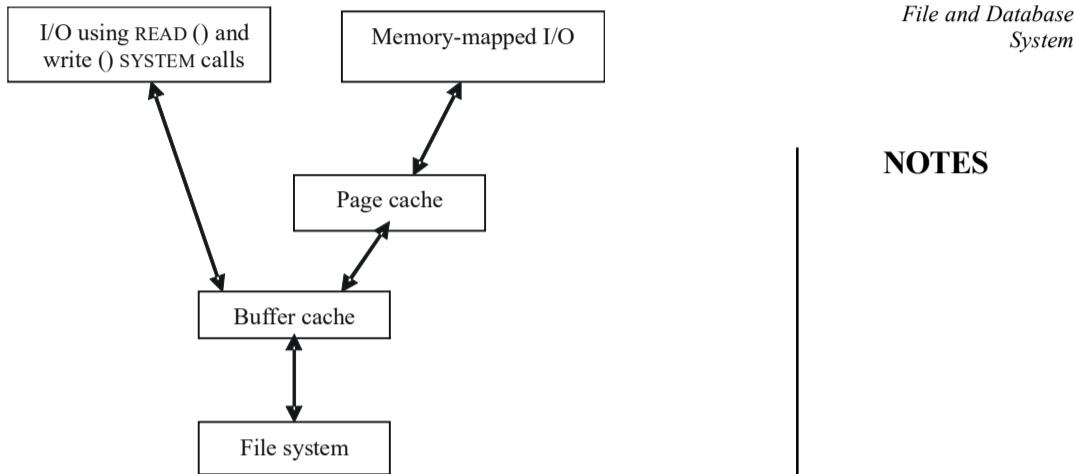
The systems read and write operations with memory are much faster as compared to the read and write operations with the disk. To reduce this time difference between disk and memory access, various disk optimization techniques, such as caching, free-behind and read-ahead are used.

To reduce the disk accesses and improve the system performance, blocks of data from secondary storage are selectively brought into main memory (or cache memory) for faster accesses. This is termed as **caching** of disk data.

When a user sends a read request, file system searches the cache to locate the required block. If the block is found then the request will be satisfied with no need for a disk access. However, if the block is not in the cache, it is first brought into the cache, and then copied to the process which requires it. All the successive requests for the same block can then be satisfied from the cache.

When there is a request to read a block from disk, a disk access is required to transfer the block from disk to the main memory. With the assumption that the block may be used again in future, it is kept in a separate section of the main memory. This technique of caching disk blocks in memory is called **block cache**. In some other systems, file data is cached as pages (using virtual-memory techniques) rather than as file system oriented blocks. This technique is called **page cache**. Caching file data using virtual addresses is more efficient as compared to caching through physical disk blocks. Therefore, some systems use page cache to cache both process pages and file data. This is called as **unified virtual memory**.

Now, consider the two alternatives to access a file from disk: memory mapped I/O and standard system calls, such as read and write. Without a unified buffer cache, the standard system calls has to go through the buffer cache, whereas, the memory mapped I/O has to use the two caches, the page cache and the buffer cache (see Figure 14.17). Memory mapped I/O requires **double caching**. First the disk blocks are read from the file system into the buffer cache, and then the contents in the buffer cache are transferred to the page cache. This is because virtual memory system cannot interface with the buffer cache. Double caching has several disadvantages. First, it wastes memory in storing copy of data in both the caches. Second, each time the data is updated in the page cache, the data in the buffer cache must also be updated to keep the two caches consistent. This extra movement of data within the memory results in the wastage of CPU and I/O time.



NOTES

Fig. 14.17 Input Output without a Unified Buffer Cache
 However, with a unified buffer cache, both memory mapped I/O and the `read` and `write` system calls can use the same page cache (see Figure 14.18). This saves the system resources which are otherwise required for double caching.

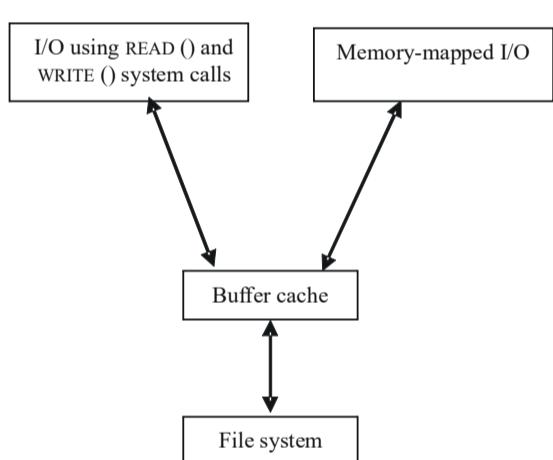


Fig. 14.18 Input Output using a Unified Buffer Cache

The cache memory has a limited space, therefore, once it is full, bringing new pages in cache requires some existing pages to be replaced. A replacement algorithm is applied to remove a page (and rewrite to the disk if it has been modified since being brought in), and then load the new page in the buffer cache. The most widely used replacement algorithms are Least Recently Used (LRU), First-In-First-Out (FIFO) and second chance. Out of these, LRU is the general-purpose algorithm for replacing pages. However, LRU must be avoided with the sequential

NOTES

access files, since the most recently used pages will be used last or may be never again. Instead, sequential access can be optimized by techniques known as free-behind and read-ahead.

The **free-behind** technique removes (free) a page from the buffer as soon as the next page is requested. The pages that are used once are not likely to be used again and they are only wasting buffer space, thus, they are chosen for replacement. Another technique is **read-ahead** in which when a request to read a page arises, the requested page and the several subsequent pages are also read and cached in advance. This is because they are likely to be accessed after the processing of the current page. Bringing the data from the disk in one transfer and caching it saves a considerable amount of time.

14.13 RECOVERY

As discussed earlier, a computer stores data in the form of files and directories on the disk and in the main memory. This data is important for the users who have created it and also for other users who are using it or might use it in future. However, a system failure (or crash) may result in loss of data and in data inconsistency. This section discusses how a system can be recovered to a previous consistent state prior to its failure. The data recovery includes creating full and incremental backups of data to restore the system to a previous working state and checking data consistency using consistency checker.

14.13.1 Backup and Restore

Creating a backup includes recording both data and control data to some other storage devices such as floppy disk, magnetic tape, or optical disc. The frequency of backup depends on the type and nature of the data. If the system is used to store critical data then backups can be created on a daily basis, otherwise, the backup can be created after a fixed interval of time for instance, after every seven or fifteen days.

However, creating full backups daily would lead to significant copying overhead. To avoid recopying complete information, organizations take incremental backups in between the two full backups. An incremental backup includes copying only the changed information (based on the date and time of last backup) to another storage device. Figure 14.19 shows an example of full and incremental backups between two distinct system states. Note that from day n , the cycle of taking incremental backups starts again.

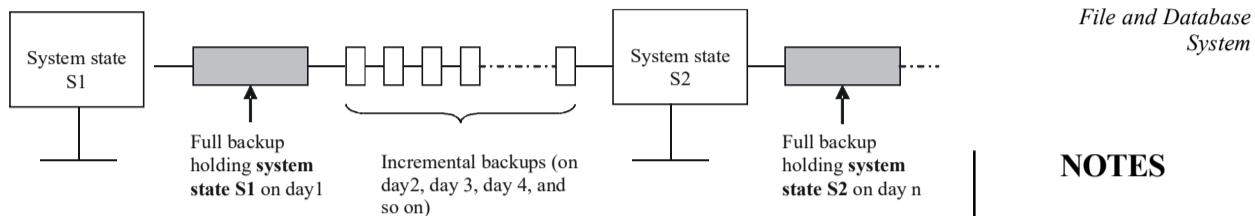


Fig. 14.19 Full and Incremental Backups between two Distinct System States

Consider a situation where a system failure leads to some data loss. To recover this data, the last saved full backup is restored on the system first. Following this, all the incremental backups are restored one by one in the reverse order. However, there may be some processes that are executed in the time between the last incremental backup and the system failure. These processes are re-executed following the recovery to update the system to the last working state (that is the state at the time of system crash).

14.13.2 Consistency Checking

Consider a situation, where due to some reasons, (such as power failure or system crash) the system goes down abruptly and leaves the system in an inconsistent state. The system is said to be in the inconsistent state when there is a difference between the directory information and the actual data on the disk.

The main reason behind the system's inconsistent state is the use of main memory to store directory information. As soon as a file operation occurs, corresponding information in directory is updated in the main memory. However, directory information on disk does not necessarily update at the same time.

To overcome this problem, most system uses a special program called **consistency checker** which runs at the time of system boot. It compares the data in the directory structure with the data blocks on the disk, and tries to fix any inconsistency it finds.

14.14 FILE ACCESS CONTROL

File Access control is a way for Project Administrators to designate the files users can have access to—this is focused on Project permissions. If a user doesn't have access to a specific files folder, the folder and the files it contains are shielded from different views (files tab, file hyperlinks, inside a task) (files tab, file hyperlinks, within a task).

The regulation of file access operates just like a bank. A vault with safety deposit boxes is inside your local bank where you can store your valuables, such

NOTES

as the deed to your house, understanding that without keys to the vault and the key to your safety deposit box, no one can access the deed. In a similar way, critical computer files can be secured by the file access control function of the operating system.

It is always the duty of I.T practitioners to ensure the data on the systems they operate is safe from the possibility of data breaches beyond a reasonable doubt. They guarantee this protection by various means, such as encryption, password use, and network segment setup, among others. Let us talk about a type of access control in this article, which is file system access controls.

Access controls for the file system allow administrators to restrict data access to only those users with approved access. The authority to access files is normally defined using the organization's function matrix.

All modern file systems provide administrators with a way to manage file and folder access based on user and user group identities. These access control systems facilitate the easy implementation and adherence to the function matrix in the organization of the security policies of an organization.

The NTFS (New Technology File System) file system is a Microsoft-designed proprietary logging file system. Starting with Windows NT 3.1, it is the default Windows NT family file system. Used on Windows servers and workstations, it offers a graphical interface that makes it easy for file owners to add and delete users and groups from file access and change access to existing files for users and groups.

In particular, for files stored on the company server, the NTFS file access control is very powerful and can be easily accessed by everyone in the organisation.

14.14.1 Setting Permissions

Operating systems manage file access by configuring file and directory permissions. Permissions can be set such that access to particular files and folders can be allowed or refused. You can access and execute any operation on the file or directory when permission is given. You cannot access the file or directory when permission is refused. Read, write, delete, and execute are the most common permissions.

- **Read:** Enables a user to open and read a directory or file.
- **Write:** It allows a user to open a file or directory, make changes, and save the changes.
- **Delete:** It helps a user to delete a directory or file.
- **Execute:** Allows an executable file to be run by a user. Some files are executable files, typically ending in an .exe or .com file that begins an application on your machine.

NOTES

Check Your Progress

16. A linear search requires O (n) comparisons. (True/False)
17. List the two important issues related to disk space allocation?
18. The widely used methods for disk allocation are _____, _____ and _____.
19. Define the term caching?
20. Which technique is most commonly used to recover lost data?
21. Explain file access control.

**14.15 ANSWERS TO CHECK YOUR PROGRESS
QUESTIONS**

1. File attributes.
2. False.
3. UNIX.
4. Index number.
5. True.
6. Data hierarchy is a basic concept in the theory of data and databases and helps to demonstrate the relationships in a database or data file between smaller and larger components.
7. A data field contains an entity's single fact or attribute.
8. Absolute path name begins at the root and follows a path down to the specified file; whereas, the relative path name defines a path from the current working directory.
9. Unauthorized access, physical damage.
10. Linear list, hash table.
11. An access control list stores the user names and the type of access allowed to each user. When a user tries to access a file, the ACL is searched for that particular file. If that user is listed for the requested access, the access is allowed. Otherwise, the user is denied access to the file.
12. Device driver.
13. False.
14. File control block.

NOTES

15. File optimization is a type of compression that is lossy.
16. True.
17. The two important issues related to disk space allocation are, optimum utilization of the available disk space and fast accessing of files.
18. Contiguous, linked, indexed.
19. When blocks of data from secondary storage are selectively brought into main memory (or cache memory) for faster accesses to reduce the disk accesses and improve the system performance. It is known as caching of disk data.
20. Backup and restore.
21. File Access control is a way for Project Administrators to designate the files users can have access to—this is focused on Project permissions. If a user doesn't have access to a specific files folder, the folder and the files it contains are shielded from different views (files tab, file hyperlinks, inside a task) (files tab, file hyperlinks, within a task).

14.16 SUMMARY

- Computer applications require large amount of data to be stored, so that it can be used as and when required.
- The storage of data on the secondary storage devices makes the data persistent that is, the data is permanently stored and can survive system failures and reboots.
- To store and retrieve files on the disk, the operating system provides a mechanism called the file system.
- The file system is primarily responsible for the management and organization of various files in a system.
- A file is a collection of related data stored as a named unit on the secondary storage.
- The additional information known as file attributes are associated with each file. This information helps the file system to manage a file within the system.
- The common file attributes are name, size, type, identifier, location, date and time, and protection.
- File operations are the functions that can be performed on a file. Operating system handles the file operations through the use of system calls.

NOTES

- The various operations that can be performed on a file are create, write, read, seek, delete, open, append, rename and close.
- The most common technique to implement a file type is by providing extension to a file. The file name is divided into two parts, with the two parts separated by a period ('.') symbol, where the first part is the name and the second part after the period is the file extension.
- Another way to implement the file type is the use of Magic Number. A magic number is a sequence of bits, placed at the starting of a file to indicate roughly the type of file.
- The file structure refers to the internal structure of the file, that is, how a file is internally stored in the system.
- The most common file structures recognized and enforced by different operating systems are byte sequence, record sequence and tree structure.
- Regardless of the file structure used, all disk I/O take place in terms of blocks (physical records).
- The information stored in the file can be accessed in one of the two ways: sequential access, or direct access.
- When the information in the file is accessed in order, one record after the other, it is called sequential access.
- When a file is viewed as a sequence of blocks (or records) which are numbered and can be read or written in any order using this number, it is called direct access.
- Data Hierarchy refers to, often in a hierarchical form, the formal arrangement of data. Lands, databases, files and so on are involved in data organisation.
- A data field contains an entity's single fact or attribute.
- The various schemes to define the structure of a directory are, single-level directory, two-level directory and hierarchical directory.
- The information stored in a system requires to be protected from the physical damage and unauthorized access.
- To protect the files from improper accesses, the access-control mechanism can make use of passwords, and access-control list.
- Every operating system imposes a file system that helps to organize, manage and retrieve data on the disk.
- The design of file system involves two key issues. The first issue involves defining a file and its attributes, operations that can be performed on a file, and the directory structure for organizing files. The second issue involves creating algorithms and data structures to map the logical file system onto the physical secondary storage devices.

NOTES

- The file system is made up of different layers, where each layer represents a level.
- The various file system components are the I/O controller, basic file system, file-organization module and logical file system.
- The file control block stores information about a file such as ownership, permissions, and location of the file content.
- Several on-disk and in-memory structures are used to implement a file system.
- The key issue related to directory implementation is the management of directories to locate files in a file system.
- The most commonly used directory-management algorithms are linear list and hash table.
- The linear list method organizes a directory as a collection of fixed size entries, where each entry contains a (fixed-length) file name, a fixed structure to store the file attributes, and pointers to the data blocks.
- The hash table is a complex data structure with 0 to n-1 table entries, where n is the total number of entries. It uses a hash function to compute a hash value (a number between 0 to n-1) based on a file attribute.
- The hash table is used along with the linear list of directory to reduce the search time.
- Optimization of files is extremely efficient at reducing Image-heavy document size, such as PDF, Word & PowerPoint files that are not well reduced with the use of other technologies for compression.
- The two main issue related to disk allocation are, optimum utilization of the available disk space and the fast access to files.
- The different allocation methods used for allocating disk space to files are, contiguous, linked and indexed.
- In contiguous allocation, each file is allocated contiguous blocks on the disk, that is, one after the other.
- In the linked allocation method, each file is stored as a linked list of disk blocks. The disk blocks are generally scattered throughout the disk, and each disk block stores the address of the next block.
- In indexed allocation, the blocks of a file are scattered all over the disk in the same manner as they are in linked allocation. However, here the pointers to the blocks are brought together at one location known as the index block.

NOTES

- The free-space list indicates the free blocks on a disk.
- The different methods that are used to implement free-space list are bit vector, linked list, grouping and counting.
- A careful selection of the disk-allocation and directory-management algorithms is most important to improve the efficiency of a disk.
- Data recovery includes creating full and incremental backups of data to restore the system to a previous working state and checking data consistency using consistency checker.
- File Access control is a way for Project Administrators to designate the files users can have access to—this is focused on Project permissions.
- If a user doesn't have access to a specific files folder, the folder and the files it contains are shielded from different views (files tab, file hyperlinks, inside a task) (files tab, file hyperlinks, within a task).

14.17 KEY WORDS

- **File system:** Primarily responsible for the management and organization of various files in a system
- **File operations:** Functions that can be performed on a file
- **Sequential access:** A type of access mode when the information in a file is accessed in order, one record after the other
- **Direct access:** When a file is viewed as a sequence of blocks (or records) which are numbered and can be read or written in any order using this number
- **File control block:** It stores information about a file such as ownership, permissions, and location of the file content
- **Hash table:** A complex data structure with 0 to $n-1$ table entries, where n is the total number of entries. A hash table is used along with the linear list of directory to reduce search time
- **Free-space list:** A list that indicates the free blocks on a disk
- **Data recovery:** The process of creating full and incremental backups of data to restore a system to a previous working state and checking data consistency using consistency checker
- **File access control:** File Access control is a way for Project Administrators to designate the files users can have access to—this is focused on Project permissions.

14.18 SELF ASSESSMENT QUESTIONS AND EXERCISES

NOTES

Short-Answer Questions

1. What is the need for storing data on secondary storage devices?
2. State the role of a file system in organizing and managing different files in a system.
3. ‘The operating system gives a logical view of the data to its user.’ Justify this statement.
4. When a user double clicks on a file listed in Windows Explorer, a program is run taking that file as parameter. List two different ways the operating system could know which program to run.
5. Name the components of data hierarchy.
6. Define the following terms:
 - (a) Path name
 - (b) Magic number
7. How file optimization works?
8. List the advantages of using linked list and index allocation over the linear list allocation method.
9. Give an example to show that pointer overhead is higher in indexed allocation than in linked allocation.
10. What is the difference between caching with and without the unified buffer cache?

Long-Answer Questions

1. Some systems simply associate a stream of bytes as a structure for a file’s data, while others associate many types of structures for a file’s data. What are the relative advantages and disadvantages of each system?
2. Explain, with examples, why some systems face more internal fragmentation as compared to other.
3. A program has just read the seventh record; it next wants to read the fifteenth record. How many records must the program read to input the fifteenth record with?
 - (a) Direct access?
 - (b) Sequential access?

NOTES

4. Give an example of an application in which data in a file is accessed in the following order:
 - (a) Sequentially
 - (b) Randomly
5. Explain the relative merits and demerits of using hierarchical directory structure over single-level and two-level directory structures?
6. Explain the role of each layer in a file system with the help of example.
7. Explain why both the linked and indexed allocation schemes use non-contiguous allocation of disk blocks to files but follow different approaches to access the information on a disk.
8. Explain the need for having a standard file system structure attached to various devices in a system.
9. Explain the various on-disk and in-memory structures that are used for implementing a file system.
10. Compare the advantages of the various schemes used for free space management.
11. Discuss the various methods used in UNIX for improving the efficiency and performance of a system.
12. Explain how data on a system can be recovered to a previous working state after a system failure without any data inconsistency.

14.19 FURTHER READINGS

- Deitel, Harvey M., Paul J. Deitel and David R. Choffnes. 2007. *Operating Systems*, 3rd Edition. London (UK): Pearson Education.
- Deitel, Harvey M. 1984. *An Introduction to Operating Systems*. Boston (US): Addison-Wesley.
- Chandra, Pramod and P. Bhatt. 2007. *An Introduction to Operating Systems: Concepts and Practice*. New Delhi: PHI Learning Pvt. Ltd.
- Silberschatz, Abraham, Peter B. Galvin and Greg Gagne. 2008. *Operating System Concepts*, 8th Edition. New Jersey: John Wiley & Sons.
- Tanenbaum, Andrew S. 2006. *Operating Systems Design and Implementation*, 3rd Edition. New Jersey: Prentice Hall.

NOTES

Tanenbaum, Andrew S. 2001. *Modern Operating Systems*. New Jersey: Prentice Hall.

Stallings, William. 1995. *Operating Systems*, 2nd Edition. New Jersey: Prentice Hall.

Milenkovic, Milan. 1992. *Operating Systems: Concepts and Design*. New York: McGraw Hill Higher Education.

Mano, M. Morris. 1993. *Computer System Architecture*. New Jersey: Prentice Hall Inc.