Maharaja Education Trust (R), Mysuru

Maharaja Institute of Technology Mysore

Belawadi, Sriranga Pattana Taluk, Mandya – 571 477

Approved by AICTE, New Delhi,
Affiliated to VTU, Belagavi & Recognized by Government of Karnataka

Lecture Notes on

Operating Systems

(17CS64)

Prepared by

Department of Computer Science and Engineering

Maharaja Education Trust (R), Mysuru
Maharaja Institute of Technology Mysore
Belawadi, Sriranga Pattana Taluk, Mandya – 571 477

Vision/ ಆಶಯ

"To be recognized as a premier technical and management institution promoting extensive education fostering research, innovation and entrepreneurial attitude"

ಸಂಶೋಧನೆ, ಆವಿಷ್ಕಾರ ಹಾಗೂ ಉದ್ಯಮಶೀಲತೆಯನ್ನು ಉತ್ತೇಜಿಸುವ ಅಗ್ರಮಾನ್ಯ ತಾಂತ್ರಿಕ ಮತ್ತು ಆಡಳಿತ ವಿಜ್ಞಾನ ಶಿಕ್ಷಣ ಕೇಂದ್ರವಾಗಿ ಗುರುತಿಸಿಕೊಳ್ಳುವುದು.

Mission/ ಧ್ಯೇಯ

- To empower students with indispensable knowledge through dedicated teaching and collaborative learning.

  ಸಮರ್ಪಣಾ ಮನೋಭಾವದ ಬೋಧನೆ ಹಾಗೂ ಸಹಭಾಗಿತ್ವದ ಕಲಿಕಾಕ್ರಮಗಳಿಂದ ವಿದ್ಯಾರ್ಥಿಗಳನ್ನು ಅತ್ಯತ್ಕೃಷ್ಟ ಜ್ಞಾನಸಂಪನ್ನರಾಗಿಸುವುದು.

- To advance extensive research in science, engineering and management disciplines.

  ವೈಜ್ಞಾನಿಕ, ತಾಂತ್ರಿಕ ಹಾಗೂ ಆಡಳಿತ ವಿಜ್ಞಾನ ವಿಭಾಗಗಳಲ್ಲಿ ವಿಸ್ತೃತ ಸಂಶೋಧನೆಗಳೊಡನೆ ಬೆಳವಣಿಗೆ ಹೊಂದುವುದು.

- To facilitate entrepreneurial skills through effective institute - industry collaboration and interaction with alumni.

  ಉದ್ಯಮ ಕ್ಷೇತಗಳೊಡನೆ ಸಹಯೋಗ, ಸಂಸ್ಥೆಯ ಹಿರಿಯ ವಿದ್ಯಾರ್ಥಿಗಳೊಂದಿಗೆ ನಿರಂತರ ಸಂವಹನಗಳಿಂದ ವಿದ್ಯಾರ್ಥಿಗಳಿಗೆ ಉದ್ಯಮಶೀಲತೆಯ ಕೌಶಲ್ಯ ಪಡೆಯಲು ನೆರವಾಗುವುದು.

- To instill the need to uphold ethics in every aspect.

  ಜೀವನದಲ್ಲಿ ನೈತಿಕ ಮೌಲ್ಯಗಳನ್ನು ಅಳವಡಿಸಿಕೊಳ್ಳುವುದರ ಮಹತ್ವದ ಕುರಿತು ಅರಿವು ಮೂಡಿಸುವುದು.

- To mould holistic individuals capable of contributing to the advancement of the society.

  ಸಮಾಜದ ಬೆಳವಣಿಗೆಗೆ ಗಣನೀಯ ಕೊಡುಗೆ ನೀಡಬಲ್ಲ ಪರಿಪೂರ್ಣ ವ್ಯಕ್ತಿತ್ವವುಳ್ಳ ಸಮರ್ಥ ನಾಗರೀಕರನ್ನು ರೂಪಿಸುವುದು.

VISION/ ಆಶಯ

"To be a leading academic department offering computer science and engineering education, fulfilling industrial and societal needs effectively."

"ಕೈಗಾರಿಕಾ ಮತ್ತು ಸಾಮಾಜಿಕ ಅಗತ್ಯಗಳನ್ನು ಪರಿಣಾಮಕಾರಿಯಾಗಿ ಪೂರೈಸುವ ಮೂಲಕ ಕಂಪ್ಯೂಟರ್ ವಿಜ್ಞಾನ ಮತ್ತು ಎಂಜಿನಿಯರಿಂಗ್ ಶಿಕ್ಷಣವನ್ನು ನೀಡುವ ಪ್ರಮುಖ ಶೈಕ್ಷಣಿಕ ವಿಭಾಗವಾಗುವುದು."

MISSION/ ಧ್ಯೇಯ

- To enrich the technical knowledge of students in diversified areas of Computer science and engineering by adopting outcome based approaches.

  ಫಲಿತಾಂಶ ಆಧಾರಿತ ವಿಧಾನಗಳನ್ನು ಅಳವಡಿಸಿಕೊಳ್ಳುವ ಮೂಲಕ ಕಂಪ್ಯೂಟರ್ ವಿಜ್ಞಾನ ಮತ್ತು ಎಂಜಿನಿಯರಿಂಗ್‌ನ ವೈವಿಧ್ಯಮಯ ಕ್ಷೇತ್ರಗಳಲ್ಲಿನ ವಿದ್ಯಾರ್ಥಿಗಳ ತಾಂತ್ರಿಕ ಜ್ಞಾನವನ್ನು ಅಭಿವೃದ್ಧಿ ಪಡಿಸುವುದು.

- To empower students to be competent professionals maintaining ethicality.

  ನೈತಿಕತೆಯನ್ನು ಕಾಪಾಡುವ ಸಮರ್ಥ ವೃತ್ತಿಪರರಾಗಿ ವಿದ್ಯಾರ್ಥಿಗಳನ್ನು ಸಶಕ್ತಗೊಳಿಸುವುದು.

- To facilitate the development of academia-industry collaboration.

  ಶೈಕ್ಷಣಿಕ-ಉದ್ಯಮ ಸಹಯೋಗದ ಅಭಿವೃದ್ಧಿಗೆ ಅನುಕೂಲವಾಗುವಂತೆ.

- To create awareness of entrepreneurship opportunities.

  ಉದ್ಯಮಶೀಲತೆ ಅವಕಾಶಗಳ ಬಗ್ಗೆ ಜಾಗೃತಿ ಮೂಡಿಸುವುದು.

Program Outcomes

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

VISION/ ಆಶಯ

"To be a leading academic department offering computer science and engineering education, fulfilling industrial and societal needs effectively."

"ಕೈಗಾರಿಕಾ ಮತ್ತು ಸಾಮಾಜಿಕ ಅಗತ್ಯಗಳನ್ನು ಪರಿಣಾಮಕಾರಿಯಾಗಿ ಪೂರೈಸುವ ಮೂಲಕ ಕಂಪ್ಯೂಟರ್ ವಿಜ್ಞಾನ ಮತ್ತು ಎಂಜಿನಿಯರಿಂಗ್ ಶಿಕ್ಷಣವನ್ನು ನೀಡುವ ಪ್ರಮುಖ ಶೈಕ್ಷಣಿಕ ವಿಭಾಗವಾಗುವುದು."

MISSION/ ಧ್ಯೇಯ

- To enrich the technical knowledge of students in diversified areas of Computer science and engineering by adopting outcome based approaches.

  ಫಲಿತಾಂಶ ಆಧಾರಿತ ವಿಧಾನಗಳನ್ನು ಅಳವಡಿಸಿಕೊಳ್ಳುವ ಮೂಲಕ ಕಂಪ್ಯೂಟರ್ ವಿಜ್ಞಾನ ಮತ್ತು ಎಂಜಿನಿಯರಿಂಗ್‌ನ ವೈವಿಧ್ಯಮಯ ಕ್ಷೇತ್ರಗಳಲ್ಲಿನ ವಿದ್ಯಾರ್ಥಿಗಳ ತಾಂತ್ರಿಕ ಜ್ಞಾನವನ್ನು ಅಭಿವೃದ್ಧಿ ಪಡಿಸುವುದು.

- To empower students to be competent professionals maintaining ethicality.

  ನೈತಿಕತೆಯನ್ನು ಕಾಪಾಡುವ ಸಮರ್ಥ ವೃತ್ತಿಪರರಾಗಿ ವಿದ್ಯಾರ್ಥಿಗಳನ್ನು ಸಶಕ್ತಗೊಳಿಸುವುದು.

- To facilitate the development of academia-industry collaboration.

  ಶೈಕ್ಷಣಿಕ-ಉದ್ಯಮ ಸಹಯೋಗದ ಅಭಿವೃದ್ಧಿಗೆ ಅನುಕೂಲವಾಗುವಂತೆ.

- To create awareness of entrepreneurship opportunities.

  ಉದ್ಯಮಶೀಲತೆ ಅವಕಾಶಗಳ ಬಗ್ಗೆ ಜಾಗೃತಿ ಮೂಡಿಸುವುದು.

Maharaja Institute of Technology Mysore

Department of Computer Science and Engineering

Overview

Subject: Operating System **Subject Code: 17CS64**

An operating system(OS) is system software that manages computer hardware,software resources, and provides common services for computer programs. Timesharingoperating systemsschedule 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 operatingsystem 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 with a market share of around 82.74%, mac OS by Apple in second place with 13.23%, and the varieties of Linux are collectively in third place with 1.57%.while other operating systems amount to just 0.3%, 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.

Course objectives

- Introduce concepts and terminology used in OS.
- Explain threading and multithreaded systems.
- Illustrate process synchronization and concept of Deadlock.
- Introduce Memory and Virtual memory management, File system and storage techniques.
- A case study on Linux operating system.

Course Outcomes

CO's	DESCRIPTION OF THE OUTCOMES
C364.1	Acquire the basic knowledge of Operating System Architecture, Operations and Services offered.
C364.2	Identify and estimate process management, thread management and deadlock management strategies.
C364.3	Illustrate Memory, File system and Disk Management techniques.
C364.4	Outline the design principles of Linux operating systems through case study.
C364.5	Demonstrate the importance of life-long learning of operating system usage.

OPERATING SYSTEMS
[As per Choice Based Credit System (CBCS) scheme]
(Effective from the academic year 2017 - 2018)
SEMESTER – VI

Subject Code	17CS64	IA Marks	40
Number of Lecture Hours/Week	4	Exam Marks	60
Total Number of Lecture Hours	50	Exam Hours	03

CREDITS – 04

Module – 1	**Teaching Hours**
Introduction to operating systems, System structures: What operating systems do; Computer System organization; Computer System architecture; Operating System structure; Operating System operations; Process management; Memory management; Storage management; Protection and Security; Distributed system; Special-purpose systems; Computing environments. Operating System Services; User - Operating System interface; System calls; Types of system calls; System programs; Operating system design and implementation; Operating System structure; Virtual machines; Operating System generation; System boot. **Process Management** Process concept; Process scheduling; Operations on processes; Inter process communication	**10 Hours**
Module – 2	
Multi-threaded Programming: Overview; Multithreading models; Thread Libraries; Threading issues. Process Scheduling: Basic concepts; Scheduling Criteria; Scheduling Algorithms; Multiple-processor scheduling; Thread scheduling. **Process Synchronization:** Synchronization: The critical section problem; Peterson's solution; Synchronization hardware; Semaphores; Classical problems of synchronization; Monitors.	**10 Hours**
Module – 3	
Deadlocks : Deadlocks; System model; Deadlock characterization; Methods for handling deadlocks; Deadlock prevention; Deadlock avoidance; Deadlock detection and recovery from deadlock. **Memory Management:** Memory management strategies: Background; Swapping; Contiguous memory allocation; Paging; Structure of page table; Segmentation.	**10 Hours**
Module – 4	
Virtual Memory Management: Background; Demand paging; Copy-on-write; Page replacement; Allocation of frames; Thrashing. **File System, Implementation of File System:** File system: File concept; Access methods; Directory structure; File system mounting; File sharing; Protection: Implementing File system: File system structure; File system implementation; Directory implementation; Allocation methods; Free space management.	**10 Hours**
Module – 5	
Secondary Storage Structures, Protection: Mass storage structures; Disk structure; Disk attachment; Disk scheduling; Disk management; Swap space management. Protection: Goals of protection, Principles of protection, Domain of protection, Access matrix, Implementation of access matrix, Access control, Revocation of access rights, Capability- Based systems. **Case Study: The Linux Operating System:** Linux history; Design principles; Kernel modules; Process	**10 Hours**

management; Scheduling; Memory Management; File systems, Input and output; Inter-process communication.	

Course outcomes: The students should be able to:

- Demonstrate need for OS and different types of OS
- Discuss suitable techniques for management of different resources
- Illustrate processor, memory, storage and file system commands
- Explain the different concepts of OS in platform of usage through case studies

Question paper pattern:

The question paper will have TEN questions.
There will be TWO questions from each module.
Each question will have questions covering all the topics under a module.
The students will have to answer FIVE full questions, selecting ONE full question from each module.

Text Books:

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating System Principles 7th edition, Wiley-India, 2006.

Reference Books

1. Ann McHoes Ida M Fylnn, Understanding Operating System, Cengage Learning, 6th Edition
2. D.M Dhamdhere, Operating Systems: A Concept Based Approach 3rd Ed, McGraw-Hill, 2013.
3. P.C.P. Bhatt, An Introduction to Operating Systems: Concepts and Practice 4th Edition, PHI(EEE), 2014.
4. William Stallings Operating Systems: Internals and Design Principles, 6th Edition, Pearson.

Maharaja Institute of Technology Mysore

Department of Computer Science and Engineering

INDEX

Subject: Operating System **Subject Code: 17CS64**

SL. No.	Contents	Page No.
Module-1: Introduction to OS		
1.1	What Operating Systems do?	M1-1
1.2	Computer System Organization	M1-2
1.3	Computer System Architecture	M1-4
1.4	Operating System Structure	M1-6
1.5	Operating System Operations	M1-7
1.6	Process Management	M1-8
1.7	Memory Management	M1-9
1.8	Storage Management	M1-9
1.9	Protection and Security	M1-11
1.10	Distributed Systems	M1-12
1.11	Special Purpose Systems	M1-12
1.12	Computing Environments	M1-13
1.13	Operating System Services	M1-15
1.14	User Operating-System Interface	M1-16
1.15	System Calls	M1-17
1.16	Types of System calls	M1-19
1.17	System Programs	M1-22
1.18	OS Design & Implementation	M1-23
1.19	Operating System Structures	M1-24
1.20	Virtual Machines	M1-27
1.21	System Boot	M1-30
1.22	Process Concepts	M1-30
1.23	Process Scheduling	M1-32
1.24	Process Operations	M1-35
1.25	Inter-Process Communication (IPC)	M1-37
Module-2: Multithreaded Programming		
2.1	Overview	M2-1
2.2	Multithreading Models	M2-2
2.3	Thread Libraries	M2-3
2.4	Threading Issues	M2-8

2.5	PROCESS SHEDULING: Basic Concepts	M2-11
2.6	Scheduling Criteria	M2-14
2.7	Scheduling Algorithms	M2-14
2.8	Multiple-Processor Scheduling	M2-20
2.9	Thread Scheduling	M2-22
2.10	Synchronization: Background	M2-23
2.11	The critical-section problem	M2-23
2.12	Peterson's solution	M2-24
2.13	Synchronization hardware	M2-26
2.14	Semaphores	M2-28
2.15	Classic problems of Synchronization	M2-31
2.16	Monitors	M2-35
Module-3: Deadlocks, Memory Management		
3.1	Deadlocks	M3-1
3.2	System Model	M3-1
3.3	Deadlock Characterization	M3-1
3.4	Methods for Handling Deadlocks	M3-4
3.5	Deadlock Prevention	M3-4
3.6	Deadlock Avoidance	M3-5
3.7	Deadlock Detection	M3-11
3.8	Recovery from Deadlock	M3-13
3.9	MEMORY MANAGEMENT STRATEGIES: Background	M3-17
3.10	Swapping	M3-22
3.11	Contiguous Memory Allocation	M3-22
3.12	Paging	M3-24
3.13	Structure of the Page Table	M3-30
3.14	Segmentation	M3-33
Module-4: Virtual Memory Management, File System		
4.1	Background	M4-1
4.2	Demand Paging	M4-2
4.3	Copy-on-write	M4-6
4.4	Page Replacement	M4-7
4.5	Allocation of Frames	M4-14
4.6	Thrashing	M4-16
4.7	File System	M4-19
4.8	File Concept	M4-19
4.9	Access Methods	M4-22
4.10	Directory Structure	M4-24
4.11	File System Mounting	M4-29
4.12	File Sharing	M4-30
4.13	Protection	M4-33

4.14	File System Structure	M4-35
4.15	File System Implementation	M4-36
4.16	Directory Implementation	M4-39
4.17	Allocation Methods	M4-40
4.18	Free Space Management	M4-44
Module-5: Secondary Storage Structures, Protection		
5.1	Overview of Mass Storage Structure	M5-1
5.2	Disk Structure	M5-2
5.3	Disk Attachment	M5-2
5.4	Disk Scheduling	M5-4
5.5	Disk Management	M5-8
5.6	Swap - Space Management	M5-10
5.7	System Protection	M5-11
5.8	Goals of Protection	M5-11
5.9	Principles of Protection	M5-12
5.10	Domain of Protection	M5-12
5.11	Access Matrix	M5-15
5.12	Implementing Access matrix	M5-18
5.13	Access Control	M5-19
5.14	Revocation of Access Rights	M5-20
5.15	Capability Based Systems	M5-21
5.16	Linux History	M5-22
5.17	Design Principles	M5-24
5.18	Kernel Modules	M5-26
5.19	Process management	M5-29
5.20	Scheduling	M5-32
5.21	Memory Management	M5-35
5.22	File System	M5-43
5.23	Input and Output	M5-48
5.24	Inter-Process Communication	M5-50

MODULE-1

INTRODUCTION TO OPERATING SYSTEMS, SYSTEM STRUCTURE

An Operating System (OS) is a system software that manages the computer hardware.

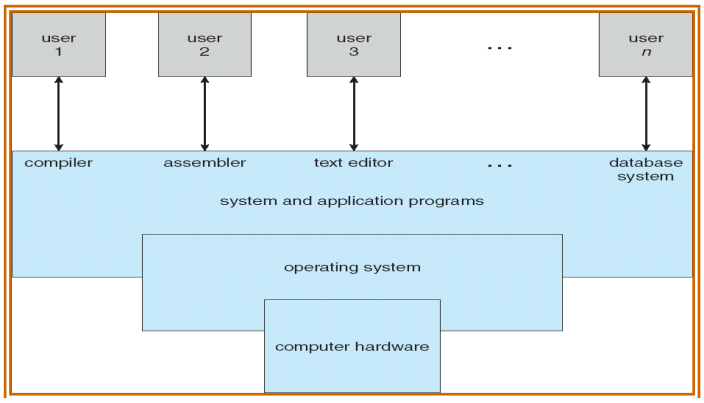
- It provides a basis for application programs and acts as an intermediary between the computer users and the computer hardware.
- The purpose of an OS is to provide an environment in which the user can execute the program in a convenient & efficient manner.

1.1 What Operating Systems do?

A computer system can be divided into **four components**

- ✓ **Hardware**: The Hardware consists of memory, CPU, ALU, I/O devices, peripherals devices & storage devices.
- ✓ **OS:** The OS controls & co-ordinates the use of hardware among various application programs for various users.
- ✓ **Application Program:** The application programs includes word processors, spread sheets, compilers & web browsers which defines the ways in which the resources are used to solve the problems of the users.
- ✓ **User:** Who works/executes the required function.

The following **figure**shows the abstract view of the components of a computer system



To completely understand the role of operating system two views are considered as below:

- **User View:**

- ✓ The user view of the computer depends on the interface used.
- ✓ Some users may use PC's. Such system is designed for one user. Here, the OS is designed for **ease of use** where some attention is mainly on performances and not on the resource utilization.
- ✓ Some users may use a terminal connected to a mainframe or minicomputers.Other users may access the same computer through other terminals. These users may share resources and exchange information. In this case the OS is designed to maximize **resource utilization**- so that all available CPU time, memory & I/O are used efficiently.

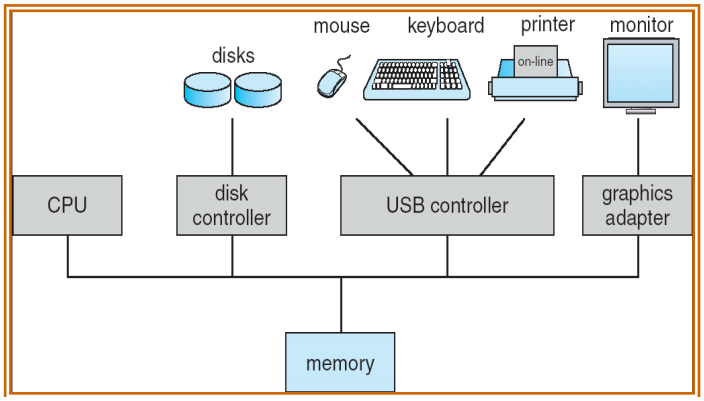
- ✓ Other users may sit at workstations, connected to the networks of other workstation and servers. In this case OS is designed to compromise between individual usability & resource utilization.

- **System View:**

  - ✓ An operating systemcan be viewed as **resource allocator**.
  - ✓ A computer system has many resources such as CPU Time, memory space, file storage space, I/O devices and so on that may be used to solve a problem.
  - ✓ The OS acts as a manager of these resources and decides how to allocate these resources to programs and the users so that it can operate the computer system efficiently and fairly.
  - ✓ A different view of an OS is that it controls various I/O devices & user programs i.e. an OS is a **control program** which manages the execution of user programs to prevent errors and improper use of the computer.

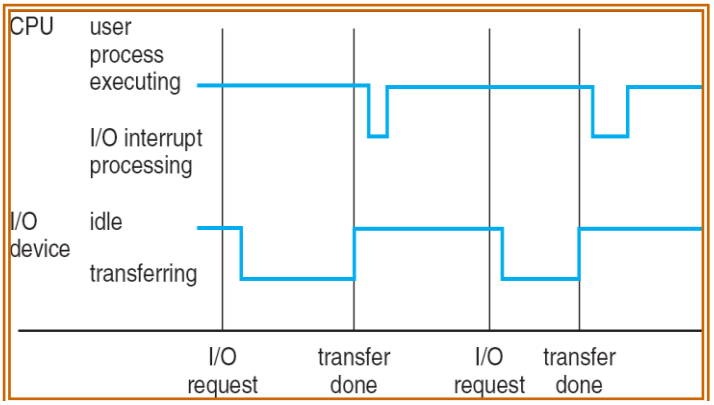
1.2 Computer System Organization

- **Computer system operation**
  - ✓ A general purpose computer system consists of one or more CPUs and device controllers connected through common bus providing access to shared memory as shown in **figure below.**



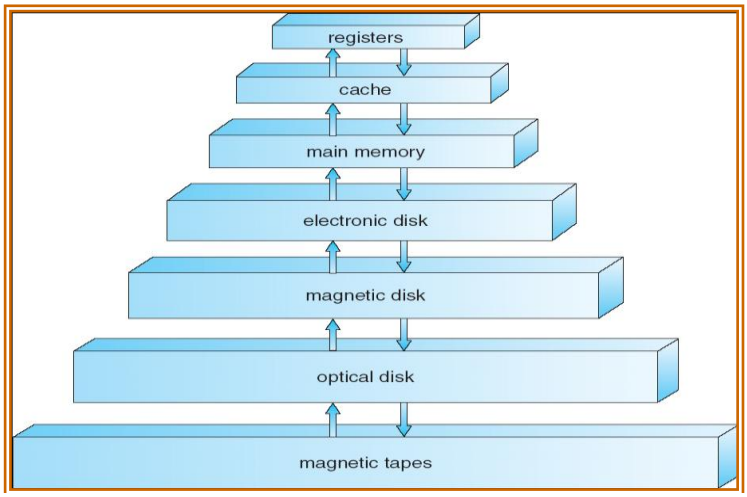
  - ✓ Each **device controller** is in-charge of a particular device type.
  - ✓ CPUs and device controllers can execute concurrently competing for memory utilization and **memory controller** synchronizes the memory access.
  - ✓ For computer to start running when it is powered-up or rebooted an initial program called **bootstrap program** is loaded.It is stored in ROM or EEPROM, generally known as **firmware.**It loads the operating system and starts executing the first process, such as **"init"** and waits for some event to occur.
  - ✓ An operating system is **interrupt** driven.The occurrence of an event is signaled by an interrupt through signal or system call.When CPU is interrupted it stops its job and immediately transfers execution to **fixed location.** Fixed location contains the starting address of interrupt service routine.On completion of the execution of interrupt service routine CPU resumes interrupted computation.
  - ✓ A timeline of this operation is shown below in **figure**.
  - ✓ Each computer has its own interrupt design mechanism but several functions are common. The interrupttransfers the control to the interrupt service routine through the

interrupt vector, which contains the starting addresses of all the interrupt service routines. The interrupted service routine executes and after completion, CPU returns back to previous execution by preserving the state of the CPU by storing registers and the program counter value.



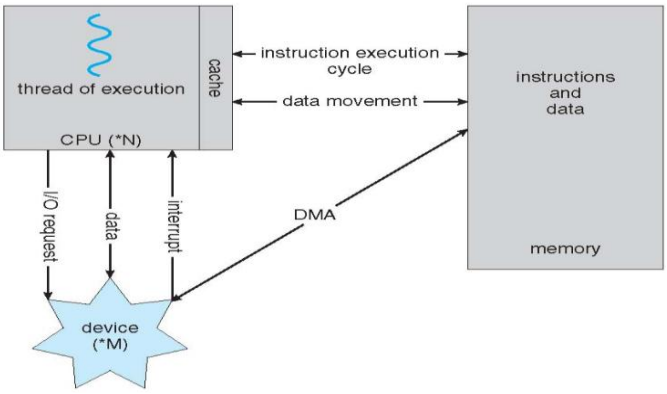
- **Storage Structure**

  - ✓ **Main memory** is the **large storage** media that the **CPU can access** directly. It forms an array of memory words. Each word has its own address. Interaction is achieved through a sequence of **load and store**instructions to specify memory addresses. The load instruction moves a word from main memory to an internal register within the CPU, where as store instruction moves the content of a register to main memory.
  - ✓ It But it is very small and volatile storage device.Computer systems have secondary storage that provides large nonvolatile storage capacity.
  - ✓ Magnetic disks are the common secondary storage devices.Other storage devices are cache memory, CD-ROM, magnetic tapes and so on.
  - ✓ Storage system is differentiated based on their speed, cost and volatility.
  - ✓ Storage systems above electronic disks are volatile,expensive but fast.Below are non-volatile, comparatively cheap and slower.
  - ✓ Electronic disks can be designed as both volatile and non-volatile. Eg. Flash memory used in cameras, robot and Personal Digital Assistants (PDA).
  - ✓ NVRAM (Non Volatile RAM) is a DRAM with a battery backup power.
  - ✓ Storage systems can be organized in a hierarchy as shown below in **figure** according to speed and cost.



- **I/O Structure**

  - ✓ Storage is one form of I/O devices.OS spends considerable amount of time in managing I/O devices because they have an impact on the **performance** and **reliability** of the system.
  - ✓ There is one **device driver** for each **device controller**. Each device controller maintains a buffer and registers. The device driver understands the device controller and presents a uniform interface to the device and to the rest of the system.
  - ✓ When an I/O operation is started, the **device driver loads** the appropriate **registers** within the **device controller**. The **device controller** examines the contents of the **registers** and determines **what action** to be taken.
  - ✓ The controller starts transfer of data from the device to the local buffer.
  - ✓ Once the transfer is complete the device controller informs the device driver via an interrupt that the operation is complete. The device driver then returns control to the OS through an interrupt.
  - ✓ This form of interrupt driven I/O works well for moving small amounts of data but it is a big overhead when bulk transfer is required. To solve this problem, **DMA (direct memory access)** is used.DMA is used for high-speed I/O devices.The device controller transfers entire block of data to or from buffer storage to main memory without **CPU intervention**.Only one interrupt is generated per block, rather than the one interrupt per byte.
  - ✓ The **figure** shows the interplay of all components of a computer system.



1.3 Computer System Architecture

A computer system can be categorized based on number of processors used.

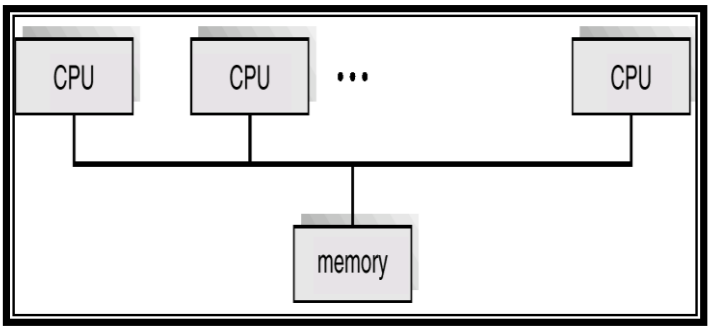
- **Single Processor Systems**

  - ✓ A system that has one main CPU and is capable of executing a general-purpose instruction set, including instructions from the user processes.
  - ✓ Some systems also have special purpose processor to perform specific task, or on mainframes, they come in the form of general purpose processors such as I/O processor.These special purpose processors have limited instruction set and do not run user processes.They are managed by the OS by sending information about the next task and monitor their status.

- ✓ **Ex1:** A **disk controller microprocessor** receives a sequence of requests from the main CPU and implements its own disk queue and scheduling algorithm. This relieves the main CPU from disk scheduling.
- ✓ **Ex2:** PCs contain a **microprocessor in the keyboard** to convert the keystrokes into codes to be sent to the CPU.
- ✓ These special purpose processors do not convert single processor system into multiprocessor system.

- **Multiprocessor Systems**

- ✓ Multiprocessor systems have more than one processor in close communication.Also known as **Tightly Coupled System** or **Parallel Systems**.
- ✓ They share computer bus, the clock, memory & peripheral devices.
- ✓ Two processes can run in parallel.
- ✓ Multi Processor Systems have **3 advantages**,
  - o **Increased Throughput**: By increasing the number of processors we can get more work done in less time. Speed up ratio with N processors is not N, but it is less than N.
  - o **Economy of Scale:**As Multiprocessor systems share peripherals, mass storage & power supplies, they can save more money than **multiple single processor** systems. If many programs operate on same data, they will be stored on one disk and all processors can share them instead of maintaining data on several systems.
  - o **Increased Reliability**: If a program is distributed properly on several processors, then the failure of one processor will not halt the system but it only slows down.
- ✓ The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Such systems that provide graceful degradation are **fault tolerant**.Fault tolerant requires a mechanism to allow failure to be detected, and diagnosed and corrected.
- ✓ Multi processor systems are of **two types**
  - o **Asymmetric Multiprocessing:** Each processor is assigned a specific task. It uses a master slave relationship. A master processor controls the system. The master processors schedules and allocates work to slave processors.
  - o **Symmetric Multiprocessing (SMP):** Each processor performs all tasks within the OS. SMP means all processors are peers i.e. no master slave relationship exists between processors. Each processor concurrently runs a copy of OS. Ex: Solaris. The following **figure** shows SMP architecture.



- ✓ The differences between symmetric & asymmetric multiprocessing may result from either hardware or software. Special hardware can differentiate the multiple processors, or the software can be written to allow only one master & multiple slaves.
- ✓ A recent trend in CPU design is to include multiple compute **cores** on a single chip.

- ✓ **Blade Servers** are recent development in which multiple processor boards, I/O boards, and networking boards are placed in same chassis. Here each processors can boot independently and run their own OS.

- **Clustered Systems**

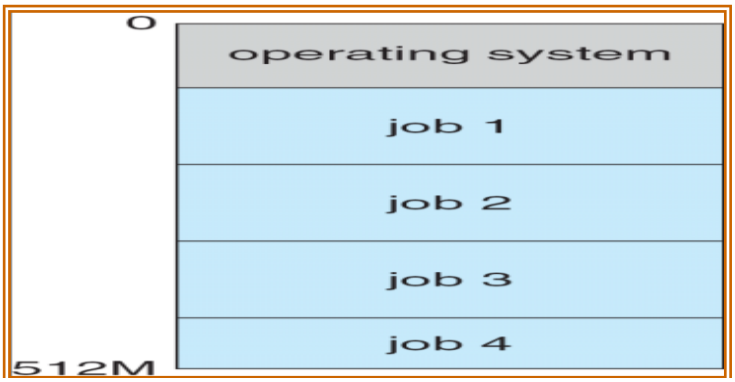
  - ✓ The clustered systems have multiple CPUs but they are composed of two or more individual systems coupled together.
  - ✓ Clustered systems share storage and are closely linked via LAN Network.
  - ✓ Clustering is usually used to provide high availability.
  - ✓ A layer of software cluster runs on the **cluster nodes**. Each node can monitor one or more of the others. If the monitored machine fails, the monitoring machine takes ownership of its storage and restarts the applications that were running on failed machine.
  - ✓ Clustered systems can be categorized into two groups

    1. **Asymmetric Clustering.**
    2. **Symmetric clustering.**

  - ✓ In **asymmetric clustering** one machine is in **hot standby mode** while others are running the application. The hot standby machine does nothing but it monitors the active server. If the server fails the hot standby machine becomes the active server.
  - ✓ In **symmetric mode** two or more hosts are running the application & they monitor each other. This mode is more efficient since it uses all the available hardware.
  - ✓ Other forms of clusters include **parallel clusters** and **clustering over WAN**.
  - ✓ Parallel clusters allow multiple hosts to access the same data on shared storage.
  - ✓ To provide this shared access, system must also supply access control and locking to ensure that no conflicting operations occur. This function known as **distributed lock manager (DLM)** is included.
  - ✓ Clustering provides better reliability than the multiprocessor systems.

1.4 Operating System Structure

- **Multiprogramming system**
  - ✓ Single user cannot keep CPU and I/O devices busy at all times.
  - ✓ Multiprogramming increases CPU utilization by organizing jobs so that CPU always has one to execute.
  - ✓ The OS has to keep several jobs in memory simultaneously as shown in **figure**
  - ✓ The OS picks up and starts executing one of the jobs.
  - ✓ Eventually if this job may not need the CPU due to some reason like, an I/O operation to complete, then in non multi-programmed system CPU would sit idle.
  - ✓ But in a multi-programmed system instead of having the CPU idle the OS switches to the next job in the memory.



- **Timesharing (multitasking)**

  - ✓ It is a logical extension of multiprogramming in which CPU switches among jobs so frequently that users can interact with each job while it is running, creating interactive computing.
  - ✓ Allows many users to share the computer simultaneously.
  - ✓ It requires an **interactive system or a hands-on** computer system that provides direct communication between the user and the system.
  - ✓ Response time should be less than 1 second.
  - ✓ Each user has at least one program executing in memory.
  - ✓ A program loaded into memory and executing is called a **process.**
  - ✓ Timesharing and multiprogramming requires several jobs to be kept simultaneously in memory.
- **Job scheduling**: A job pool consists of all processes residing on disk and awaiting allocation of main memory. If several jobs are ready to be brought into memory and if there is not enough room for them, then the system must choose among them. Making this decision is Job scheduling.
- **CPU scheduling**: If several jobs are ready to run at the same time then the system must choose among them. Making this decision is CPU scheduling.
- **Swapping**: In time shared system processes are swapped in and out of main memory into the disk to ensure reasonable response time.A common method for achieving this is **Virtual memory.**The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual physical memory.

1.5 Operating System Operations

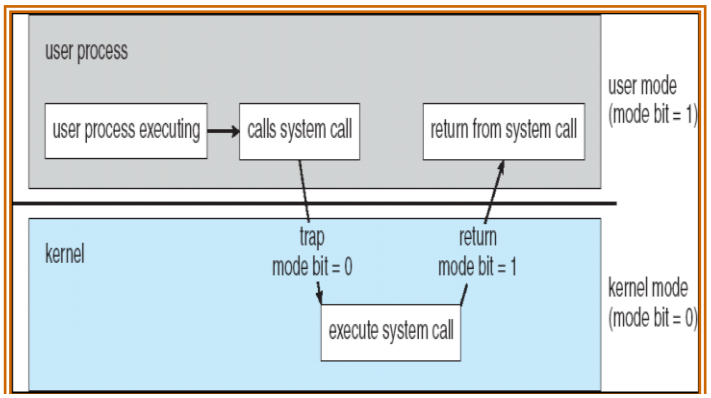
- ✓ Modern OS is **Interrupt driven**.
- ✓ Events are signaled by an **interrupt or trap**.
- ✓ An **exception or trap** is a software generated interrupt either by an error.

(For ex: Division by zero, invalid memory access)or specific request from a user program.

- **Dual-mode operation**

  - ✓ To ensure proper execution of the OS, we must be able to distinguish between **OS code** and **user defined code.**
  - ✓ Most computer systems provide hardware support to differentiate among various modes of execution.

- ✓ **Two modes** of operation are
    1. **User mode**
    2. **kernel mode** (also called supervisor, system or privileged mode)
- ✓ **Mode bit** is added to the hardware to indicate current mode User mode(1) and kernel mode(0).
- ✓ When system is executing on behalf of user application, the system is in user mode.
- ✓ When a user application requests a service from OS, it must transit from user to kernel mode to fulfill the request as shown in **figure**
- ✓ At **system boot time**, the hardware starts in kernel mode. The OS is then loaded and starts user applications in user mode. Whenever a **trap or interrupt** occurs, the hardware switches from user mode to kernel mode. Thus whenever the OS gains control of the computer, it is in **kernel mode**. The system always switches to user mode before passing control to user program. This allows protection to OS.



✓ The hardware allows **privileged instructions** to be executed only in kernel mode. If an attempt is made to execute privileged instructions in user mode, the hardware does not execute it but rather treats it as an illegal and traps it to the OS.

✓ **Examples**:Instruction to switch to user mode, I/O control instructions, timer management instructions and interrupt management instructions.

✓ **User program** asks OS to perform OS tasks through system call.

- **Timer**

- ✓ Timer is used to prevent a program from getting stuck in an **infinite loop**or not calling system services and never returning control to the OS.
- ✓ Timer can be set to **interrupt** the computer after a specific period.
- ✓ The period may be **fixed or variable**.The **variable timer** is implemented by fixed rate clock and a counter.Whenever the clock ticks, operating system decrements the counter.When counter reaches zero it generates an interrupt.
- ✓ Timer has to be set before scheduling process to regain control or terminate program that exceeds allotted time.

1.6 Process Management

- ✓ A process is a **program in execution**. **Ex1**. A time-shared user program like a compiler is a process. **Ex2**. A word processing program run by an individual user on a PC is a process.

- ✓ A process requires certain **resources** like CPU time, Memory, I/O devices to complete its task.
- ✓ When the process terminates, the OS reclaims all the reusable resources.
- ✓ A program in a file is stored on disk and is a **passive entity**, where as a process is an **active entity**located on main memory.
- ✓ A single threaded process has one **PC (program counter)** specifying the address of the next instruction to be executed. Such processes are sequential i.e. the CPU executes one instruction after the other.
- ✓ A multi-threaded process has multiple **program counters** each pointing to the next instruction to execute for a given thread.
- ✓ A system consists of a collection of processes, some of which are **OS processes** and the rest are **user processes**. All these processes can execute concurrently by multiplexing the CPU among them on a single CPU.
- ✓ The OS is responsible for the following activities of the process management,
  - o Creating & deleting of the user & system processes.
  - o Suspending and resuming processes.
  - o Providing mechanisms for process synchronization.
  - o Providing mechanisms for process communication.
  - o Providing mechanisms for deadlock handling.

1.7 Memory Management

- ✓ Main memory is the **central** to the operation of the computer system.
- ✓ Main memory is the large array of words or bytes, ranging in size from hundreds of thousands to billions. Each word or byte will have their **own address**.

- ✓ The CPU reads the instruction from main memory during **instruction fetch cycle**& during the **data-fetch cycle**it reads & writes the data.
- ✓ The main memory is the only storage device in which a CPU is able to address & access directly.
- ✓ For a program to be executed, it must be loaded into memory & mapped to absolute addresses. When the program terminates, all available memory will be returned back.
- ✓ To improve the utilization of CPU & the response time several programs will be kept in memory.
- ✓ Several memory management schemes are available & selection depends on the **Hardware design** of the system.
- ✓ The OS is responsible for the following activities
  - o Keeping track of which parts of the memory are used & by whom.
  - o Deciding which process and data to move into and out of memory.
  - o Allocating & reallocating memory space as needed.

1.8 Storage Management

- **File System Management**

- ✓ File management is one of the most visible components of an OS.
- ✓ Computer can store information on different types of **physical media** like Magnetic Disks, Magnetic tapes, optical disks etc.
- ✓ These devices have their own **unique characteristics** like access speed, capacity, data transfer rate, and access method (sequential or random).

- ✓ OS implements the abstract concept of a file by managing mass storage media like tapes, disks etc.,
- ✓ A **file** is a collection of related information defined by its **creator**. They commonly represent programs (source and object) and data. Data files may be numeric, alphabetic or alphanumeric.
- ✓ Files can be organized into **directories** to make them easier to use.
- ✓ The OS is responsible for the following activities,
    - Creating & deleting files.
    - Creating & deleting directories.
    - Supporting primitives for manipulating files & directories.
    - Mapping files onto secondary storage.
    - Backing up files on stable(non volatile) storage media.

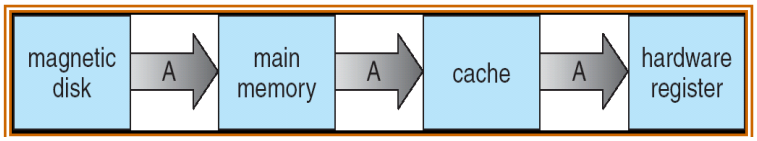
- **Mass Storage management**

- ✓ Computer system must provide **secondary storage** toback up main memory because,
    - It is too small to accommodate all data and programs.
    - Data held in this memory is lost when power goes off.
- ✓ Most programs including compilers, assemblers, word processors, editors etc are stored on the disk until loaded into the memory and then use disk as both source and destination of processing. Hence proper management of disk storage is very important.

- ✓ The OS is responsible for the following activities,
    - Free space management.
    - Storage allocation.
    - Disk scheduling.
- ✓ Slower and low cost but high capacity backup storage devices are called **tertiary devices** which are used for back-up of the regular disk data, seldom used data, long term archival storage etc. **Eg**. Magnetic tapes and their drives, CD and DVD drives and platters like tape and optical platters.

- **Caching**

- ✓ It is an important principle of a computer system and is a **fast memory** which is used for storing information on a temporary basis.
- ✓ First, the cache is searched when a particular piece of information is required during processing. If the information already available, then it is directly used from the cache, otherwise we use information from the source, putting a copy in the cache, under the assumption that we will need the information again very soon.
- ✓ Internal **programmable registers** like index registers can be used as high-speed cache for the main memory.
- ✓ Caches have limited size and thus **Cache management** is an important design problem.
- ✓ In a hierarchical storage structure, the same data may appear in different levels of storage system.**For ex**, Suppose that an integer A is to be incremented by 1 is located in file B which resides on disk, the migration of integer A from Disk to Register is shown in below



- ✓ Once the increment to A takes place in the internal registers, the value of A differs in various storage systems. The value of A becomes same only after the new value of A is written from the internal register back to the **disk**.
- ✓ In **multitasking environments**, extreme care must be taken to use most recent value, not matter where it is stored in the storage hierarchy.
- ✓ The situation becomes more complicated in **multiprocessor environment**, where each CPU is associated with local cache. A care must be taken to make sure that an update to the value of A in one cache is immediately reflected in all other caches. This situation is called as **cache coherency.**
- ✓ The situation becomes even more complex in a **distributed environment**. Several copies of the same file can be kept on different computers. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place all other replicas are also updated as soon as possible.

- **I/O Systems**

- ✓ OS hides peculiarities of hardware devices from the user.
- ✓ I/O subsystem consists of several components like,
  - The memory management component that includes buffering, caching and spooling.
  - A general device driver interface.
  - Drivers for specific hardware devices.
- ✓ Only the device driver knows the peculiarities of each of the devices.

1.9 Protection and Security

- ✓ If a computer system has multiple users and allows the concurrent execution of multiple processes, then a protection mechanism is required to regulate access to data.
- ✓ System resources like files, memory segments, CPU etc. are made available to only those processes which have gained **authorization** from OS.
- ✓ **Protection** is a mechanism for controlling the access of processes or users to the resources defined by a computer system.
- ✓ The mechanism must specify the controls to be imposed and what for.
- ✓ The **advantages** of providing protection are: it can improve reliability by detecting latent errors at the interfaces between component subsystems and early detection of interface errors can prevent corruption of good subsystems by another malfunctioning subsystem. Protection can prevent misuse by an unauthorized or incompetent user.
- ✓ The **security system** must defend the system from external and internal attacks. **Attacks** can be of various types like viruses, worms, denial-of-service attack, identity theft, theft of service etc. **Ex**. If a user's authentication information is stolen then the owner's data can be stolen, corrupted or deleted.
- ✓ The mechanism of protection and security must be able to distinguish among all its users.This is possible because the system maintains a list of all **user ids**. These ids are unique per user.
- ✓ When it is required to distinguish among a set of users rather than individual users then group functionality is implemented.

- ✓ A system-wide list of **group names and group ids** are stored.
- ✓ If a user needs to **escalate privileges** for gaining extra permissions then different methods are provided by the OS.

1.10 Distributed Systems

- ✓ A distributed system is a collection of physically separate **heterogeneous computer systems** that are networked to provide the users with access to various resources that the system maintains.
- ✓ A distributed system is one in which Hardware or Software components located at the networked computerscommunicate & coordinate their actions only by passing messages.
- ✓ Distributed systems depend on **networking** for their functionality.Network may vary by the protocols used, distance between nodes (LAN, WAN, MAN, etc) & transport media.
- ✓ A **network operating system** is an OS that provides features such as file sharing across the network and allows different processes on different computers to exchange messages.
- ✓ **The advantages** of Distributed Systems are,
  - Resource sharing
  - Higher reliability
  - Better price performance ratio
  - Shorter response time
  - Higher throughput
  - Incremental growth

1.11 Special Purpose Systems

The special purpose computers are those whose functions are more limited and whose objectives are to deal with limited computation domains. **Egs**.Realtime Embedded Systems, Multimedia Systems and Handheld Systems.

- **Real- Time Embedded Systems**

- ✓ Embedded computers are found almost everywhere from car engines, robots, alarm systems, medical imaging systems, industrial control systems, microwave ovens, weapon systems etc.
- ✓ This class of computers have very **specific task** and run an OS with very **limited features**.Usually they have limited or **no user interface**.

- ✓ Embedded systems runs on **real time OS**.
- ✓ A real time system should have well defined, **fixed time** constraints.Processing must be done within the defined constraints or the system will fail.Hence they are often

- ✓ used as **controlled device** in a dedicated application.Real time OS uses priority scheduling algorithm to meet the response requirement of a real time application.
- ✓ Real time systems are of **two types**
  - Hard Real Time Systems
  - Soft Real Time Systems

- ✓ **A hard real time system** guarantees that the critical tasks to be completed on time. This goal requires that all delays in the system be bounded from the retrieval of stored data to time that it takes the OS to finish the request.
- ✓ In **soft real time system** is a less restrictive one where a critical real time task gets priority over other tasks & retains the property until it completes. Soft real time system is achievable goal that can be mixed with other type of systems. They have limited utility than hard real time systems.Soft real time systems are used in area of multimedia, virtual reality & advanced scientific projects. It cannot be used in robotics or industrial controls due to lack of deadline support.Soft real time requires

  two conditions to implement, CPU scheduling must be priority based & dispatch latency should be small.

- **Multimedia Systems**

  - ✓ A recent trend in technology is the incorporation of **multimedia data**.
  - ✓ Multimedia data consists of audio and video files along with conventional files(text files, word document, etc).
  - ✓ The difference from conventional data is that the multimedia data must be delivered or streamed according to some **time** restrictions.
  - ✓ Multimedia applications include video conferencing, news stories download over the internet, live webcasts of speeches and so on.

- **Handheld Systems**

  - ✓ Handheld systems include Personal Digital Assistants (PDAs), Cellular telephones, palm and pocket PCs and so on, which uses special purpose embedded OS.
  - ✓ **Drawbacks**are, because of smaller size they have small amount of memory, slow processors and small display screen.
  - ✓ **Memory**-Because of **small size**, OS and applications must manage the memory efficiently. (Making sure all memory allocated is returned back to the memory manager if no longer used).Many handheld devices do not use virtual memory techniques.
  - ✓ **Speed-**Processors run at a fraction of the speed of the PC processor. Faster processors require more power. Therefore, OS and applications must not tax the processor.
  - ✓ The **small display screen** also limits the output options. To allow the display of the contents of the web pages **web clipping** is done where only a small subset of the web page is delivered and displayed on the screen.
  - ✓ **Advantages are**
    - Ability to synchronize with desktops.
    - Small size hence can be carried around easily.

1.12 Computing Environments

- **Traditional Computing**

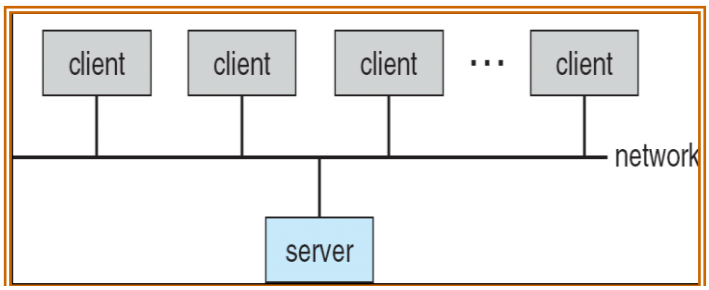
  - ✓ Consider the **"typical office environment"**: Few year's back it consisted of PCs connected to the network with servers providing file and print service, Remote access looked tough and portability was achieved through laptop.

- ✓ Terminals attached to mainframes were common at many companies with even few remote access and portability option.
- ✓ The web technologies are stretching the boundaries of traditional computing. Companies have **portals,** which provide web access to their internal servers.Network computers are terminals that understand the web based computing. Hand held PDAs can also connect to wireless networks to use company's web portal.

- **Client-Server Computing**

  - ✓ Many of today's systems act as **server systems** to satisfy requests of clients.
  - ✓ This form of specialized distributed system, called **client-server system** has general structure as shown in below**figure**

  - ✓ Server system can be **classified** as follows
    a. **Compute-Server Systems**: Provides an interface to which client can send requests to perform some actions, in response the server execute the action and send back result to the client. **Ex**. A server running a database that responds to client requests for data.
  b. **File-Server Systems**: Provides a file system interface where clients can create, update, read & delete files. **Ex**. A web server that delivers files to clients running web browsers.



- **Peer-to-Peer(P2P) Computing**

  - ✓ It is another form of a distributed system.Here, clients and servers are not distinguished from one another.

  - ✓ All nodes within the system are considered as **peers.**Each can act as a server or a client depending on who is requesting or providing a service.
  - ✓ The **advantage** is the removal of bottleneck as the services can be provided by several nodes that are distributed throughout the network.
  - ✓ To participate in a P2P system a node must first join the network of peers. On joining, the new node can provide and request for services.
  - ✓ **Determining** what services are available in the network can be accomplished in one of **two** methods,

    1. When a node joins a network, it registers its services with a centralized lookup service on the network. Any node wants service, first contacts the centralized lookup service to determine which nodes provides the service. Then the communication takes place between the client and the service provider.

2. A peer which is a client broadcasts a request for service to all nodes in the network. The nodes that provide the service responds to the requesting peer. A discovery protocol is used by the peers to discover the services provided by other peers.

- **Web Based Computing**

  - ✓ It leads to more access by wider variety of devices other than PCs workstations, PDAs, and cell phones.
  - ✓ Web computing has increased the emphasis on networking.
  - ✓ Devices that were not previously networked have been wired or wireless nowadays.
  - ✓ The network connectivity is faster through improved network technology and optimized network implementation code.
  - ✓ Web based computing has given rise to a new category of devices called **load balancers** which distribute network connections among a pool of similar servers.

1.13 Operating System Services

An OS provides an environment for the execution of the programs. The common services provided by the OS are

1. **User interface:** Almost all operating systems have a user interface (UI).This interface can take several forms.
   a. **Command-line interface(CLI):** uses text commands and a specific method for entering them.
   b. **Batch Interface:** commands and directives to control are entered into files and those files are executed.
   c. **Graphical User Interface (GUI):** most common. Interface is a window system with a pointing device directing the I/O, choose from menus, make selections along with keyboard to enter text.
2. **Program Execution:** The OS must able to load the program into memory & run that program. The program must be able to end its execution either normally or abnormally.
3. **I/O Operation:** A running program may require I/O(file or an I/O device). Users cannot control the I/O devices directly. So the OS must provide a means for controlling I/O devices.
4. **File System manipulation**: Program needs to read and write files and directories. They also need to create and delete files, search for a given file and list file information. Some programs include **permission management** to deny access to files or directories based on file ownership.
5. **Communication**: In certain situation one process may need to exchange information with another process. This communication may takes place in two ways.
   i. Between the processes executing on the same computer.
   ii. Between the processes executing on different computer that are connected by a network.

   Communications can be implemented via **shared memory** or by **message passing**, in which packets of information are moved between processes by the OS.
6. **Error Detection**: Errors may occur in CPU, I/O devices or in Memory Hardware. The OS constantly needs to be aware of possible errors. For each type of errors the OS should take appropriate actions to ensure correct & consistent computing.

7. **Resource Allocation**: When multiple users logs onto the system or when multiple jobs are running, resources must be allocated to each of them. The OS manages different types of OS resources. Some resources may need some special allocation codes & others may have some general request & release code.
8. **Accounting:** We need to keep track of which users use how many & what kind of resources. This record keeping may be used for accounting. This accounting data may be used for statistics or billing. It can also be used to improve system efficiency.
9. **Protection and security**: Protection ensures that all the access to the system are controlled. Security starts with each user having authenticated to the system, usually by means of a password. External I/O devices must also be protected from invalid access. In multi process environment it is possible that one process may interface with the other or with the OS, so protection is required.

1.14 User Operating-System Interface

There are two fundamental approaches for users to interface with OS,

1. Command Interpreter
2. Graphical User Interface

- **Command Interpreter(CI)**

  - ✓ Some OS include CI in the kernel, and in others like windows-XP and UNIX, it is treated as a special program that is running when a job is initiated or when a user first logs on.
  - ✓ On systems with multiple command interpreters to choose from, the interpreters are known as **shells. For ex:**On UNIX and Linux systems, there are different shells a user may choose from including Bourne shell, C shell and Korn shell etc
  - ✓ The main function of the command interpreter is to get and execute the next user-specified command.
  - ✓ Commands are implemented in two ways:

    1. In one approach, the command interpreter itself has the code to execute the command. Ex. a command to delete a file.
    2. This will result in the command interpreter to go to a section of its code that sets up the parameters and makes the appropriate system call.In this method, the size of the command interpreter depends on the number of commands that can be given.
    3. Alternative approach used by UNIX is most commands are implemented through system programs. The command interpreter uses the command to identify a file to be loaded into memory and executed. Ex. ***rmfile.txt*** would make the command interpreter search for a **file rm**, load that file into memory and execute it with parameter ***file.txt.***

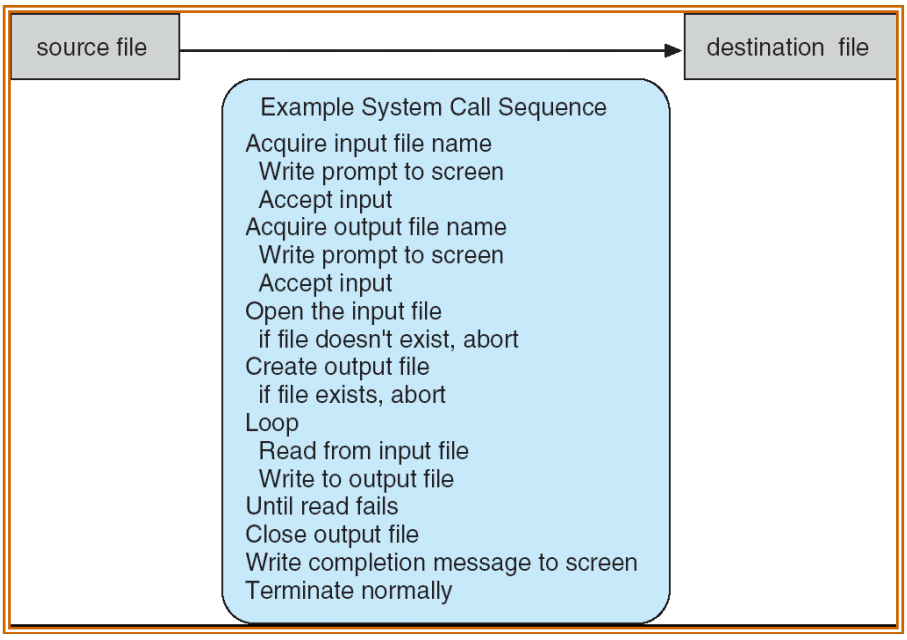
  - ✓ Advantages of CIs are,
    - Command interpreter program is small.
    - Command interpreter does not have to be changed when new commands are added.
    - New commands can be easily added to the system.

- **Graphical User Interface**

  - ✓ GUI provides user-friendly **desktop** metaphor interface where the mouse is moved to position where images or icons on the desktop that represent programs, files directories and other system functions.
  - ✓ Depending on mouse pointer's location, clicking the mouse button can invoke the corresponding program, select a file or directory known as folders or pull down a menu that contains commands.
  - ✓ First appeared in 1970s as a part of research at **Xerox Parc research facility**.
  - ✓ It became widespread with the coming of **Apple Macintosh** in 1980s.
  - ✓ Microsoft's first version of Windows was based on GUI interface for MS-DOS. The various windows systems that have been appeared and had enhancements in the GUI.
  - ✓ UNIX later implemented GUI in CDE (Common Desktop Environment) and X-Windows Systems.Also seen in Solaris and IBM's AIX system.

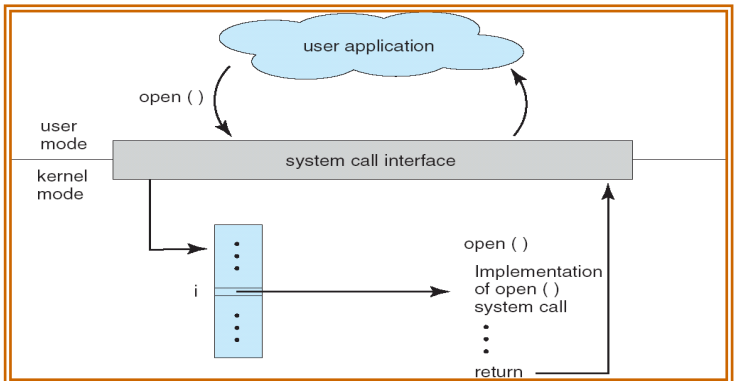
1.15 SYSTEM CALLS

- ✓ System provides interface to the services made available by an OS.
- ✓ These calls are generally available as routines written in C and C++, although certain low-level tasks may need to be written using assembly language instruction.
- ✓ System call sequence to read the contents of one file and copy to another file is illustrated in below**figure**
  - The first input that the program will need is the names of two files which can be specified in many ways. This sequence requires many I/O system calls.
  - Next, the program must open the input file which requires another system call. If opening of file fails, it should display error message on console (another system call) and should terminate abnormally (another system call).
  - Next, the program must create the output file (another system call), If fails, it should display error message on console (another system call) and should also abort (another system call).



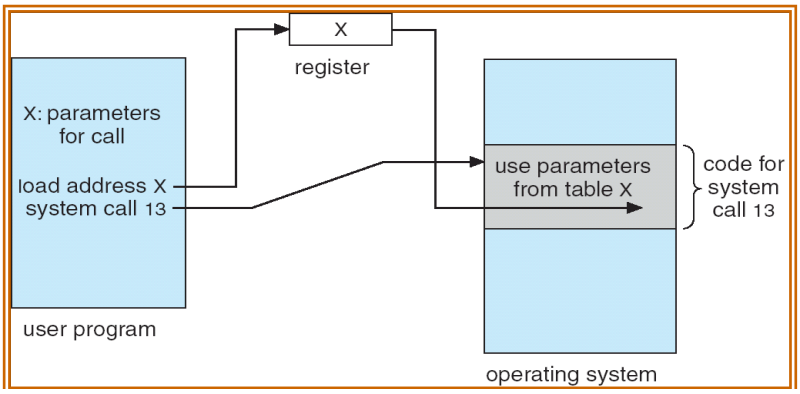
- Next, we enter a loop that reads from input file (system call) and writes to the output file (system call).Write/read operation may fail, which needs another systemcall to continue.
- Finally, after the entire file is copied, the program may close both files (system call), write message to console (system call)), and terminate normally (system call).

- ✓ Application developers design programs according to an **Application Program Interface (API)**. API specifies the set of functions that are available to an application programmerincluding the parameter that are passed to each function and returns values the programmer can expect.
- ✓ Three most common APIs are **Win32** API for Windows, **POSIX** API for POSIX-based systems (UNIX, Linux, and Mac OS X), and **Java** API for the Java virtual machine (JVM).
- ✓ The runtime support system(a set of functions built into libraries included with a compiler) for most programming languages provides a **system call interface** that serves as the link to system calls made available by the OS.
- ✓ The system call interface intercepts function call in the API and invokes the necessary system call within the OS.
- ✓ A**number** is associated with each system call and the **system-call interface** maintains a **table** indexed according to these numbers.
- ✓ The **system call interface** invokes intended system call in OS kernel and returns status of the system call and any return values.
- ✓ The caller needs to know nothing about how the system call is implemented or what it does during execution.
- ✓ The**figure** illustrates how the OS handles a user application which is invoking **open()** system call.



Three general methods are used to pass the parameters to the OS.

1. The simplest approach is to pass the parameters in registers.
2. In some cases there can be more parameters than registers. In these cases the parameters are stored in a **block or table** in memory and the address of the block is passed as a parameter in register. It is shown in below **figure**. This approach is used by Linux and Solaris.
3. Parameters can also be placed or **pushed** onto **stack** by the program &**popped** off the stack by the OS.



Some OS prefer the **block or stack** methods, because those approaches do not limit the number or length of parameters being passed.

1.16 Types of System calls

System calls may be grouped roughly into **5 categories**

1. **Process control**

- **end, abort**

  - ✓ A running program needs to be able to halt its execution either **normally** or **abnormally**. In an abnormal termination a dump of memory is taken and an error message is generated.Dump is written to disk and examined by the debugger to determine the cause of problem.
  - ✓ In normal or abnormal situations the OS must transfer the control to the command interpreter system, which then reads the next command given by the user.
  - ✓ In batch system the command interpreter terminates the execution of job & continues with the next job.Batch-systems uses **control cards** to indicate the special recovery action to be taken in case of errors. It is a command to manage the execution of a process.
  - ✓ More severe errors can be indicated by a higher level error parameter. Normal & abnormal termination can be combined by defining normal termination as an error at **level 0**.The command interpreter uses this error level to determine next action automatically.

- **load, execute**

  - ✓ A process executing one program may want to **load and execute** another program. This feature allows the command interpreter to execute programs as directed by the user.
  - ✓ The question of where to return the control when the loaded program terminates is related to the problem of whether the existing program is lost, saved or allowed to continue execution concurrently with the new program. There is a system call for this purpose (**create or submit process**).

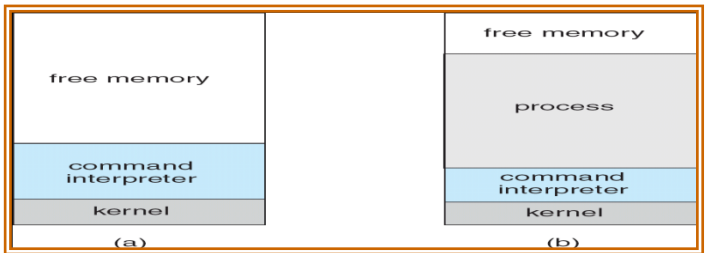
- **create process, terminate process**

  - ✓ If we create a new job or process, itshould be able to control its execution.

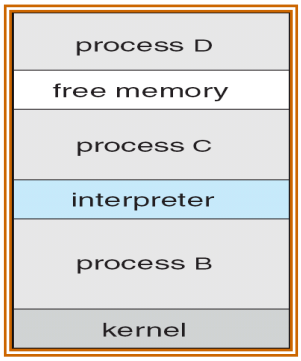
- ✓ This control requires the ability to determine and reset the attributes of a process, including the process priority, its maximum allowable execution time, and so on (get process attributes and set process attributes).
- ✓ We may also want to terminate a process that we created (terminate process)

- **wait event, signal event**
  - ✓ When new jobs have been created, we may want to wait for certain amount of time using **wait time** system call.
  - ✓ When a job has to wait for a certain event to occur **wait event** system call is used.
  - ✓ When the event has occurred the job should signal the occurrence through the **signal event** system call.
- **get process attributes, set process attributes**
- **wait for time**
- **allocate and free memory**

  - **In MS-DOS**
  - ✓ MS-DOS is an example of single tasking system, which has command interpreter system that is invoked when the computer is started as shown in **figure a.**
  - ✓ To run a program MS-DOS uses simple method. It does not create a new process when one process is running.
  - ✓ It loads the program into memory and gives the program as much memory as possible as shown in **figure b.**



  - **In FreeBSD**
  - ✓ Free BSD is an example of multitasking system.
  - ✓ In free BSD the command interpreter may continue running while other program is executed as shown in **figure**



  - ✓ fork() is a system call used to create new process.
  - ✓ Then, the selected program is loaded into memory via an exec() system call, and then program is executed.

2. File management

- create file, delete file
  - ✓ System calls can be used to create & delete files.System calls may require the name of the files and attributes for creating & deleting of files.
- open, close file
  - ✓ Opens the file for usage and finally we need to close the file.
- read, write, reposition
  - ✓ Other operation may involve the reading of the file, write & reposition the file after it is opened.

- get and set file attributes
  - ✓ For directories, some set of operation are to be performed. Sometimes it is required to reset some of the attributes on files & directories. The system call **get file attribute**&**set file attribute** are used for this type of operation.

3. Device management

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

  - ✓ The system calls are also used for accessing devices.
  - ✓ Many of the system calls used for files are also used for devices.
  - ✓ A system with multiple users may require us to first request the device, to ensure exclusive use of it.
  - ✓ After using the device, it must be released using **release** system call. These functions are similar to open & close system calls of files.
  - ✓ Read, write & reposition system calls may be used with devices.
  - ✓ MS-DOS & UNIX merge the I/O devices & the files to form **file-device structure**.

4. Information maintenance

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

  - ✓ Many system calls exist for the purpose of transferring information between the user program and the operating system.
  - ✓ For example, most systems have a system call to return the current **time** and **date**.
  - ✓ Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.
  - ✓ The operating system also keeps information about all its processes, and system calls are used to access this information.
  - ✓ System calls are also used to reset the process information (**get process attributes and set process attributes**).

5. Communications

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

There are two models of inter-process communication:

❖ **Message Passing model**

- ✓ In message passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common **mailbox**.
- ✓ Each computer in a network will have a **host name.** Similarly, each process has a **process name**, and this name is translated into an identifier by which the operating system can refer to the process. The **get hostid** and **get processid** system calls do this translation.
- ✓ The identifiers are then passed to the general purpose **open** and **close** calls provided by the file system or to specific **open connection** and **close connection** system calls, depending on the system's model of communication.
- ✓ The recipient process must give itspermission for communication to take place with an **accept connection** system call.
- ✓ The receiving daemons execute a **wait forconnection** call and are awakened when a connection is made.
- ✓ The source of the communication, known as the **client**, and the receiving daemon, known as a **server**, exchange messages by using **read message** and **write message** system calls.
- ✓ The **close connection** call terminates the communication.

❖ **Shared Memory model**

- ✓ In shared memory model, processes use **shared memory create** and **shared memory attach** system calls to create and gain access to regions of memory owned by other processes.
- ✓ The OS tries to prevent one process from accessing another process's memory, so several processes have to agree to remove this restriction. Then they exchange information by reading and writing in the shared areas.
- ✓ The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

1.17 System Programs

- ✓ **System programs**, also known as system utilities, provide a convenient environment for program development and execution.Some of them are simply user interfaces to system calls and others are considerably more complex. They can be divided into these **categories**:

  i. **File management:** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

ii. **Status information:** Some programs asks the system for the date, time, amount of available memory or disk space, number of users, or similar status
iii. information. Others are more complex, providing detailed performance, logging, and debugging information.
iv. **File modification:** Several text editors are available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.
v. **Programming language support:** Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.
vi. **Program loading and execution:** Once a program is assembled or compiled, it must be loaded into memory to be executed. The operating system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders.
vii. **Communications:** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screen, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

In addition to system programs, most operating systems are supplied with **application programs** that are useful in solving common problems or performing common operations. Such application programs are word processors, text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages and games.

1.18 OS Design & Implementation

- **Design goals**
  - ✓ The first aspect in designing a system is defining goals and specifications. Next we need to define the mechanisms and policies to be implemented. Finally the implementation takes place.
  - ✓ At the highest level, the system design will be affected by the choice of hardware and type of system like timesharing, batch, distributed, real time, single user, multiuser OS or general purpose etc.
  - ✓ At the next level the requirements can be divided into two basic groups: **user** goals and **system** goals.
  - ✓ The **user goals** basically comprises of convenient to use, easy to learn and to use, reliable, safe and fast etc.
  - ✓ The **system goals** are from the designer's perspective that the system must be easy to design, create and maintain. It should also be flexible, reliable, error free and efficient.
  - ✓ The requirements vary from system to system. Different requirements result in different solutions and hence different Operating Systems.

- **Mechanisms and policies**
  - ✓ Mechanisms determine **how** to do something. Mechanisms that are insensitive to changes in policy are more desirable.

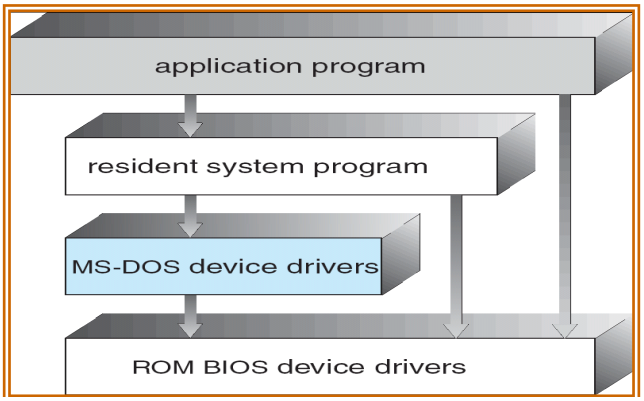
- ✓ Policy determines **what** will be done. Policies are likely to change across places and over time.
- ✓ Change in policy may require redefinition of certain parameters of the system.
- ✓ Policy decisions are important for all resource allocation.
  - **Ex**. **timer construct** is a **mechanism** to ensure CPU protection, whereas for **how long** the timer is to be set for a particular user is a **policy** decision.

- **Implementation**
  - ✓ Once an operating system is designed, it must be implemented.
  - ✓ Traditionally, operating systems have been written in **assembly language**.
  - ✓ MS-DOS was initially implemented in Intel 8088 and was available on Intel CPUs only. Master Control Program (MCP) written in ALGOL, MULTICS in PL/I, and Linux is with C and available on Intel 8086, Motorola 680, SPARC and MIPS RX000.
  - ✓ Now, they are most commonly written in **higher-level languages** such as **C or C++.**
  - ✓ The **advantages** of using higher-level languages are, the code can be written faster, it is more compact, and is easier to understand and debug.
  - ✓ The improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation.
  - ✓ Finally, an operating system is easier to **por**t(tomove to some other hardware) if it is written in a higher-level language.
  - ✓ The only possible **disadvantages** of implementing an operating system in a higher-level language are **reduced speed and increased storage requirements.**

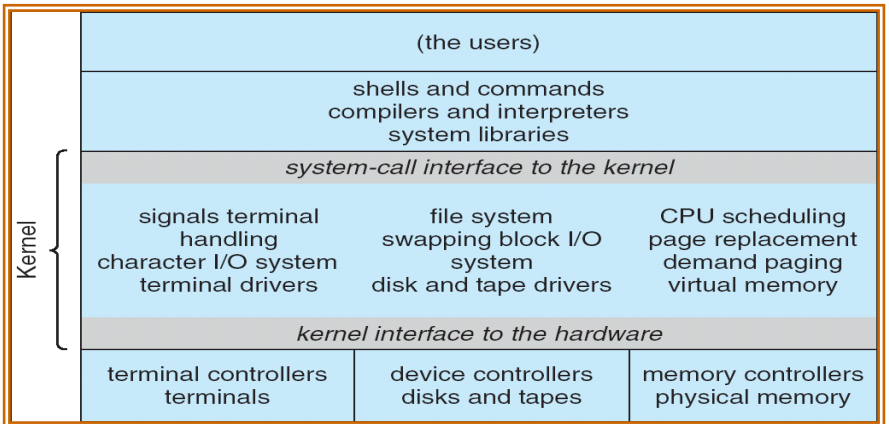
1.19 Operating System Structures

- ✓ Modern OS is large & complex. It consists of different types of components.These components are interconnected & melded into kernel.
- ✓ For designing the system, different types of structures are used. They are,
  - Simple structures.
  - Layered Approach.
  - Micro kernels.

- **Simple Structures**
  - ✓ Simple structure OS are small, simple & limited systems. The structure is not well defined.
  - ✓ MS-DOS is an example of simple structure OS. MS-DOS layer structure is shown in below **figure**

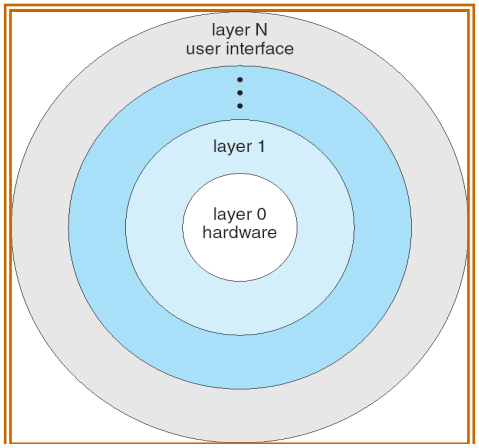


- ✓ In MS-DOS, the interfaces and levels of functionality are not well separated.
- ✓ For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives.
- ✓ **UNIX** is another example for simple structure. Initially it was limited by hardware functions.
  - It consists of **two** separable parts: the kernel and the system programs.
  - The kernel is further separated into series of interfaces & device drivers.
  - We can view the traditional UNIX operating system as being layered, as shown in **figure**
- ✓ Everything below the system-call interface and above the physical hardware is the kernel.
- ✓ Kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.
- ✓ This **monolithic** structure was difficult to implement and maintain.



- **Layered Approach**

- ✓ A system can be made modular in many ways. One method is the **layered approach** in which the OS is divided into number of layers, where one layer is built on the top of another layer.
- ✓ The bottom layer (layer 0) is **hardware** and higher layer (layer N) is the **user interface**. This layering structure is depicted in below **figure 2.8**.



- ✓ An OS is an implementation of **abstract object** made up of data & operations that manipulate these data.

- ✓ A typical operating-system layer say layer *M* consists of data structures and a set of routines that can be invoked by higher-level layers. Layer *M*, in turn can invoke operations on lower level layers.
- ✓ The main **advantage** of layered approach is the **simplicity** i.e. each layer uses the services & functions provided by the lower layer. This approach simplifies the debugging & verification. Once first layer is debugged the correct functionality is guaranteed while debugging the second layer. If an error is identified then it is a problem in that layer because the layer below is already debugged.
- ✓ Each layer tries to hide some data structures, operations & hardware from the higher level layers.
- ✓ A problem with layered implementation is that they are less efficient.

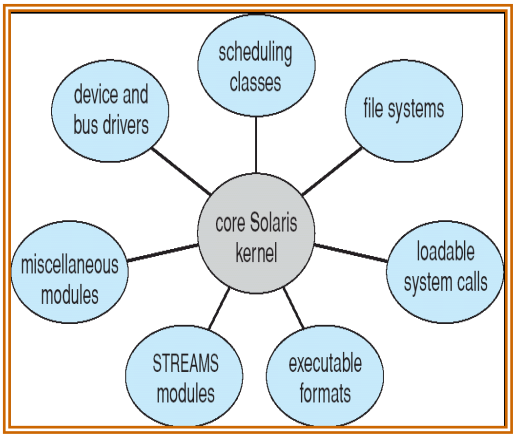
- **Micro Kernels**

- ✓ In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the operating system by removing all nonessential components from the kernel and implementing them as system and user level programs. The result is a smaller kernel.
- ✓ The main function of the micro kernels is to provide **communication facilities** between the client program and various services that are running in user space.
- ✓ This approach provided a high degree of flexibility and modularity.
- ✓ It includes the ease of extending OS. All the new services are added to the user space & do not need the modification of kernel.
- ✓ This approach also provides more **security**&**reliability**.
- ✓ Most of the services will be running as user process rather than the kernel process.
- ✓ A micro kernel in Windows NT provides **portability** and **modularity**.

- **Modules**

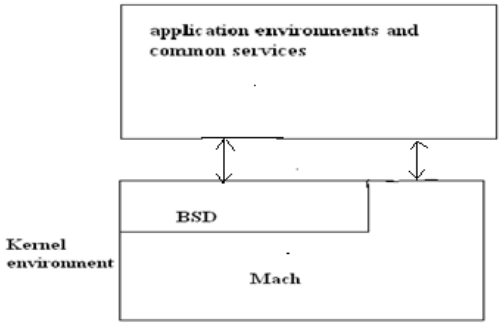
- ✓ The best current methodology for operating-system design involves using object-oriented programming techniques to create a modular kernel.
- ✓ Here, the kernel has a set of core components and links in additional services either during boot time or during run time. Such a strategy uses dynamically loadable modules and is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X.
- ✓ For example, the Solaris operating system structure, shown in the **figure**, is organized around a core kernel with **seven types** of loadable kernel modules:

1.Scheduling classes
2.File systems
3.Loadable system calls
4.Executable formats
5.STREAMS modules
6.Miscellaneous
7.Device and bus drivers



- ✓

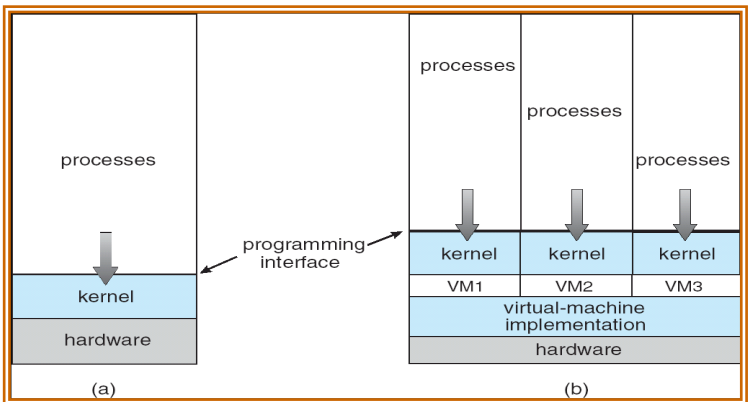
- ✓ The Apple Macintosh Mac OS X operating system uses a hybrid structure. It is a layered system in which one layer consists of the Mach microkernel. The structure of Mac OS X appears as shown in **figure**



- ✓ The top layers include application environments and a set of services providing a graphical interface to applications.
- ✓ Below these layers is the kernel environment, which consists primarily of **the Mach microkernel and the BSD kernel.**
- ✓ Mach provides memory management, support for remote procedure calls (RPCs) and interprocess communication facilities, including message passing and thread scheduling.
- ✓ The BSD component provides a BSD command line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads.

1.20Virtual Machines

- ✓ The fundamental idea behind a virtual machine is to **abstract** the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so on) into several different **execution environments**, thereby creating the illusion that each separate execution environment is running its own **private computer**.
- ✓ By using CPU scheduling and virtual-memory techniques, an operating system can create the illusion that a process has its own processor with its own (virtual) memory.
- ✓ Each process is provided with a (virtual) copy of the underlying computer as shown in the below **figure**



(a) Non virtual machine (b) virtual machine

- ✓ A major difficulty with the virtual machine approach involves disk systems. Suppose that the physical machine had three disk drives but wanted to support seven virtual

machines. Clearly, it could not allocate a disk drive to each virtual machine, because the virtual machine software itself will need substantial disk space to provide virtual memory and spooling. The solution is to provide virtual disks-termed **minidisks**in IBM's VM operating system which are identical in all respects except size.

- **Implementation**

  - ✓ It is difficult to implement VM concept. Much work is required to provide an **exact**duplicate of the underlying machine.
  - ✓ The machine typically has two modes: **user mode and kernel mode**.
  - ✓ The virtual-machine software can run in kernel mode, since it is the operating system. The virtual machine itself can execute in only user mode.
  - ✓ The major difference between virtual and non virtual m/c is **time**. The real I/O might have taken 100 milliseconds, the virtual I/O might take less time (because it is spooled) or more time (because it is interpreted). In addition, the CPU is being multi programmed among many virtual machines, further slowing down the virtual machines in unpredictable ways.

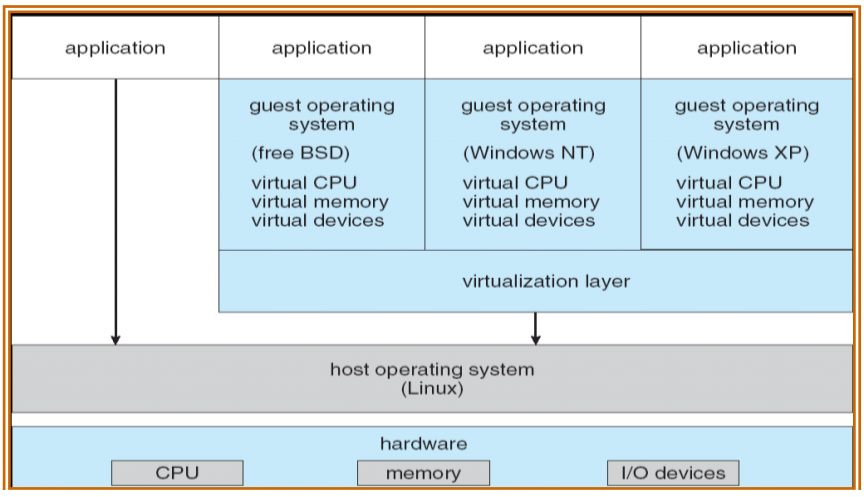
- **Benefits**

  - ✓ The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation permits no direct sharing of resources.
  - ✓ A virtual-machine system is a perfect vehicle for operating-systems research and development.
  - ✓ System programmers are given their own VM, and system development is done on the virtual machine instead on a physical machine. Thus changing OS will not cause any problem.

Examples

1. **VMware**

   - ✓ The architecture is shown below **figure.**

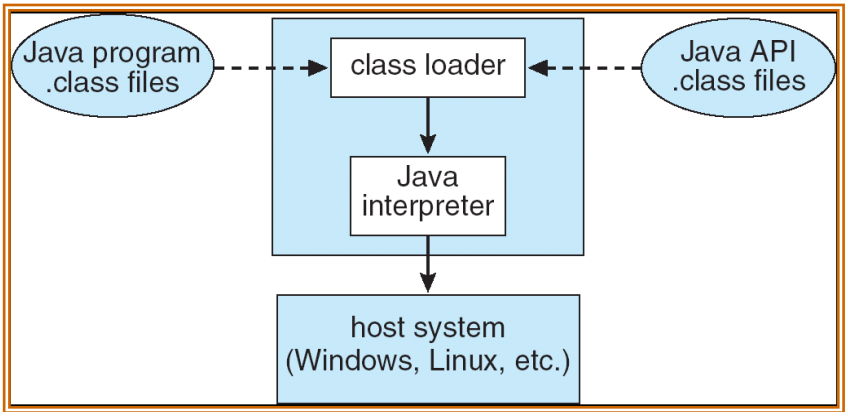


- ✓ It is a popular commercial application that abstracts Intel X86 and compatible hardware into isolated virtual machines.
- ✓ It runs as an application on a host operating system such as Windows or Linux and allows this host system to concurrently run several different guest operating systems as independent virtual machines.
- ✓ Here, Linux is running as the host operating system and FreeBSD, Windows NT, and Windows XPare running as guest operating systems.
- ✓ The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems.
- ✓ Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so on.

VMware architecture

2. Java virtual machine

- ✓ Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995.
- ✓ In addition to a language specification and a large API library, Java also provides a specification for a Java virtual machine-or JVM.
- ✓ Java objects are specified with the **class** construct. A Java program consists of one or more classes. For each Java class, the compiler produces an **architecture-neutral bytecode output(.class) file** that will run on any implementation of the JVM.
- ✓ The JVM is a specification for an abstract computer. It consists of a class loader and a Java interpreter that executes the architecture-neutral byte codes, as given in **figure**



- ✓ The class loader loads the compiled **.class** files from both the Java program and the Java API for execution by the Java interpreter.
- ✓ After a class is loaded, the verifier checks that the **.class** file is valid Java bytecode and does not overflow or underflow the stack. It also ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access.
- ✓ If the class passes verification, it is run by the Java interpreter.
- ✓ The JVM also automatically manages memory by performing **garbage collection** - the practice of reclaiming memory from objects no longer in use and returning it to the system.
- ✓ The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or Mac OS X, or as part of a Web browser.

1.21 System Boot

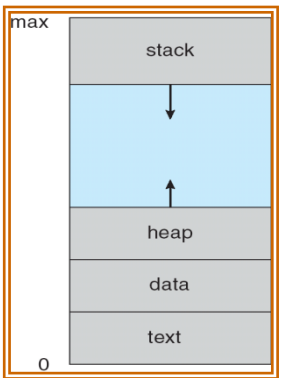
- ✓ The procedure of starting a computer by loading the kernel is known as **booting** the system.
- ✓ **Bootstrap program or Bootstrap loader** locates the kernel, loads it into main memory and start its execution.
- ✓ Bootstrap program is in the form of **read only memory (ROM)** because the RAM is in unknown state at a system startup. All forms of ROM are knows as **firmware**.
- ✓ For large OS like Windows, Mac OS, the Bootstrap loaders is stored in firmware and the OS is on disk.
- ✓ Bootstrap has a bit code to read a single block at a fixed location from disk into the memory and execute the code from that **boot block**.
- ✓ A disk that has a boot partition is called a **boot disk or system disk**.

1.22 PROCESS CONCEPTS: Process Concepts

- ✓ Process is an **active** entity. A process is a sequence of instruction execution. Process exists in a limited span of time. Two or more process may execute the same program by using its own data & resources.
- ✓ A program is a **passive** entity which is made up of program statement. Program contains instructions.

- **The Process**

  - ✓ A process is more than the program code which is also called **text section**.
  - ✓ It contains **program counter** which represents the current activity and also the contents of the processor's registers.
  - ✓ A process also consists of a process **stack section** which contains temporary data &**data section** which contains global variables.
  - ✓ A process may also include a **heap**, which is memory that is dynamically allocated during process run time.
  - ✓ The structure of a process in memory is shown in below **figure**



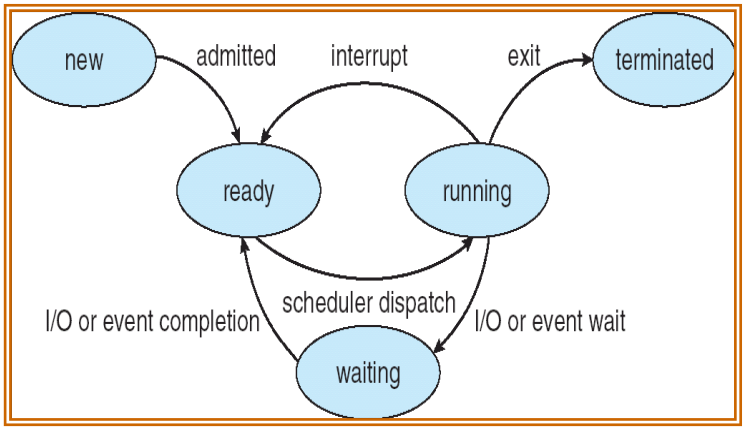
- **Process State**

  - ✓ As process executes it changes its state and each process may be in one of the following states:
    - **New:** The process is being created
    - **Running:** Instructions are being executed

- **Waiting:** The process is waiting for some event to occur
- **Ready:** The process is waiting to be assigned to a process
- **Terminated:** The process has finished execution

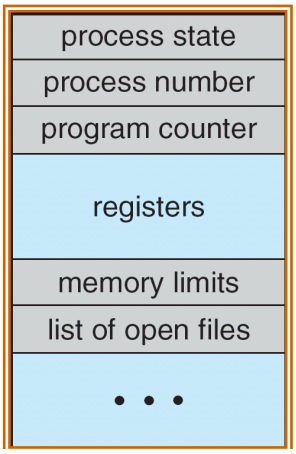
✓ Only one process can be running on any processor at any instant. Many processes may be ready and waiting.

✓ The **state diagram** corresponding to these states is shown below **figure**



- **Process Control Block**

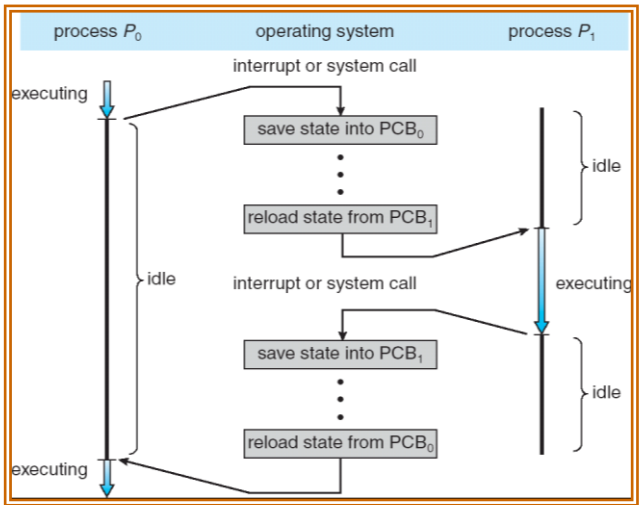
✓ A process in an operating system is represented by a **data structure** known as a **Process Control Block (PCB)** and it is also called as **task control block**. The following **figure** shows the process control block.



✓ The PCB contains important **information** about the specific process including,

- **Process state:** The current state of the process i.e., whether it is ready, running, waiting, halted and so on.
- **Program counter:** Indicates the address of the next instruction to be executed for a process.
- **CPU registers:** The registers vary in **number** and **type**. Along with program counter this state information should be saved to allow process to be continued correctly after an interrupt occurs.
- **CPU scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

- **Memory-management information:** This information may include the value of **base** and **limit** registers, the page tables, or the segment tables, depending on the memory system used by the OS.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.



CPU switch from process to process

- **Threads**

✓ A process is a program that performs a single **thread** of execution. **For example**, when a process is running a word-processor program, a single thread of instruction is being executed. This single thread of control allows the process to perform only one task at one time. The user cannot simultaneously type in characters and run the spell checker within the same process.

✓ Many modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. On a system that supports threads, the PCB is expanded to include information for each thread. Eg: Windows OS and UNIX

1.23 Process Scheduling

The **process scheduler** selects an available process for execution on the CPU.

- **Scheduling queues**

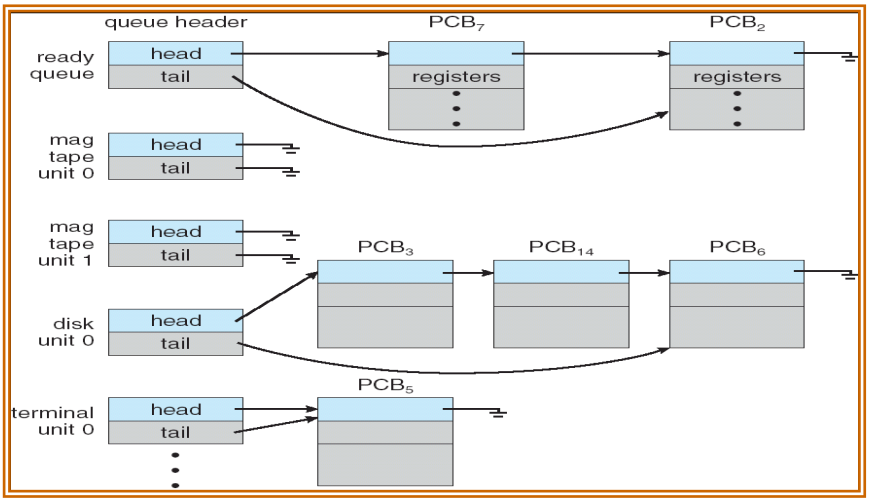
✓ The following are the different types of process scheduling queues.

- **Job queue:** set of all processes in the system.

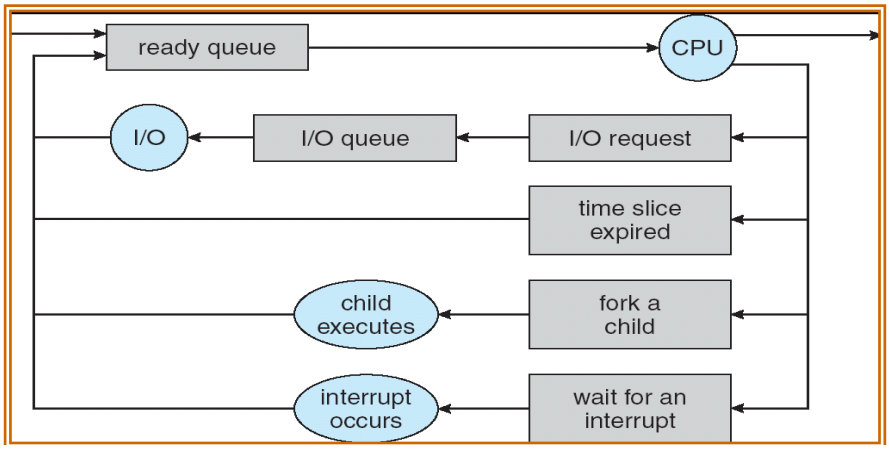
- **Ready queue**:The processes that are placed in main memory and are **ready and waiting** to execute are placed in a list called the ready queue. This is in the form of linked list, the header contains pointer to the first and final PCB in the list. Each PCB contains a pointer field that points to next PCB in ready queue.

- **Device queue**:The list of processes waiting for a particular **I/O device** is called device queue. When the CPU is allocated to a process it may execute for some time and may quit or interrupted or wait for the occurrence of a particular event like completion of an I/O request, but the I/O may be busy with some other processes. In this case the process must wait for I/O and it will be placed in device queue. Each device will have its own queue.

✓ The below **figure** shows Ready queue and various I/O Device queues



✓ The process scheduling is represented using a **queuing diagram** as shown in below **figure 3.5**. **Queues** are represented by the **rectangular box** and **resources** they need are represented by **circles**, and the **arrows** indicate the **flow of processes** in the system.



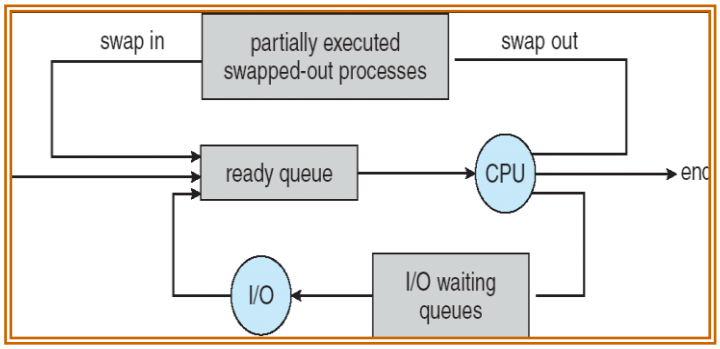
✓ A new process is initially put in the ready queue and it waits there until it is selected for execution or **dispatched**. Once the process is assigned CPU and is executing, the following **events** can occur,

- It can execute an I/O request and is placed in I/O queue.
- The process can create a sub process & wait for its termination.
- The process may be removed from the CPU as a result of interrupt and can be put back into ready queue.

- **Schedulers**

  ✓ The following are the **different types** of schedulers,

  - **Long-term scheduler (or job scheduler):** selects which processes should be brought into the ready queue. Long-term scheduler is invoked very infrequently (in terms of seconds or minutes). The long-term scheduler controls the **degree of multiprogramming** (number of processes in main memory)**.**
  - **Short-term scheduler (or CPU scheduler):** selects which process should be executed next and allocates CPU. Short-term scheduler is invoked very frequently**.**
  - **Medium-term schedulers:** Some OS introduces intermediate level of scheduling called Medium-term schedulers as shown in **figure**. It can be advantageous to remove processes from memory and thus reduce the degree of multiprogramming. The process can be later reintroduced into memory, and its execution can be continued where it left off. The process is **swapped out**, and is later **swapped in**, by the **medium-term scheduler**.



  ✓ Processes can be described as either:
  - **I/O-bound process**: processes spend more time doing I/O than computations.
  - **CPU-bound process**: processes spend more time doing computations; and generates I/O request less frequently.

- **Context Switch**

  ✓ When an **interrupt** occurs, the system needs to save the current**context** of the process running on the CPU. The context is represented in the **PCB** of the process.

  ✓ When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process. A **state save** of the current state of the CPU, and then **state restore** to resume operations is performed.

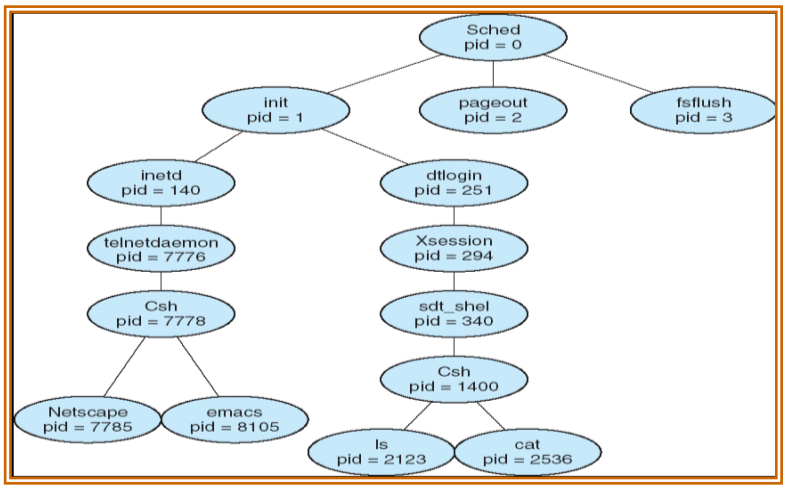
  ✓ Context-switch time is overhead and the system does no useful work while switching. Context-switch times are highly dependent on hardware support. Context-switch speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions.

1.24 Process Operations

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

- **Process Creation**

  - ✓ A process may create several new processes by some **create-process** system call,during the course of execution.
  - ✓ The creating process is called **parent** process and the created one is called the **child** process. Each of the new process may in turn create other processes, forming a **tree** of processes. Processes are identified by unique **process identifier (** or**pid).**
  - ✓ **Figure3.7** shows the process tree for the solaris OS. The process at the top of the tree is **sched** process, with pid of 0,and this creates several children processes. The sched process creates several children processes including **pageout and fsflush**. These processes are responsible for managing memory and file systems. The sched process also creates the **init** process, which serves as the root parent process for all user processes. These processes are responsible for managing memory and file systems.
  - ✓ inetd and dtlogin are two children of init where inetd is responsible for networking services such as telnet and ftp; dtlogin is the process representing a user login screen.
  - ✓ When a user logs in, dtlogin creates an X-windows session (Xsession), which in turns creates the sdt_shel process. Below sdt_shel, a user's command-line shell, the C-shell or csh is created. In this command line interface, the user can then invoke various child processes, such as the ls and cat commands.
  - ✓ There is also csh process with pid of 7778 representing a user who has logged onto the system using telnet. This user has started the Netscape browser (pid of 7785) and the emacs editor (pid of 8105).
  - ✓ A process needs certain resources to accomplish its task. Along with the various logical and physical resources that a process obtains when it is created, **initialization data** may be passed along by the parent process to the child process.

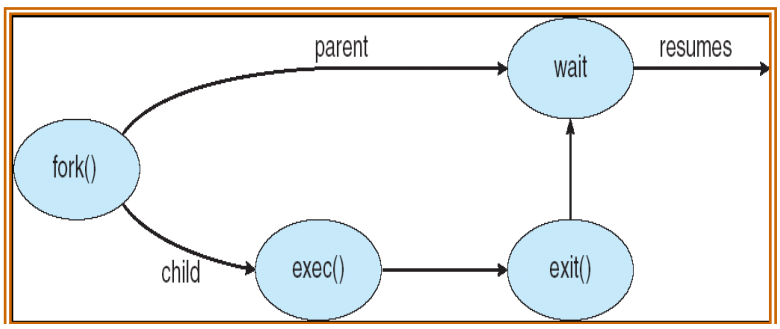


- ✓ When a process creates a new process, **two possibilities** exist in terms of **execution.**
  1. The parent continues to execute concurrently with its children.
  2. The parent waits until some or all of the children have terminated.
- ✓ There are also **two possibilities** in terms of the **address space** of the new process.
  1. The child process is a duplicate of the parent process.
  2. The child process has a new program loaded into it.
- ✓ In UNIX OS,**fork()** system call creates new process. In windows CreateProcess() does the job.
- ✓ **Exec()** system call is called after a **fork()** to replace the process memory space with a new program.
- ✓ The **C program** shown below illustrates these system calls.

```
int main()
{
Pid_tpid;
pid = fork();/* fork another process */

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 */
         {
        wait (NULL);/* parent will wait for the child to complete */
printf ("Child Complete");
        exit(0);
        }
  }
```

- ✓ If there are two different processes running a copy of the same program, the pid for child is **zero** and for the parent it is **greater than zero**. The parent process waits for the child process to complete with the **wait()** system call.
- ✓ When the child process completes, the parent process resumes from the call to wait(), where it completes using **exit()** system call. This is shown in below**figure**



- **Process Termination**

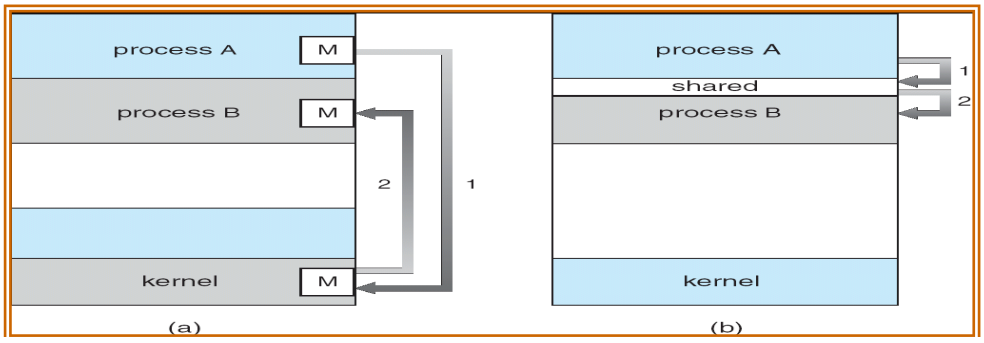
- ✓ A process terminates when it finishes executing its last statement and asks the operating system to delete it by using **exit()** system call.

- ✓ Process resources are deal located by the operating system. A process can terminate another process via **Terminate Process()** system call. A Parent may terminate execution of children processes (**abort**) for the following **reasons**.
  - Child has exceeded usage of allocated resources.
  - Task assigned to child is no longer required.
  - If parent is exiting some operating system do not allow child to continue if its parent terminates.
- ✓ Some systems does not allow child to exist if its parent has terminated. If process terminates then all its children must also be terminated, this phenomenon is referred as **cascading termination**.

1.25 Interprocess Communication (IPC)

- ✓ Processes executing concurrently in the operating system may be either **independent** processes or **cooperating** processes.
- ✓ A process is **independent** if it cannot affect or be affected by the other processes executing in the system. Any process that **does not share data** with any other process is independent.
- ✓ A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Any process that **shares data** with other processes is a cooperating process.
- ✓ **Advantages** of process cooperation are,
  - **Information sharing**: several users may be interested in same piece of information, so an environmentmust be provided to allow concurrent access to such information.
  - **Computation speed-up**: If we want particular task to run faster,it can be broken into subtasks,each of which will be executing in parallel with each other.
  - **Modularity**: If the system is to be constructed in modular fashion, then system functions can be divided into separate processes or threads.
  - **Convenience**: An individual user may work on many tasks at the same time.

- ✓ Cooperating processes require an **Interprocess Communication (IPC)** mechanism that will allow them to **exchange** data and information. There are **two fundamental models** of IPC as shown in below **figure**.
  1. **Shared memory**: A region of memory that is shared by cooperating processes is established. Processes then exchange information by reading and writing data to the shared region.
  2. **Message passing**: Communication takes place by means of message exchange between the cooperating processes.



✓ **The differences between these two models are,**

Message passing	Shared memory
a. Useful for exchangingsmall amount of data.	a. large data
b. easy to implement	b. complex
c. slower	c. faster
d. implemented using system calls	d.system calls are required only toestablish Shared memory region

- **Shared Memory System**

✓ A region of memory that is shared by cooperating processes is established. Processes then exchange information by reading and writing data to the shared region.
✓ To illustrate cooperating processes, consider **producer-consumer problem**.
✓ Producer process produces information that is consumed by a consumer process.
✓ One **solution** to producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be a **buffer of items** that can be filled by a producer and emptied by consumer.The buffer will reside in a shared memory region.
✓ The producer can produce one item while the consumer is consuming another item.The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced by the producer.

✓ **Two types** of buffers can be used.
  - **unbounded-buffer:**places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
  - **bounded-buffer:** assumes that there is a fixed buffer size, sothe consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

✓ The following **variables** reside in a region of memory shared by the producer and consumer processes

```
#define BUFFER_SIZE 10
typedefstruct {
……..
……..

}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

✓ The shared buffer is implemented as a circular array with **two** logical pointers **in and out**. The variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer. The buffer is empty when in= =out, the buffer is full when ((in+ 1)% BUFFER_SIZE) = = out.

✓ ***The code for the producer process is shown below.***

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

✓ ***The code for the consumer process is shown below.***

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

- **Message Passing System**

✓ Communication takes place by means of **message exchange** between the cooperating processes.
✓ Message passing facility provides two operations.
  - Send(message)
  - Receive(message)

✓ Message size can be fixed or variable.
✓ If P and Q wish to communicate, they need to establish a **communicationlink** between them and communication link can be,
  - physical (eg: shared memory, hardware bus)
  - logical (eg: logical properties)

✓ Several methods for logically implementing a link are,
  - Direct or Indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

➔ **Naming**

✓ Processes that want to communicate must have a **way to refer** to each other. They can use either **direct** or **indirect** communication.
✓ Under **Direct Communication** processes must name each other explicitly
✓ The send() and receive() primitives are defined as,
  - **Send(P, message)** – send a message to process P.
  - **Receive(Q, message)** – receive a message from process Q.

✓ **Properties** of communication link in this scheme are,
  - Links are established automatically between every pair of processes that want to communicate.
  - A link is associated with exactly two communicating processes.
  - Between each pair there exists exactly one link.

✓ This scheme exhibits **two** types of **addressing**,
  - **Symmetry**: Both sender and receiver must name the other to communicate.

- **Asymmetry**: Only the sender names the recipient, the recipient is not required to name the sender.The send() and receive() primitives are defined as,

  Send (P, message) – send a message to process P

  Receive(id, message) – receive a message from any process; the variable **id** is set to the name of the process with which communication has taken place.

- ✓ In **Indirect Communication** messages are sent and received from **mailboxes** (also referred to as **ports**)
- ✓ A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- ✓ Each mailbox has a unique **id** and processes can communicate only if they share a mailbox.

- ✓ The send() and receive() primitives are defined as,
  - **Send(A, message) –** send a message to mailbox A
  - **Receive(A, message)** – receive a message from mailbox A
- ✓ Properties of communication link are,
  - Links are established only if processes share a common mailbox.
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
- ✓ OS allows the process to do the following operations
  - **create**a new mailbox
  - **send** and receive messages through mailbox
  - **destroy** a mailbox

➔ **Synchronization**

- ✓ Message passing may be either **blocking or non-blocking. Blocking** is considered **synchronous. Non-blocking** is considered **asynchronous.**

  - **Blocking send:** The sending process is blocked until the message isreceived by the receiving process or by the mailbox.
  - **Non-blocking send:** The sending process sends the message and resumesoperation.
  - **Blocking receive:** The receiver blocks until a message is available.
  - **Non-blocking receive:** The receiver retrieves either a valid message or a null.

➔ **Buffering**

- ✓ Messages exchanged by communicating processes reside in a temporary queue, and such queues can be implemented in one of **three** ways,
  1. **Zero capacity** – Maximum length is **zero** and sender must wait for the receiver.
  2. **Bounded capacity** – **finite** length ofn messages, sender must wait if link is full.
  3. **Unbounded capacity** – **infinite** length and sender never waits.

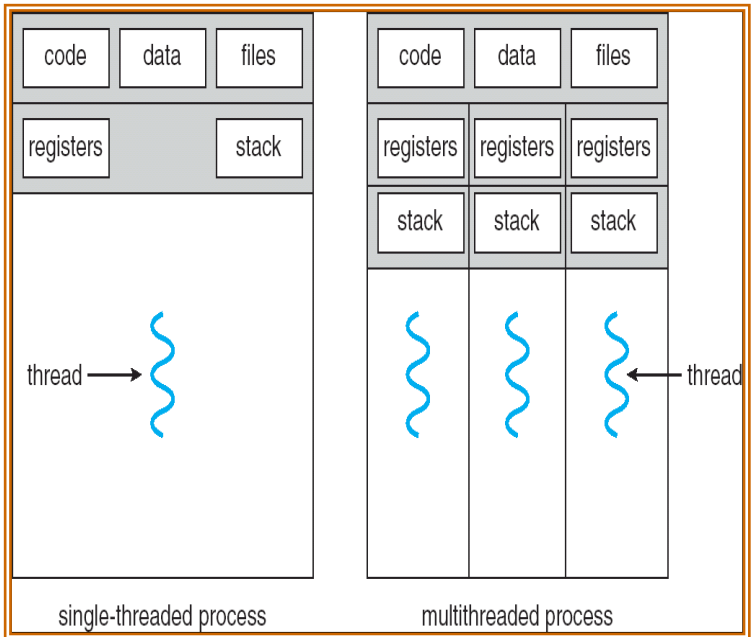
---0o0---

MODULE-2
MULTITHREADED PROGRAMMING

2.1 Overview

Threads

- ✓ A thread is a basic unit of CPU utilization.
- ✓ It comprises a **thread ID, a program counter, a register set, and a stack**. It shares its **code section, data section**, and other operating-system resources, such as **open files and signals** with **other threads** belonging to the same process.
- ✓ A traditional (or **heavyweight**) process has a **single thread** of control. If a process has **multiple threads** of control, it can perform more than one task at a time.
- ✓ The below f**igure 4.1** illustrates the difference between a **traditional single threaded** process and a **multithreaded** process.



- **Motivation**

  - ✓ Many software packages that run on modern desktop PCs are multithreaded.
  - ✓ An application is implemented as a separate process with several threads of control. Eg: A Web browser might have one thread to display images ortext while another thread retrieves data from the network.
  - ✓ **Process creation** takes **more time** than **threadcreation.**It is more efficient to use process that contains multiple threads, so that the amount of time that a client have to wait for its request to be serviced from the web server will be less.
  - ✓ Threads also play an important role in **remote procedure call.**

- **Benefits**

  - ✓ The benefits of multithreaded programming
    1. **Responsiveness:** Multithreading allows program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
    2. **Resource Sharing:** Threads share the memory and resources of the process to which they belong. The benefit of sharing code and data is that it allows an

application to have several different threads of activity within the same address space.

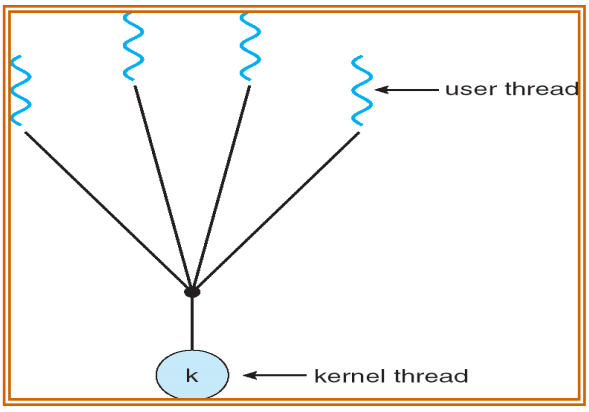
3. **Economy:** Because of resource sharing context switching and thread creation are fast when working with threads.
4. **Utilization of multiprocessor architectures:** Threads can run in parallel on different processors. Multithreading on multi-CPU machine increases concurrency.

2.2 Multithreading Models

- ✓ Support for threads may be provided either at user level for **user threads** or by the kernel for **kernel threads**.
- ✓ There **must be a relationship** between user threadsand kernel threads. **Three** common ways of establishing this relationship are**,**

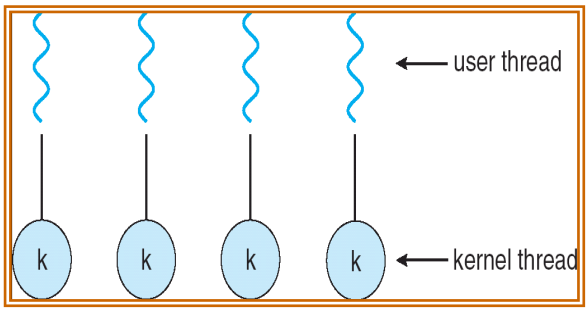
- **Many-to-One**

  - ✓ Many user-level threads are mapped to single kernel thread as shown in below **figure below**.
  - ✓ This model is efficient as the thread management is done by the thread library in user space, but the entire process will block if a thread makes a blocking system call.
  - ✓ As only one thread can access the kernel thread at a time, multiple threads are unable to run in parallel on multiprocessors.
  - ✓ **Examples:**Solaris Green Threads, GNU Portable Threads



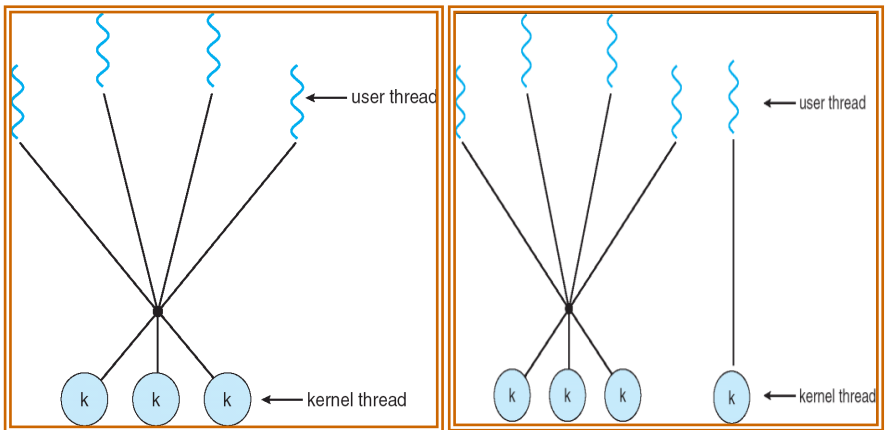
- **One-to-One**

  - ✓ Each user-level thread maps to kernel thread as shown in **figure 4.3**
  - ✓ It provides more concurrency than Many-to-One model by allowing thread to run when a thread makes a blocking system call.
  - ✓ It allows multiple threads to run in parallel on multiprocessors.
  - ✓ The only **drawback** is, creating a user thread requires creating the corresponding kernel thread and it burdens performance of an application.
  - ✓ **Examples:** Windows NT/XP/2000, Linux



- **Many-to-Many Model**

  - ✓ **One-to-One** model **restricts** creating more user threads and **Many-to-One** model allows creating more user threads but kernel can schedule only one thread at a time. These drawbacks can be **overcome** by Many-to-Many model as shown in below **figure.(Left side)**
  - ✓ Many-to-Many model allows many user level threads to be mapped to many kernel threads.
  - ✓ It allows the operating system to create a sufficient number of kernel threads.
  - ✓ When thread performs a blocking system call, the kernel can schedule another thread for execution.
  - ✓ It allows user-level thread to be bound to a kernel thread and this is referred as **two- level model**as shown in below **figure**.**(Right side)**
  - ✓ **Examples:** IRIX, HP-UX, Solaris OS.



2.3 Thread Libraries

- ✓ A thread library provides the programmer an **API** for creating and managing threads.
- ✓ There are **two primary ways** of implementing a thread library.
  - o The first approach is to provide a **library entirely in user space** with **no kernel support.** All code and data structures for the library exist in user space.
  - o The second approach is to implement a **kernel-level library supported directly by the operating system.**
- ✓ Three primary thread libraries are

- POSIX Pthreads: extension of posix standard, they may be provided as either a user bor kernel library.
- Win32 threads: is a kernel level library available on windows systems.
- Java threads: API allows creation and management directly in Java programs. However, on windows java threads are implemented using win32 and on UNIX and Linux using Pthreads

- **Pthreads**

✓ Pthreads, the threads extension of the POSIX standard, may be provided as either a user or kernel-level library.
✓ Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.
✓ This is a specification for thread behavior, not an implementation.
✓ Operating system designers may implement the specification in any way they wish. Numerous systems implement the Pthreads specification, including Solaris, Linux, Mac OS X, and Tru64 UNIX.
✓ **Shareware** implementations are available in the public domain for the various Windows operating systems as well.

Ex: Multithreaded C program using the Pthreads API

```
#include <pthread.h>
#include <stdio.h>
    int sum; /* this data is shared by the thread(s) */
    void *runner(void *param); /* the thread */
    int main(intargc, char *argv[])
    {
    pthread_ttid; /* the thread identifier */
    pthread_attr_tattr;  /* set of thread attributes */

    if (argc != 2)
    {
    fprintf(stderr,"usage: a.out<integer value>\n");
    return -1;
    }
    if (atoi(argv[1]) < 0)
    {
    fprintf(stderr,"%d must be>= 0\n",atoi(argv[1]));
    return -1;
    }
    pthread_attr_init(&attr);/* get the default attributes */
    pthread_create(&tid,&attr,runner,argv[1]);/* create the thread */
    pthread_join(tid,NULL);/*wait for the thread to exit */
    printf("sum = %d\n",sum);
    }

void *runner(void *param)/* The thread will begin control in this function */
    {
    int i, upper= atoi(param);
    sum = 0;
```

```
for (i = 1; i <= upper; i++)
sum += i;
pthread_exit(0) ;
}
```

- **Win32 Threads**

  - ✓ The Win32 thread library is a kernel-level library available on Windows systems.
  - ✓ The technique for creating threads using the Win32 thread library is similar to the Pthreads technique in several ways. We must include the windows.h header file when using the Win32 API.
  - ✓ Threads are created in the Win32 API using the CreateThread() function and a set of attributes for the thread is passed to this function.
  - ✓ These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state.
  - ✓ The parent thread waits for the child thread using the WaitForSingleObject() function, which causes the creating thread to block until the summation thread has exited.

Ex: Multithreaded C program using the Win32 API

```
#include <Windows.h>
#include <stdio.h>
DWORD Sum;   /* data is shared by the thread(s) */
/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
DWORD Upper = *(DWORD*)Param;
for (DWORD i = 0; i <= Upper; i++)
Sum += i;
return 0;
}

int main(intargc, char *argv[])
{
DWORD ThreadId;
HANDLE ThreadHandle;
intParam;

if (argc != 2)/* perform some basic error checking */
{
fprintf(stderr,"An integer parameter is required\n");
return -1;
}
Param = atoi(argv[1]);
if (Param< 0)
{
fprintf(stderr,"An integer>= 0 is required\n");
return -1;
}
                                        /*create the thread*/
```

```
ThreadHandle = CreateThread(NULL, 0,Summation,&Param,0,&ThreadId);

                                        // NULL:default security attributes
                                        // 0: default stack size
                                        // Summation: thread function
                                        // &Param: parameter to thread
function
                                        // 0: default creation flags
                                              //ThreadId: returns the
thread identifier

if (ThreadHandle != NULL)
 {
WaitForSingleObject(ThreadHandle,INFINITE);// now wait for the thread to
finish
CloseHandle(ThreadHandle);// close the thread handle
printf("surn = %d\n" ,Sum);
}
}
```

- **Java Threads**

  - ✓ The Java thread API allows thread creation and management directly in Java programs.
  - ✓ Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads.
  - ✓ All Java programs comprise at least a single thread of control and even a simple Java program consisting of only a **main()** method runs as a single thread in the JVM.
  - ✓ There are **two techniques** for creating threads in a Java program. One approach is to create a new class that is derived from the **Thread class** and to override its **run()** method. An alternative and more commonly used technique is to define a class that implements the **Runnable interface**. The **Runnable interface** is defined as follows:

```
public interface Runnable
{
public abstract void run () ;
}
```

  - ✓ When a class implements Runnable, it must define a **run()** method. The code implementing the run() method runs as a separate thread.
  - ✓ Creating a Thread object does not specifically create the new thread but it is the **start()** method that actually creates the new thread. Calling the **start()** method for the new object does two things:
    - o It allocates memory and initializes a new thread in the JVM.
    - o It calls the **run()** method, making the thread eligible to be run by the JVM.
  - ✓ As Java is a **pure object-oriented** language, it has **no notion of global data.** If two or more threads have to share data means then the sharing occurs by passing reference to the shared object to the appropriate threads.
  - ✓ This shared object is referenced through the appropriate **getSum() and setSum()** methods.

- ✓ As the Integer class is **immutable**, that is, once its value is set it cannot change, a new **sum** class is designed.
- ✓ The parent threads in Java uses join() method to wait for the child threads to finish before proceeding.

Ex: Java program for the summation of a non-negative integer.

```
class Sum
{
privateint sum;
publicintgetSum()
{
return sum;
}

public void setSum(int sum)
{
this.sum= sum;
}
}
class Summation implements Runnable
{
privateint upper;
private Sum sumValue;

public Summation(int upper, Sum sumValue)
{
this.upper = upper;
this.sumValue = sumValue;
}

public void run()
{
int sum = 0;
for (int i = 0; i <= upper; i++)
sum += i;
sumValue.setSum(sum);
      }
}

public class Driver
{
public static void main(String[]args)
{
if (args.length> 0)
{
if (Integer.parseint(args[O]) < 0)
System.err.println(args[O] + "must be>= 0.");
else
{
    Sum sumObject = new Sum();      //create the object to be shared
```

```
int upper= Integer.parseint(args[O]);
Thread thrd =new Thread(new Summation(upper, sumObject));
thrd.start();
try
{
        thrd. join () ;
System.out.println("The sum of "+upper+" is "+sumObject.getSum());
}
catch (InterruptedExceptionie) { }
}
    }
else
System.err.println("Usage: Summation <integer value>");
 }
```

2.4 Threading Issues

- **The fork() and exec() System Calls**

  - ✓ The semantics of the **fork() and exec()** system calls change in a multithreaded program. Some UNIX systems have chosen to have **two versions** of **fork(),one** that duplicates all threads and **another** that duplicates only the thread that invoked the fork() system call.
  - ✓ If a thread invokes the **exec()** system call, the program specified in the parameter to **exec()**will replace the entire process including all threads.
  - ✓ Which of the two versions of **fork()** to use depends on the application. If **exec()** is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate.

- **Thread Cancellation**

  - ✓ Thread cancellation is the task of terminating a thread before it has completed. **For example**, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled.
  - ✓ A thread that is to be cancelled is often referred to as the **target thread**. Cancellation of a target thread may occur in **two different** scenarios:
    - o **Asynchronous cancellation:** One thread immediately terminates the target thread.
    - o **Deferred cancellation:**The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

  - ✓ The **difficulty** with asynchronous cancellation occurs in situations where resources have been allocated to a cancelled thread or where a thread is cancelled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a cancelled thread but will not reclaim all resources. Therefore, cancelling a thread asynchronously may not free a necessary system-wide resource.

- ✓ With deferred cancellation, one thread indicates that a target thread is to be cancelled, but cancellation occurs only after the target thread has checked a flag to determine if it should be cancelled or not. This allows a thread to check whether it should be cancelled at a point when it can be cancelled safely. Pthreads refers to such points as **cancellation points.**

- **Signal Handling**

  - ✓ A signal is used in UNIX systems to notify a process that a particular event has occurred.
  - ✓ A signal may be received either synchronously or asynchronously, depending on the **source of** and **the reason** for the event being signaled.
  - ✓ All signals, whether synchronous or asynchronous, follow the same pattern:
    - o A signal is generated by the occurrence of a particular event.
    - o A generated signal is delivered to a process.
    - o Once delivered, the signal must be handled.
  - ✓ **Examples** of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal.
  - ✓ When a signal is generated by an event external to a running process, that process receives the signal **asynchronously.Examples** of such signals include terminating a process with specific keystrokes (such as <control><C>) and having **a timer expires**. An asynchronous signal is sent to another process.
  - ✓ Every signal may be handled by one of **two possible handlers**,
    - o A default signal handler
    - o A user-defined signal handler
  - ✓ Every signal has a **default signal handler** that is run by the kernel when handling that signal.
  - ✓ This default action can be overridden by a **user-defined signal handler** that is called to handle the signal.
  - ✓ Signals may be handled in different ways. Some signals (such as changing the size of a window) may simply be ignored; others (such as an illegal memory access) may be handled by terminating the program.
  - ✓ Delivering signals is more complicated in multithreaded programs. The following **options exist**to delivera signal:

    - o Deliver the signal to the thread to which the signal applies.
    - o Deliver the signal to every thread in the process.
    - o Deliver the signal to certain threads in the process.
    - o Assign a specific thread to receive all signals for the process.

- **Thread Pools**

  - ✓ The idea behind a thread pool is to **create a number of threads** at process startup and place them into a pool, where they sit and wait for work.
  - ✓ When a server receives a request, it awakens a thread from this pool and passes the request to it to service.
  - ✓ Once the thread completes its service, it returns to the pool and waits for more work.
  - ✓ If the pool contains no available thread, the server waits until one becomes free.
  - ✓ The **benefits**of Thread pools are,

- Servicing a request with an existing thread is usually faster than waiting to create a thread.
- A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

✓ The **number of threads** in the pool can be set based on **factors** such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests.

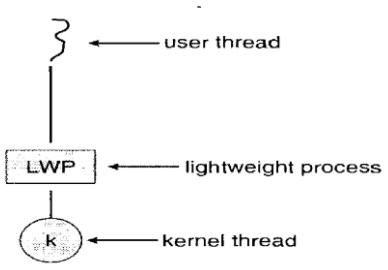
- **Thread-Specific Data**

✓ Threads belonging to a process share the data of the process. This sharing of data provides one of the benefits of multithreaded programming. But, in some circumstances, each thread might need its own copy of certain data. Such data is called as **thread-specific data**. **For example**, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction may be assigned a unique identifier.

✓ Most thread libraries including Win32 and Pthreads provide support for thread-specific data.

- **Scheduler Activations**

✓ Many systems implementing either the many-to-many or two-level model place an **intermediate data structure** between the user and kernel threads. This data structure is known as a lightweight process, or LWP as shown in the following figure



✓ An application may require any number of LWPs to run efficiently.

✓ In a CPU-bound application running on a single processor only one thread can run at once, so one LWP is sufficient. An application that is I/O- intensive may require multiple LWPs to execute.

✓ One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It **works** as follows: The kernel provides an application with a set of **virtual processors (LWPs)**, and the application can schedule user threads onto an available virtual processor. The kernel must inform an application about certain events. This procedure is known as an **upcall.**

✓ Upcalls are handled by the thread library with an upcall handler, and upcall handlers must run on a virtual processor.

✓ One event that triggers an upcall occurs when an application thread is about to block. In this situation, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread. The kernel then allocates a new virtual processor to the application.

- ✓ The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and gives up the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor.
- ✓ When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run.
- ✓ The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor.
- ✓ After marking the unblocked thread as eligible to run, the application schedules an eligible thread to run on an available virtual processor.

➢ Differences

- ✓ The differences between process and thread are,

	Process	Thread
1.	It is called heavyweightprocess.	It is called lightweight process.
2.	Process switching needs interface with OS.	Thread switching does not need interface with OS.
3.	Multiple processes use more resources than multiple threads.	Multiple threaded processes use fewer resources than multiple processes.
4.	In multiple process implementations each process executes same code but has its own memory and file resources.	All threads can share same set of open files.
5.	If one server process is blocked no other server process can execute until the first process unblocked.	While one server thread is blocked and waiting, second thread in the same task could run.
6.	In multiple processes each process operates independently of others.	One thread can read, write or even completely wipeout another threads stack.

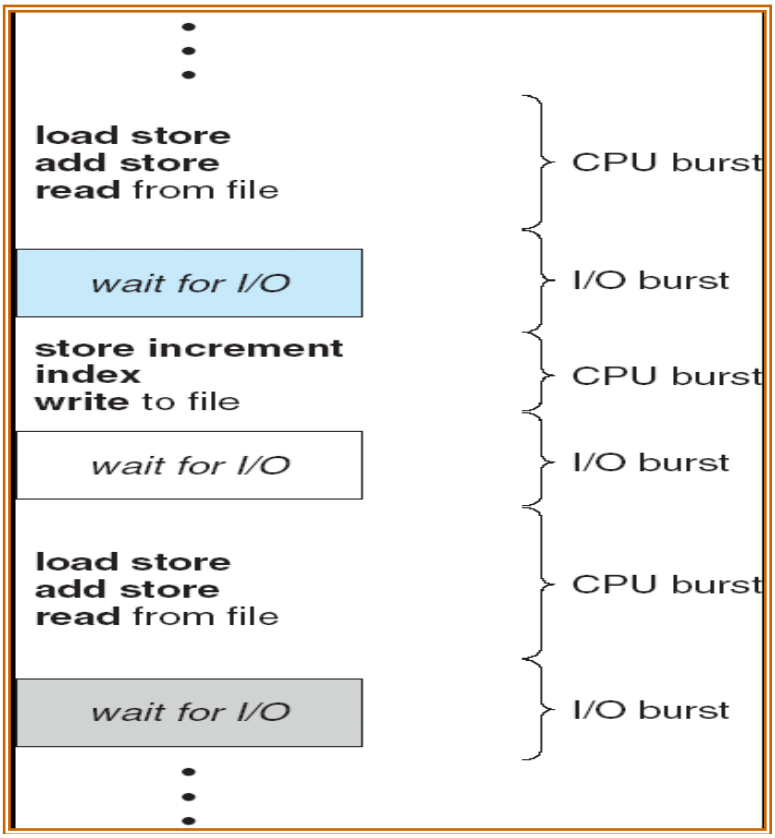
PROCESS SHEDULING

2.5 Basic Concepts

- ✓ In a single-processor system, only one process can run at a time and others must wait until the CPU is free and can be rescheduled.
- ✓ The objective of **multiprogramming** is to have some process running at all times, to **maximize CPU utilization**.
- ✓ With multiprogramming, several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.
- ✓ The CPU is one of the primary computer resources. Thus, its scheduling is central to
- ✓ operating-system design.

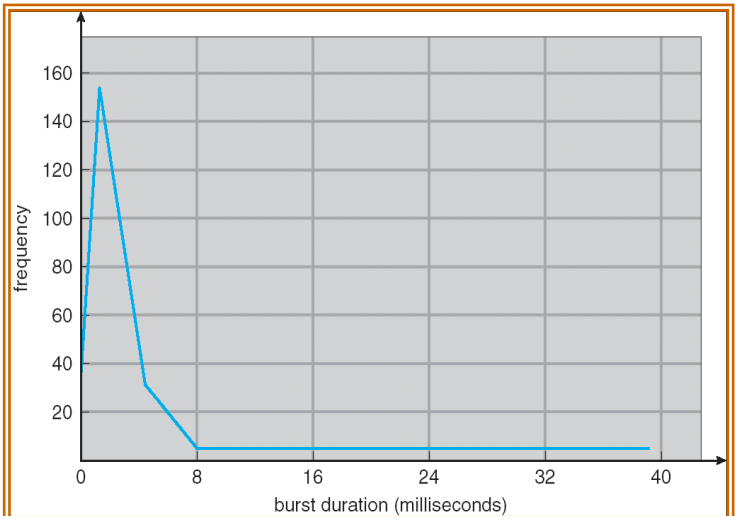
- **CPU–I/O Burst Cycle**

- ✓ Process execution consists of a **cycle** of CPU execution and I/O wait
- ✓ Process execution starts with **CPU burst** and this is followed by **I/O burst** as shown in below **figure.**
- ✓ The final CPU burst ends with a system request to terminate execution.



- ✓ The duration of CPU bursts vary from process to process and from computer to computer.

- ✓ The **frequency curve** is as shown below**figure.**



- **CPU Scheduler**

  - ✓ Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. This selection is carried out by the **short-term scheduler (or CPU scheduler)**.
  - ✓ The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.
  - ✓ A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. But all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The **records** in the queues are **Process Control Blocks (PCBs)** of the processes.

- **Preemptive scheduling**

  - ✓ **CPU-scheduling decisions** may take place under the following **four circumstances**.
    1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes)
    2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
    3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
    4. When a process terminates

  - ✓ When scheduling takes place only under circumstances **1 and 4**, we say that the scheduling scheme is **nonpreemptive** or **cooperative**; otherwise, it is **preemptive**.
  - ✓ Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. A scheduling algorithm is preemptive if, once a process has been given the CPU and it can be taken away.

- **Dispatcher**

  - ✓ Another component involved in the CPU-scheduling function is the dispatcher.
  - ✓ The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
  - ✓ This function involves the following:
    - Switching context
    - Switching to user mode
    - Jumping to the proper location in the user program to restart that program

  - ✓ The dispatcher should be as fast as possible, since it is invoked during every process switch.
  - ✓ The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency.**

2.6 Scheduling Criteria

- ✓ Many criteria have been suggested for comparing CPU scheduling algorithms. The **criteria** include the following:

  - **CPU utilization:** CPU must be kept as busy as possible.CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
  - **Throughput**: If the CPU is busy executing processes, then work is being done. One **measure of work** is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
  - **Turnaround time**: The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
  - **Waiting time**: The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
  - **Response time:** The measure of the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

2.7 Scheduling Algorithms

CPU Scheduling deals with the problem of **deciding** which of the processes in the ready queue is to be allocated the CPU. Following are some **scheduling algorithms,**

- FCFS Scheduling.
- Round Robin Scheduling.
- SJF Scheduling.
- Priority Scheduling.
- Multilevel Queue Scheduling.
- Multilevel Feedback Queue Scheduling.

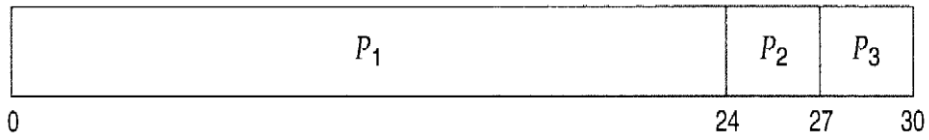
- **First-Come-First-Served (FCFS) Scheduling**

  - ✓ The simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm.
  - ✓ With this scheme, the process that requests the CPU first is allocated the CPU first.
  - ✓ The implementation of the FCFS policy is easily managed with a FIFO queue.
  - ✓ When a process enters the ready queue, its PCB is linked onto the tail of the queue.
  - ✓ When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

- ✓ The average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- ✓ If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following Gantt chart:



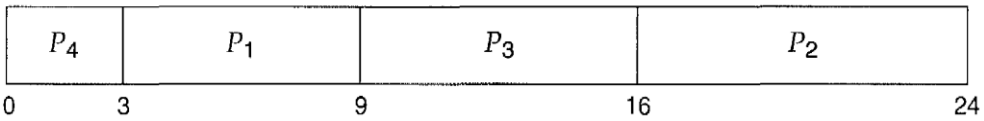
- ✓ The waiting time is 0 milliseconds for process P_1, 24 milliseconds for process P_2, and 27 milliseconds for process P_3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.
- ✓ The FCFS scheduling algorithm is **nonpreemptive**. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

- **Shortest-Job-First Scheduling**

- ✓ This algorithm associates with each process the length of the process's next CPU burst.
- ✓ When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- ✓ If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- ✓ As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- ✓ Using SJF scheduling, we would schedule these processes according to the following Gantt chart:

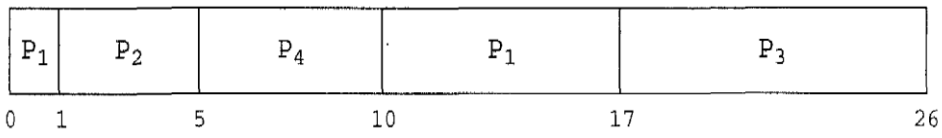


- ✓ The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2, 9 milliseconds for process P_3, and 0 milliseconds for process P_4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds.

- ✓ The SJF scheduling algorithm is **optimal**,it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one, decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.
- ✓ The SJF algorithm can be either **preemptive or nonpreemptive**. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first scheduling.**
- ✓ As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- ✓ If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



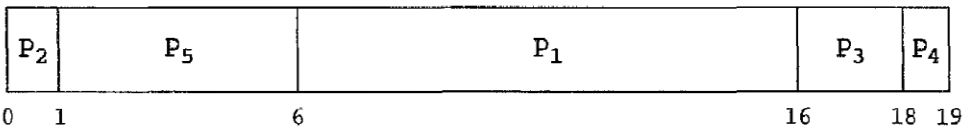
- ✓ Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1.
- ✓ The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is ((10 -1) + (1-1) + (17 -2) + (5- 3))/4 = 26/4 = 6.5 milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

- **Priority Scheduling**

- ✓ The SJF algorithm is a special case of the general priority scheduling algorithm.
- ✓ A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- ✓ Equal-priority processes are scheduled in FCFS order.
- ✓ An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
- ✓ As an example, consider the following set of processes, assumed to have arrived at time 0, in the order P_1, P_2, … , P_5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- ✓ Using priority scheduling, we would schedule these processes according to the following Gantt chart:



- ✓ The average waiting time is 8.2 milliseconds.
- ✓ Priority scheduling can be either **preemptive or nonpreemptive**. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
- ✓ A **major problem** with priority scheduling algorithms is **indefinite blocking, or starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low- priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
- ✓ A **solution** to the problem of indefinite blockage of low-priority processes is **aging**. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

- **Round-Robin Scheduling**

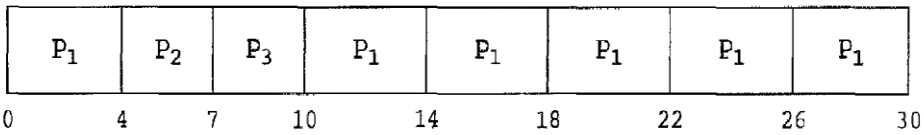
- ✓ The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- ✓ It is similar to FCFS scheduling, but preemption is added to switch between processes.
- ✓ A small unit of time, called a **time quantum or time slice**, is defined.A time quantum is generally from 10 to 100 milliseconds.
- ✓ The ready queue is treated as a circular queue.The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- ✓ To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.
- ✓ The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.One of two things will then happen.
  - o The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
  - o Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the

ready queue. The CPU scheduler will then select the next process in the ready queue.

- ✓ The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

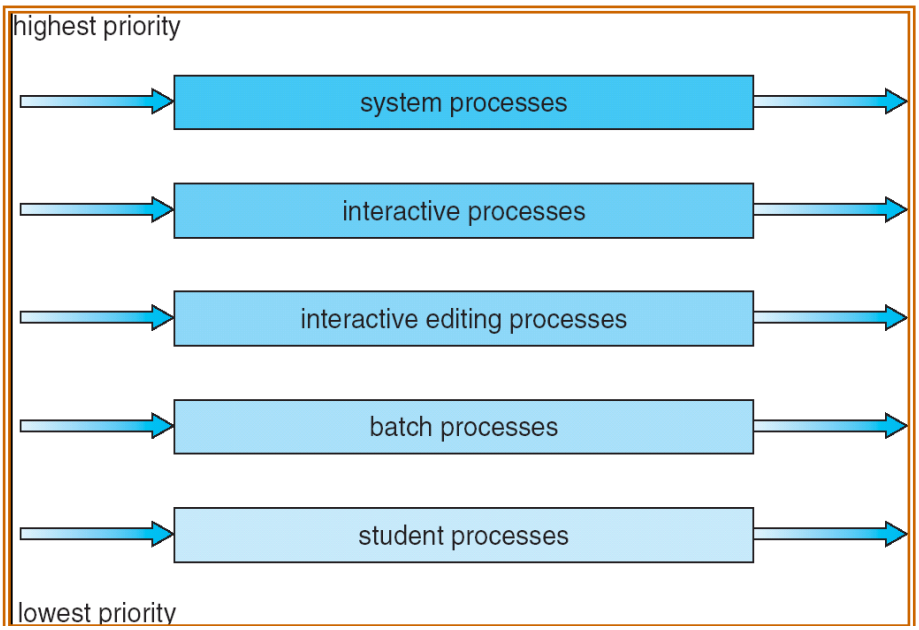
- ✓ If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue ie.process P_2. Since process P_2 does not need 4 milliseconds, it quits before its time quantum expires.
- ✓ The CPU is then given to the next process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is



- ✓ The average waiting time is 17/3 = 5.66 milliseconds.
- ✓ The RR scheduling algorithm is thus preemptive.
- ✓ If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units. Each process must wait no longer than (n-1) * q time units until its next time quantum. **For example**, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

- **Multilevel Queue Scheduling**

- ✓ Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between **foreground (interactive) processes and background (batch) processes**.
- ✓ These two types of processes have different **response-time** requirements and may have **different scheduling** needs.
- ✓ Foreground processes have priority over background processes.
- ✓ A multilevel queue scheduling algorithm partitions the ready queue into several separate queues as shown in **figure 5.3**.
- ✓ The processes are permanently assigned to one queue based on some property of the process, such as memory size, process priority, or process type.
- ✓ Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes.
- ✓ The foreground queue might be scheduled by an **RR algorithm**, while the background queue is scheduled by an **FCFS algorithm**.

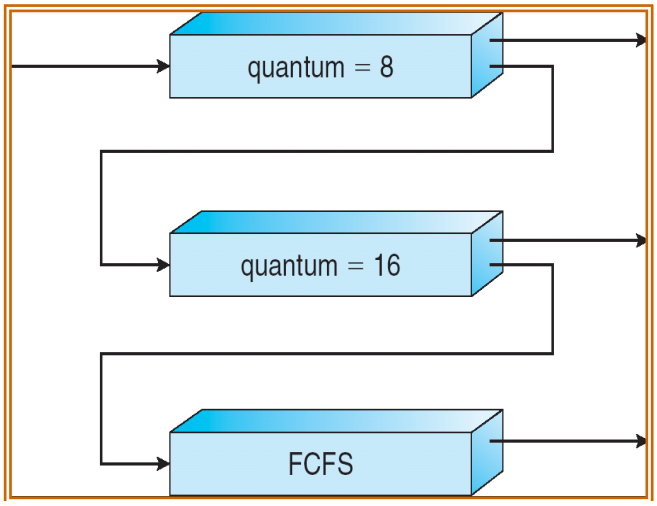


- ✓ There must be scheduling among the queues, which is commonly implemented as **fixed-priority preemptive** scheduling. For example, the foreground queue may have absolute priority over the background queue.
- ✓ An example of a multilevel queue scheduling algorithm with **five queues**, listed below in order of priority:
  1. System processes
  2. Interactive processes
  3. Interactive editing processes
  4. Batch processes
  5. Student processes
- ✓ Each queue has absolute priority over lower-priority queues. No process in the batch queue could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.
- ✓ If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.
- ✓ Another possibility is to **time-slice** among the queues. So, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For example, the foreground queue can be given **80 percent** of the CPU time for RR scheduling among its processes, whereas the background queue receives **20 percent** of the CPU to give to its processes on an FCFS basis.

- **Multilevel Feedback-Queue Scheduling**

- ✓ When the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system.
- ✓ The multilevel feedback-queue scheduling algorithm allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts.
- ✓ If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- ✓ A process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

- ✓ For example,consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 as shown in below **figure5.4.**
- ✓ The scheduler first executes all processes in queue 0. Only when queue 0 is empty it will execute processes in queue 1.
- ✓ Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2.
- ✓ A process in queue 1 will in turn be preempted by a process arriving for queue 0.



- ✓ A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are scheduled on an FCFS basis but they run only when queue 0 and 1 are empty.
- ✓ This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.
- ✓ A multilevel feedback-queue scheduler is defined by the following **parameters:**
  - The number of queues.
  - The scheduling algorithm for each queue.
  - The method used to determine when to upgrade a process to a higher-priority queue.
  - The method used to determine when to demote a process to a lower-priority queue.
  - The method used to determine which queue a process will enter when that processneeds service.

2.8 Multiple-Processor Scheduling

- **Approaches to Multiple-Processor Scheduling**

- ✓ One approach to CPU scheduling in a multiprocessor system is where all scheduling decisions, I/O processing, and other system activities are handled by a single processor i.e., **the master server**. The other processors execute only user code. This **asymmetric multiprocessing** is simple because only one processor accesses the system data structures, reducing the need for data sharing.
- ✓ A second approach uses **symmetric multiprocessing (SMP),** where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.

- Some of the **issues** related to SMP are,

a. Processor Affinity

- ✓ The data most recently accessed by the process is populated in the **cache** for the processor and successive memory accesses by the process are often satisfied in cache memory.
- ✓ If the process **migrates** to another processor, the contents of cache memory must be invalidated for the processor being **migrated from**, and the cache for the processor being **migrated to** must be re-populated. Because of the **high cost** of invalidating and re-populating caches, most SMP systems try to avoid migration of processes from one processor to another and instead tries to keep a process running on the same processor. This is known as **processor affinity**, i.e., a process has an affinity for the processor on which it is currently running.
- ✓ Processor affinity takes **several forms**. When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing that it will do so, a situation is known as **soft affinity**. Here, it is possible for a process to migrate between processors.
- ✓ Some systems such as Linux provide system calls that support **hard affinity**, thereby allowing a process to specify that it must not migrate to other processors.

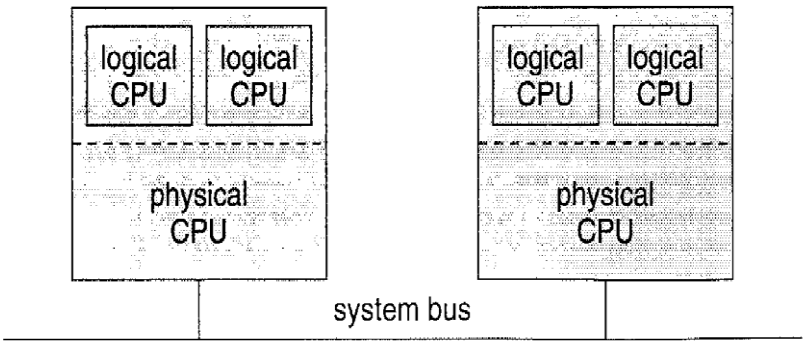
b. Load Balancing

- ✓ On SMP systems, it is important to keep the workload balanced among all processors to utilize the benefits of having more than one processor. Otherwise, one or more processors may sit idle while other processors have high workloads along with lists of processes awaiting the CPU.
- ✓ Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.
- ✓ There are two general approaches to load balancing: **push migration** and **pull migration**.
- ✓ With push migration, a specific task periodically checks the load on each processor and if it finds an imbalance it evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.
- ✓ Pull migration occurs when an idle processor pulls a waiting task from a busy processor.

c. Symmetric Multithreading

- ✓ SMP systems allow several threads to run concurrently by providing multiple physical processors.

- ✓ An alternative strategy is to provide **multiple logical processors** rather than physical processors. Such a strategy is known as symmetric multithreading (or SMT).
- ✓ The idea behind SMT is to create multiple logical processors on the same physical processor, presenting a view of several logical processors to the operating system, even on a system with only a single physical processor.
- ✓ Each logical processor has its own architecture state, which includes general-purpose and machine-state registers and is responsible for its own interrupt handling, meaning that interrupts are delivered to and handled by logical processors rather than physical ones. Otherwise, each logical processor shares the resources of its physical processor, such as cache memory and buses.
- ✓ The following **figure5.5** illustrates a typical **SMT architecture** with two physical processors, each housing two logical processors. From the operating system's perspective, four processors are available for work on this system.



2.9 Thread Scheduling

- ✓ On operating systems that support user-level and kernel-level threads, thekernel-level threads are being scheduled by the operating system.
- ✓ User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).
- ✓ One **distinction** between user-level and kernel-level threads lies in how they are **scheduled.**On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP, a scheme known as **process-contention scope (PCS),** since competition for the CPU takes place among threads belonging to the same process.
- ✓ To decide which kernel thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS).** Competition for the CPU with SCS scheduling takes place among all threads in the system.
- ✓ PCS is done according to priority. The scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer. PCS will preempt the currently running thread in favour of a higher-priority thread.

- **Pthread Scheduling**

  - ✓ Pthreads identifies the following contention scope values:

    - o PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.

- PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

✓ On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs.
✓ The number of LWPs is maintained by the thread library using scheduler activations. The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.
✓ The Pthread IPC provides the following **two functions** for getting and setting the contention scope policy;

- pthread_attr_setscope (pthread_attr_t *attr, int scope)
- pthread_attr_getscope (pthread_attr_t *attr, int *scope)

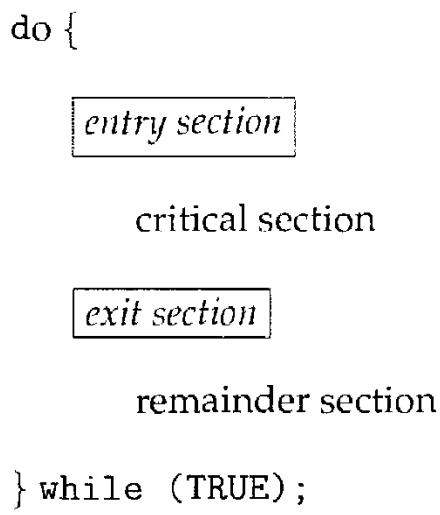
✓ The first parameter for both functions contains a pointer to the attribute set for the thread.
✓ The second parameter for the pthread_attr_setscope () function is passed either the THREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS value, indicating how the contention scope is to be set. In the case of pthread_attr_getscope(), this second parameter contains a pointer to an int value that is set to the current value of the contention scope. If an error occurs, each of these functions returns non-zero values.

2.10 Synchronization: Background

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against race condition, we require that the processes must be synchronized in some way.

2.11 The critical-section problem

✓ Consider a system consisting of n processes $\{P_0, P_1 ... P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be changing the common variables, updating a table, writing a file, and so on.
✓ The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.
✓ The **critical-section problem** is to design a protocol that the processes can use to cooperate.
✓ Each process must request permission to enter its **critical section**. The section of code implementing this request is the **entry section**.
✓ The critical section may be followed by an **exit section**.
✓ The remaining code is the **remainder section**.
✓ The **general structure** of a typical process P_i, is shown in the following figure



- ✓ A **solution** to the critical-section problem must satisfy the following **three** requirements:
  1. **Mutual exclusion.** If process P, is executing in its critical section, then no other processes can be executing in their critical sections.
  2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
  3. **Bounded waiting.** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
- ✓ We assume that each process is executing at a **nonzero** speed.We can make no assumption concerning the relative speed of the n processes.
- ✓ **Two general approaches** are used to **handle** critical sections in operating systems.
  1. Pre-emptive kernel 2. Nonpreemptive kernel.
- ✓ A **pre-emptive kernel** allows a process to be pre-empted while it is running in kernel mode. It is **difficult to design** for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors. It is more **suitable** for **real-time programming**, as it will allow a real-time process to pre-empt a process currently running in the kernel. Also, pre-emptive kernel may be **more responsive**, since there is less risk that a kernel-mode process will run for an arbitrarily long period before giving up the processor to waiting processes.
- ✓ A **Nonpreemptive kernel** does not allow a process running in kernel mode to be pre-empted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. It is essentially free from **race conditions** on kernel data structures, as only one process is active in the kernel at a time.

2.12 Peterson's solution

- ✓ A classic **software-based solution** to the critical-section problem is known as **Peterson's solution.**
- ✓ Peterson's solution is restricted to **two processes** that alternate execution between their critical sections and remainder sections.
- ✓ The processes are numbered P_0 and P_1 **or** P_i and P_j where j=1-i.
- ✓ Peterson's solution requires **two data items** to be shared between the two processes,

```
int turn;
boolean flag[2];
```

- ✓ The variable **turn** indicates whose turn it is to enter its critical section. That is, if turn == i, then process P_i, is allowed to execute in its critical section.
- ✓ The **flag array** is used to indicate if a process is ready to enter its critical section. **For example**, if flag[i] is true, this value indicates that P_i is ready to enter its critical section.
- ✓ To enter the critical section, process P_i first sets **flag[i]** to be true and then sets **turn** to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- ✓ If both processes try to enter at the same time, **turn** will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.
- ✓ The eventual value of turn decides which of the two processes is allowed to enter its critical section first.
- ✓ The following algorithm describes the structure of P_i in Peterson's solution.

```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

- ✓ To prove that this solution is correct, we need to show that,
  1. Mutual exclusion is preserved.
  2. The progress requirement is satisfied.
  3. The bounded-waiting requirement is met.

- ✓ To prove **property 1**, we note that each P_i enters its critical section only if **either flag[j] = = false or turn = = i.** For P_0 to enter, turn must be equal to 0 and for P_1 to enter, turn must be equal to 1 because flag[0] = = flag[1] = = true. Since the value of turn can be either 0 or 1 but cannot be both, hence P_0 and P_1 cannot enter into critical section simultaneously.
- ✓ To prove **properties 2 and 3**, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] = = true and turn = = j. If P_j is not ready to enter the critical section, then flag

[j] = = false, and P_i can enter its critical section. If turn = = j, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset flag[j] to false, allowing P_i to enter its critical section. If P_j resets flag[j] to true, it must also set turn to i. Thus, P_i will enter the critical section (progress) after at most one entry by P_i (bounded waiting).

2.13 Synchronization hardware

- ✓ Any solution to the critical-section problem requires a simple tool called **a lock**. **Race conditions** are prevented by requiring that critical regions be protected by locks.
- ✓ That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section. This is illustrated below,

```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (TRUE);
```

- ✓ Hardware features can make any programming task easier and improve **system efficiency.**
- ✓ The critical-section problem can be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.
- ✓ This solution is **not feasible** in a multiprocessor environment. Disabling interrupts on a multiprocessor can be **time consuming**, as the message is passed to all the processors. This message passing delays entry into each critical section, and **system efficiency decreases.**
- ✓ Many modern computer systems therefore provide special hardware instructions that allow us either to **test** and **modify** the content of a word or to **swap** the contents of two words **atomically** that is, as one uninterruptible unit.
- ✓ We can use these special instructions to solve the critical-section problem in a relatively simple manner.TheTestAndSet() instruction can be defined as below,

```
booleanTestAndSet(boolean *target)
  {
```

```
    booleanrv = *target;
  *target = TRUE;
returnrv;
}
```

- ✓ The important characteristic is that this instruction is executed atomically. Thus, if two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
- ✓ The structure of process P_i is shown below,

```
do {
    while (TestAndSetLock(&lock))
    ; // do nothing

    // critical section

    lock = FALSE;

    // remainder section
    }while (TRUE);
```

- ✓ The Swap() instruction, in contrast to the TestAndSet() instruction, operates on the contents of two words as shown below,

```
void Swap(boolean *a, boolean *b)
        {
            boolean temp = *a;
                *a = *b;
                *b = temp;
        }
```

- ✓ It is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows.
- ✓ A global Boolean variable **lock** is declared and is initialized to false. In addition, each process has a local Boolean variable **key**. The structure of process P_i is shown below,

```
do {
    key = TRUE;
    while (key = = TRUE)
        Swap(&lock, &key
    };

        // critical section

    lock = FALSE;
```

```
                    // remainder section
            }while (TRUE);
```

- ✓ Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.
- ✓ Another algorithm is given below using the TestAndSet() instruction that satisfies all the critical-section requirements. The common **data structures** are,

```
boolean waiting[n];
        boolean lock;
```

- ✓ These data structures are initialized to **false**.

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key= TestAndSet(&lock);
        waiting[i] = FALSE;

                // critical section

        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;

        if (j = = i)
                lock = FALSE;
        else
                waiting[j] = FALSE;

                // remainder section
        } while (TRUE);
```

2.14 Semaphores

- ✓ The **hardware-based solutions** to the critical-section problem are **complicated** for application programmers to use.
- ✓ To **overcome this difficulty**, we can use a **synchronization tool** called a**semaphore.**
- ✓ A **semaphore S** is an integer variable it is accessed only through **two standard atomic operations:wait()** and **signal()**. The wait() operation was termed as **P**; signal() was called **V**. The definition of **wait()** is as follows,

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

✓ The definition of **signal()** is as follows,

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

✓ The wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

- **Usage**

  ✓ Operating systems often distinguish between **counting and binary** semaphores.
  ✓ The value of a **counting semaphore** can range over an unrestricted domain.
  ✓ The value of a **binary semaphore** can range only between 0 and 1.
  ✓ Binary semaphores are known as **mutex locks**, as they are locks that provide mutual exclusion.
  ✓ We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a semaphore, **mutex**, initialized to 1. Each process P_i is organized as shown,

```
do {
waiting(mutex);

        // critical section

signal (mutex) ,

        // remainder section
}while (TRUE);
```

✓ **Counting semaphores** can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When count=0, all resources are being used. Then the processes that wish to use a resource will block until the count becomes greater than 0.
✓ We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2. Suppose we require that P_2 be executed only after S_1 has completed. We

can implement this by letting P_1 and $P_2$to share a common semaphore **synch**, initialized to 0, and by inserting the statements

S_1;
signal(synch);

in process P_1, and the statements

wait(synch);
S_2;

in process P_2. Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal (synch), which is after statement S_1 has been executed.

- **Implementation**

  - ✓ The main **disadvantage** of the semaphore definition given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
  - ✓ This type of semaphore is also called a **spin lock** because the process "spins" while waiting for the lock.
  - ✓ To overcome the need for busy waiting, we can **modify** the definition of the **wait() and signal()** semaphore operations. When a process executes the wait() operation and finds that the semaphore value is not positive, then rather than engaging in busy waiting, the process can **block**itself. The block operation places a process into a **waiting queue** associated with the semaphore, and the state of the process is switched to the **waiting state.**
  - ✓ A process that is blocked, waiting on a semaphore S, should be **restarted** when some other process executes a signal() operation. The process is restarted by a **wakeup()** operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. To implement semaphores under this definition, we define a semaphore as a "C' struct:

```
typedefstruct {
        int value;
        struct process *list;
} semaphore;
```

  - ✓ Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process. The wait() semaphore operation can now be defined as,

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

The signal () semaphore operation can now be defined as,

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from< S->list;
                wakeup(P);
        }
}
```

- ✓ The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.
- ✓ This implementation, semaphore values may be negative, if a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.
- ✓ The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs.

- **Deadlocks and Starvation**

  - ✓ The implementation of a semaphore with a waiting queue may result in a **deadlock** situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. When such a state is reached, these processes are said to be **deadlocked.**
  - ✓ To illustrate this, we consider a system consisting of two processes, $P_0$and P_1, each accessing two semaphores, S and Q, set to the value 1:

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

  - ✓ Suppose that P_0executes wait(S) and then P_1 executes wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly, when P_1 executes wait(S), it must wait until P_0executes signal(S). Since these signal() operations cannot be executed, $P_0$and P_1 are deadlocked.
  - ✓ Another problem related to deadlocks is **indefinite blocking, or starvation,** a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

2.15 Classic problems of Synchronization

- Bounded-buffer problem
- The Readers-Writers Problem
- The Dining-Philosophers Problem

- **The Bounded-Buffer Problem**

  - ✓ We assume that the pool consists of **n** buffers, each capable of holding one item.
  - ✓ The **mutex** semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
  - ✓ The **empty** and **full** semaphores count the number of empty and full buffers.
  - ✓ The semaphore **empty** is initialized to the value n; the semaphore **full** is initialized to the value 0.
  - ✓ The code for the producer process is shown:

```
do{
// produce an item in nextp
        ...
wait(empty);
wait (mutex);
        ...
// add nextp to buffer
        ...
signal(mutex);
signal (full);
}while (TRUE);
```

  - ✓ The code for the consumer process is shown:

```
do {
wait (full);
wait(mutex);
...
//remove an item from buffer to nextc
...
signal(mutex);
signal(empty);
//consume the item in nextc
...
}while (TRUE);
```

    - ✓ We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

- **The Readers-Writers Problem**

  - ✓ A database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
  - ✓ **Readers** - processes that only read the database.
  - ✓ **Writers** - processes performing both read and write (update).
  - ✓ **Problem**: If two readers access the shared data simultaneously, no problem will result. But if a writer and some other thread (either a reader or a writer) access the database simultaneously, problem arises.

- ✓ This synchronization problem is referred to as the readers-writers problem.
- ✓ The readers-writers problem has several variations.
- ✓ The **first**readers-writers problem, which requires that no reader must be kept waiting unless a writer has already obtained permission to use the shared object.
- ✓ The **second**readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible.
- ✓ A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.
- ✓ In the solution to the first readers-writers problem, the reader processes share the following data structures:

```
semaphoremutex, wrt;
intreadcount;
```

- ✓ The semaphores **mutex** and **wrt** are initialized to 1and **readcount** is initialized to 0.
- ✓ The semaphore wrt is common to both reader and writer processes.
- ✓ The **mutex** semaphore is used to ensure **mutual exclusion** when the variable **readcount** is updated.
- ✓ The **readcount** variable keeps track of how many processes are currently reading the object.
- ✓ The semaphore **wrt** functions as a mutual-exclusion semaphore for the writers.

- ✓ The code for a writer process is,

```
do {
wait(wrt);
…..
//writing is performed
signal(wrt);
} while (TRUE);
```

- ✓ The code for a reader process is shown,

```
do {
wait (mutex);
readcount++;
if (readcount = = 1)
        wait (wrt);
signal(mutex);
    ……..
  //reading is performed
    ……..
wait(mutex);
readcount--;
if (readcount= = 0)
        signal(wrt);
signal(mutex);
} while (TRUE);
```

- ✓ If a writer is in the critical section and n readers are waiting, then one reader is queued on **wrt**, and n-1 readers are queued on **mutex**.
- ✓ Also, when a writer executes **signal(wrt)**, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

- **The Dining-Philosophers Problem**

- ✓ Consider **five** philosophers who spend their lives thinking and eating.
- ✓ The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- ✓ In the centre of the table is a bowl of rice, and the table is laid with five single chopsticks (below figure).
- ✓ When a philosopher is thinking, she does not interact with her colleagues.
- ✓ When a philosopher gets hungry, she tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- ✓ When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

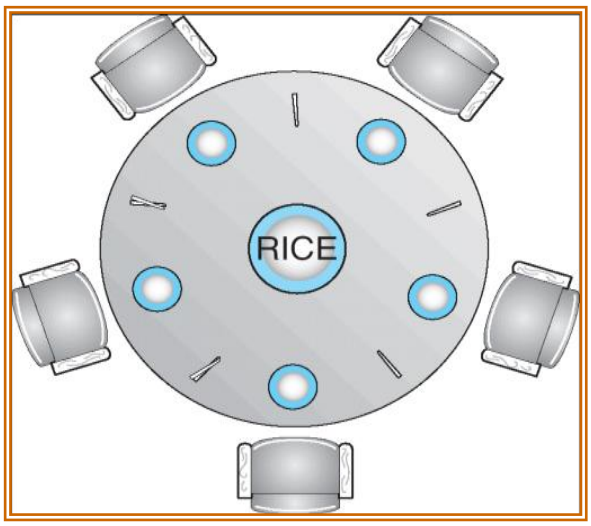


figure 6.1:The situation of the dining philosophers

- ✓ One simple solution is to represent each chopstick with a semaphore.
- ✓ A philosopher tries to **grab** a chopstick by executing a wait() operation on that semaphore; she **releases** her chopsticks by executing the **signal()** operation on the appropriate semaphores.
- ✓ Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1.

- ✓ The structure of philosopher i is shown below,

```
do {
wait(chopstick[i]);
k[(i+l) % 5]);
```

```
……
//eat
.....

signal(chopstick[i]);
signal(chopstick[(i+l) % 5]);
……
//think
.....
} while (TRUE);
```

- ✓ The disadvantage is it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.
- ✓ Several possible remedies to the deadlock problem are available.
  - o Allow at most four philosophers to be sitting simultaneously at the table.
  - o Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
  - o Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

2.16 Monitors

- ✓ All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.
- ✓ This may result in various difficulties.
  - o Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
……
critical section
……
wait(mutex);
```

- ✓ In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections.
  - o Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes

```
wait(mutex);
……
critical section
……
wait(mutex);
```

In this case, a deadlock will occur.

- Suppose that a process omits the wait (mutex), or the signal (mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

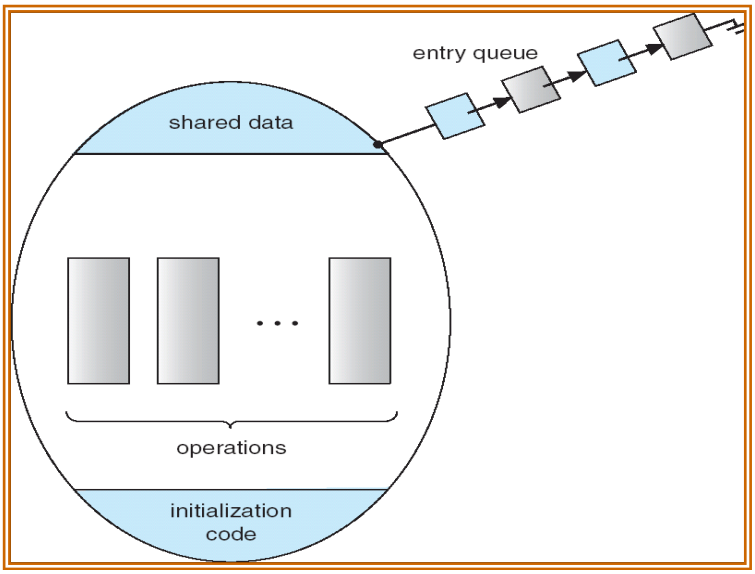
✓ These **examples** illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem. To deal with such errors, one fundamental high-level synchronization construct-the **monitor** is used.

- **Usage**

✓ The syntax of a monitor is shown below,

```
monitormonitor name
{
    //shared variable declarations
        procedure P1 ( . . . ) {
        ……
        }
        procedure P2 ( . . . ) {
        …….
        }
                .
                .
                .
        procedurePn ( . . . ) {
        ……….
        }
        initialization code ( . . . ) {
        ……….
        }
}
```

✓ The monitor construct ensures that only one process at a time is active within the monitor. The programmer does not need to code this synchronization constraint explicitly.

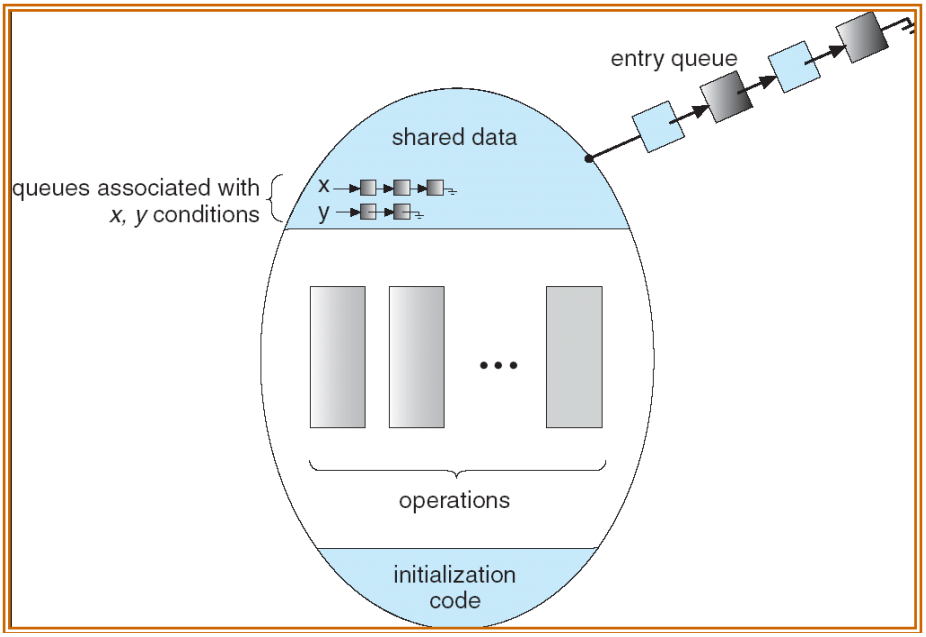


Schematic view of a monitor

- ✓ The monitor construct, as defined so far is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the **condition** construct. A programmer can define one or more variables of the type **condition**,

condition x, y;

- ✓ The only operations that can be invoked on a condition variable are wait() and signal().
- ✓ The operation x.wait() means that the process invoking this operation is suspended until another process invokes x.signal();
- ✓ The x. signal() operation resumes exactly one suspended process.
- ✓ If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation has never been executed. It is shown in below figure.



Monitor with condition variables

- ✓ Suppose when the x. signal () operation is invoked by a process P, there exists a suspended process Q associated with condition x. If the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. However both processes can continue with their execution. Two possibilities exist,
  - **Signal and wait.** *P* either waits until *Q* leaves the monitor or waits for another condition.
  - **Signal and continue.** *Q* either waits until *P* leaves the monitor or waits for another condition.

- **Dining-Philosophers Solution Using Monitors**

  - ✓ It is a deadlock-free solution to the dining-philosophers problem.
  - ✓ This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
  - ✓ We use the following data structure to distinguish among three states in which we may find a philosopher.

    enum {thinking, hungry, eating}state [5];

  - ✓ Philosopher i can set the variable **state[i] = eating** only if her two neighbors are not eating.
  - ✓ We also need to declare

    condition self [5];

    where philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs.
  - ✓ The distribution of the chopsticks is controlled by the monitor **dp**, whose definition is shown below.

```
monitordp
{
enum {THINKING, HUNGRY, EATING} state[5];
conditionself[5];

void pickup(int i) {
state[i] =HUNGRY;
test(i);
if (state [i] ! = EATING)
        self [i] . wait() ;
}

void putdown(int i) {
state[i] =THINKING;
test((i + 4) % 5);
test((i + 1) % 5);
}

void test(int i) {
if ((state[(i + 4) % 5] !=EATING) &&
        (state[i] ==HUNGRY) &&
```

```
                    (state[(i + 1) % 5] !=EATING)) {
                        state[i] =EATING;
                        self[i] .signal();
                    }
                }

                initialization_code() {
                for (int i = 0; i < 5; i++)
                state[i] =THINKING;
                }
            }
```

- ✓ Each philosopher, before starting to eat, must invoke the operation pickup().After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown() operation. Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence,

```
dp.pickup(i);
...
eat
...
dp.putdown(i);
```

- **Implementing a Monitor Using Semaphores**

  - ✓ For each monitor, a semaphore **mutex** (initialized to 1) is provided.
  - ✓ A process must execute **wait(mutex)** before entering the monitor and must execute **signal(mutex)** after leaving the monitor.
  - ✓ Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, **next** (initialized to 0) is introduced, on which the signaling processes may suspend themselves.
  - ✓ An integer variable **next_count** is also provided to count the number of processes suspended on **next**. Thus, each external procedure **F** is replaced by ,

```
wait(mutex) ;
        ...
        body of F
        ...
if (next_count> 0)
        signal(next);
else
        signal(mutex);
```

Mutual exclusion within a monitor is ensured.

  - ✓ Condition variables are implemented as follows. For each condition x, semaphore**x_sem** and an integer variable **x_count**, both initialized to 0 are introduced.
  - ✓ The operation x.wait() can now be implemented as

```
x_count++;
if (next_count> 0)
        signal(next);
```

```
        else
                signal(mutex);
        wait(x_sem);
        x_count--;
```

- ✓ The operation x.signal() can be implemented as

```
        if (x_count> 0) {
        next_count++;
        signal(x_sem);
        wait(next) ;
        next_count--;
        }
```

- **Resuming Processes within a Monitor**

  - ✓ If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then the problem is how to determine which of the suspended processes should be resumed next.
  - ✓ One simple solution is to use an FCFS ordering, so that the process that has been waiting for the longest time is resumed first.
  - ✓ In many circumstances, such simple scheduling scheme is not adequate. For this purpose, the **conditional-wait** construct can be used; it has the form

    x.wait(c);

    where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a **priority number** is then stored with the name of the process that is suspended.
  - ✓ When x.signal() is executed, the process with the smallest priority number is resumed next.
  - ✓ To illustrate this, consider the **ResourceAllocator** monitor shown below, which controls the **allocation of a single resource** among competing processes.

```
monitorResourceAllocator
{
        boolean busy;
        condition x;

        void acquire(int time) {
        if (busy)
                x.wait(time);
        busy = TRUE;
        }

        void release() {
        busy = FALSE;
        x. signal() ;
        }

        initialization_code() {
        busy = FALSE;
        }
}
```

- ✓ Each process, when requesting an allocation of this resource, it specifies the maximum time it plans to use the resource.
- ✓ The monitor allocates the resource to the process that has the shortest time-allocation request. A process that needs to access the resource in question must observe the following sequence:

R.acquire(t);

……

access the resource;

……

R. release() ;

where R is an instance of type ResourceAllocator.

- ✓ The monitor concept cannot guarantee that the preceding access sequence will be observed and the following problems can occur,
  - o A process might access a resource without first gaining access permission to the resource.
  - o A process ntight never release a resource once it has been granted access to the resource.
  - o A process might attempt to release a resource that it never requested.
  - o A process might request the same resource twice (without first releasing the resource).
- ✓ The same difficulties are encountered with the use of semaphores.

---0o0---

MODULE-3

DEADLOCKS

3.1 Deadlocks

- ✓ When processes request resources and if the resources are not available (held by other process) at that time, then process enters into waiting state. This situation is called **deadlock**.

3.2 System Model

- ✓ A system consists of finite number of resources and is distributed among number of processes. A process must **request** a resource before using it and it must **release** the resource after using it. It can request any number of resources to carry out a designated task. The amount of resource requested may not exceed the total number of resources available.
- ✓ A process may utilize the resources in only the following **sequence**,
  1. **Request:** If the request is not granted immediately then the requesting process must wait it can acquire the resources.
  2. **Use:**The process can operate on the resource.
  3. **Release:**The process releases the resource after using it.
- ✓ To illustrate deadlock, consider a system with one printer and one tape drive. If a process Pi currently holds a printer and a process Pj holds the tape drive. If process Pi request a tape drive and process Pj request a printer then a deadlock occurs.
- ✓ Multithread programs are good candidates for deadlock because they compete for shared resources.

3.3 Deadlock Characterization

- **Necessary Conditions**

  A deadlock situation can occur if the following **4 conditions** occur simultaneously in a system.

  - **Mutual Exclusion:** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests for the resource, the requesting process must be delayed until the resource has been released.
  - **Hold and Wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.
  - **No Preemption:** Resources cannot be preempted i.e., only the process holding the resources must release it after the process has completed its task.
  - **Circular Wait:** A set $\{P_0,P_1……..P_n\}$ of waiting process must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, P_{n-1} is waiting for resource held by process P_n and P_n is waiting for the resource held by P_0.

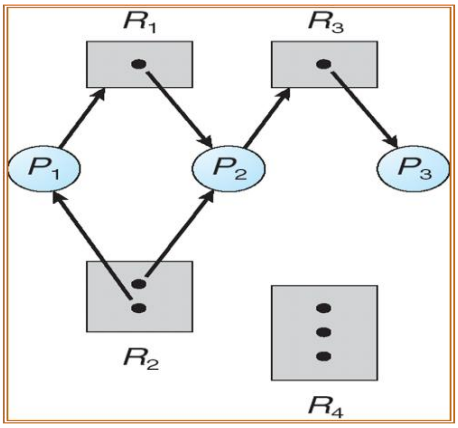
- **Resource Allocation Graph**

  - ✓ Deadlocks are described by using a directed graph called **system resource allocation graph**. The graph consists of set of **vertices (V)** and set of **edges (E)**.
  - ✓ The set of vertices (V) can be described into two different types of nodes,
  - ✓ $P=\{ P_1,P_2……..P_n\}$ a set consisting of all active processes and $R=\{R_1,R_2…….R_n\}$ a set consisting of all resource types in the system.
  - ✓ A directed edge from process P_i to resource type $R_j$is denoted by $P_i \rightarrow R_j$,it indicates that P_i has requested an instance of resource type R_j and is waiting for that resource. This edge is called **Request edge.**
  - ✓ A directed edge $R_j \rightarrow P_i$ signifies that an instance of resource type R_j has been allocated to process P_i. This is called **Assignment edge.**
  - ✓ Process $P_i$is represented as **circle** and each resource type $R_j$as a **rectangle**.
  - ✓ Since resource type $R_j$may have more than one instance, we represent each such instance as a **dot** within the rectangle.
  - ✓ Request edge points to only the rectangle R_j, whereas an assignment edge must also designate one of the dots in the rectangle.
  - ✓ The below **figure** shows the Resource allocation graph which denotes,
  - ✓ The sets P, R and E:

    $P = \{P_1, P_2, P_3\}$

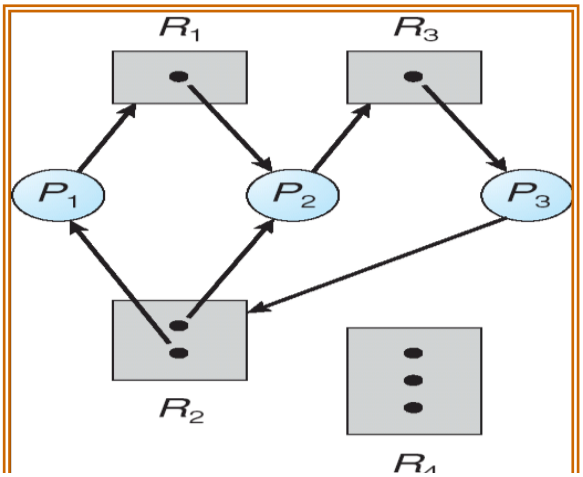
    $R = \{R_1, R_2, R_3, R_4\}$

    $E = \{ P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
  - ✓ Resource instances:
    - One instance of resource type R_1
    - Two instances of resource type R_2
    - One instance of resource type R_3
    - Three instances of resource type R_4
  - ✓ Process states:
    - Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
    - Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
    - Process $P_3$is holding an instance of R_3.

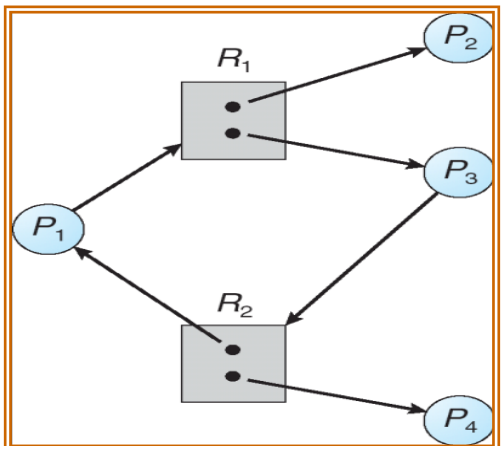


  - ✓ If the graph contains no cycle, then no process in the system is deadlocked. If the graph contains a cycle then a deadlock may exist.
  - ✓ If each resource type has exactly one instance than a cycle implies that a deadlock has occurred.

- ✓ If each resource type has several instances then a cycle do not necessarily implies that a deadlock has occurred.
- ✓ Consider the resource-allocation graph shown in above **figure**.
- ✓ Suppose, process P_3requests an instance of resource type R_2. Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph which results in below **figure**.



- ✓ Now, two minimal cycles exist in the system:
  - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
  - $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
- ✓ Processes P_1, P_2, and $P_3$are deadlocked.
- ✓ Process $P_2$is waiting for the resource R_3, which is held by process P_3.
- ✓ Process $P_3$is waiting for either process P_1or process $P_2$to release resource R_2.
- ✓ In addition, process $P_1$is waiting for process $P_2$to release resource R_1.
- ✓ Consider the resource-allocation graph in below **figure**, which also have a cycle $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$,but there is no deadlock.



- ✓ P_4may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.
- ✓ If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.

3.4 Methods for Handling Deadlocks

- ✓ There are three ways to deal with deadlock problem
  - We can use a **protocol to prevent or avoid deadlocks**, ensuring that the system will never enter into the deadlock state.
  - We can allow a system to enter into deadlock state, **detect it and recover** from it.
  - We can **ignore** the problem and pretend that the deadlock never occur in the system. This is used by most OS including UNIX.
- ✓ To ensure that the deadlock never occurs, the system can use either deadlock avoidance or deadlock prevention.
- ✓ Deadlock prevention is a set of method for ensuring that at least one of the necessary conditions does not occur.
- ✓ Deadlock avoidance requires the OS is given advance information about which resource a process will request and use during its lifetime.
- ✓ If a system does not use either deadlock avoidance or deadlock prevention then a deadlock situation may occur. In this situation the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from deadlock.
- ✓ Undetected deadlock will result in deterioration of the system performance.

3.5 Deadlock Prevention

- ✓ For a deadlock to occur each of the four necessary conditions must hold. If at least one of these conditions **does not hold** then we can prevent occurrence of deadlock.
  - **Mutual Exclusion:**This holds for non-sharable resources. For ex, A printer can be used by only one process at a time. Mutual exclusion is not possible in sharable resources and thus they cannot be involved in deadlock. Read-only files are good examples for sharable resources.
  - **Hold and Wait:**This condition can be eliminated by forcing a process to release all its resources held by it when it requests a resource. Two possible **solutions(protocols)** to achieve this are,
    - One protocol can be used is that each process is allocated with all of its resources before it starts execution.
    - Another protocol that can be used is to allow a process to request a resource when the process has none.
    - To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.
    - The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.
    - Both protocols have two main **disadvantages**. First, resource utilization is low, since resources may be allocated but unused for a long period.

- Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

- **No Preemption:**To ensure that this condition never occurs the resources must be preempted. The following protocols can be used.
  - If a process is holding some resource and request another resource that cannot be immediately allocated to it, then all the resources currently held by the requesting process are preempted and added to the list of resources for which other processes may be waiting. The process will be restarted only when it regains the old resources and the new resources that it is requesting.
  - When a process request resources, we check whether they are available or not. If they are available we allocate them else we check that whether they are allocated to some other waiting process. If so we preempt the resources from the waiting process and allocate them to the requesting process. Otherwise, the requesting process must wait.
- **Circular Wait:**One way to ensure that this condition never holds is to impose total ordering of all resource types and each process requests resource in an increasing order.For ex, Let $R=\{R_1,R_2,\ldots\ldots R_m\}$ be the set of resource types. We assign each resource type with a unique integer value. This allows us to compare two resources and determine whether one precedes the other in ordering. We can define a one to one function $F:R \rightarrow N$ as follows,

  F(disk drive)=5, F(printer)=12, F(tape drive)=1

Deadlock can be prevented by using the following protocols.

- Each process can request the resource in **increasing order**. A process can request any number of instances of resource type say R_i and it can request instances of resource type R_j only $F(R_j) > F(R_i)$.
- Alternatively when a process requests an instance of resource type R_j, it has released any resource R_i such that $F(R_i) >= F(R_j)$.

If these two protocols are used then the circular wait cannot hold.

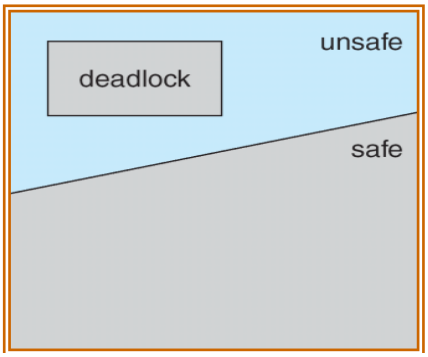
3.6 Deadlock Avoidance

- ✓ Deadlock prevention algorithm may lead to low device utilization and reduces system throughput.
- ✓ Avoiding deadlocks requires additional information about how resources are to be requested. With the knowledge of the complete sequences of requests and releases we can decide for each requests whether the process should waitor not.
- ✓ For each requests it requires checking of the resources **currently available**, resources that are **currently allocated** to each processes, future **requests and release** of each process to decide whether the current requests can be satisfied or must wait to avoid future possible deadlock.
- ✓ A deadlock avoidance algorithm dynamically examines the resources allocation state to ensure that a circular wait condition never exists. The resource allocation state is defined by the number of available and allocated resources and the maximum demand of each process.

- **Safe State**

  - ✓ A state is a **safe state** in which there exists at least one order in which all the process will run completely without resulting in a deadlock.

- ✓ A system is in safe state if there exist a **safe sequence**.
- ✓ A sequence of processes <P_1,P_2,…….P_n> is a safe sequence for the current allocation state if, for each P_i, the resources requests that P_i can still make and it can be satisfied by the currently available resources plus the resources held by all p_j, with j<i.
- ✓ If the resources that P_i requests are not currently available then P_i can wait until all $P_j$have finished. When they have finished, P_i can obtain all of its needed resource to complete its designated task.
- ✓ A safe state is not a deadlocked state. But, a deadlocked state is an unsafe state.
- ✓ Not all unsafe states are deadlocked. An unsafe state maylead to a deadlock. It is shown in below **figure.**



- ✓ **For Ex**, Consider a system with 12 magnetic tape drives and three processes P_o, P_1, and P_2. Process $P_o$requires ten tape drives, process P_1may need as many as four tape drives, and process P_2 may need up to nine tape drives. Suppose that, at time t_o, process $P_o$is holding five tape drives, process P_1 is holding two tape drives, and process P_2 is holding two tape drives. (there are 3 free tape drives.)

	Maximum Needs	Current Needs (allocated)
P_0	10	5
P_1	4	2
P_2	9	2

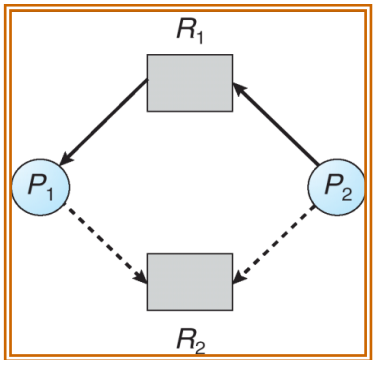
- ✓ At time t_0, the system is in a safe state. The sequence <P_1, P_0, P_2> satisfies the safety condition.
- ✓ A system can go from a safe state to an unsafe state. Suppose that, at time t_1, process P_2 requests and is allocated one more tape drive. The system is no longer in a safe state. Only process P_1 can be allocated all its tape drives.
- ✓ Whenever a process request a resource that is currently available, the system must decide whether resources can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state.

- **Resource Allocation Graph Algorithm**

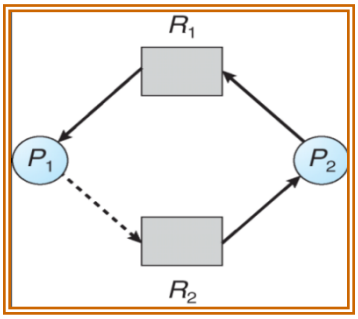
- ✓ This algorithm is used only if we have **one instance** of a resource type. In addition to the request edge and the assignment edge a new edge called **claim**

edge is used.A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request R_j in future. The claim edge is represented by a **dotted line**.

- ✓ When a process P_i requests the resource R_j, the claim edge is converted to a request edge.When resource R_j is released by process P_i, the **assignment edge** $R_j \rightarrow P_i$ is replaced by the claim edge $P_i \rightarrow R_j$.
- ✓ When a process Pi requests resource R_j the request is granted only if converting the request edge $P_i \rightarrow R_j$ to as assignment edge $R_j \rightarrow P_i$ do not result in a cycle.
- ✓ Cycle detection algorithm is used to detect the cycle. If there are no cycles then the allocation of the resource to process leave the system in safe state.
- ✓ To illustrate this algorithm, we consider the resource-allocation graph shown in below **figure**.



- ✓ Suppose that P_2requests R_2 but we cannot allocate it to P_2 even if R_2 is currently free, because this will create a cycle in the graph as shown in **figure**.



- ✓ A cycle indicates that the system is in an unsafe state. If P_1 requests R_2, and P_2 requests R_1, then a deadlock will occur.

- **Banker's Algorithm**

  - ✓ This algorithm is applicable to the system with **multiple instances** of each resource types, but this is less efficient than the resource allocation graph algorithm.
  - ✓ When a new process enters the system it must declare the maximum number of resources that it may need. This number may not exceed the total number of resources in the system. The system must determine that whether the allocation of the resources will leave the system in a safe state or not. If it is so resources are allocated else it should wait until the process release enough resources.
  - ✓ Several **data structures** are used to implement the banker's algorithm. Let 'n' be the number of processes in the system and 'm' be the number of resources types. The following data structures are needed.

- **Available**: A vector of length m indicates the number of available resources. If Available[j]=k, then k instances of resource type R_j is available.
- **Max**: An n*m matrix defines the maximum demand of each process.If Max[i][j]=k, then P_i may request at most k instances of resource type R_j.
- **Allocation**: An n*m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j]=k, then P_i is currently allocated k instances of resource type R_j.
- **Need**: An n*m matrix indicates the remaining resources need of each process. If Need[i][j]=k, then P_i may need k more instances of resource type R_j to complete its task. So Need[i][j]=Max[i][j]-Allocation[i][j].

- **Safety Algorithm**

  - ✓ This algorithm is used to find out whether a system is in safe state or not. The algorithm can be described as follows,

    Step 1. Let Work and Finish be two vectors of length m and n respectively.Initialize work = available and Finish[i]=false for i=1,2,3,……n
    Step 2. Find i such that both
    Finish[i]==false
    $Need_i \leq Work$

    If no such i exists, then go to step 4

    Step 3. Work = Work +Allocation
    Finish[i]=true
    Go to step 2
    Step 4. If Finish[i]==true for all i, the system is in safe state.
  - ✓ This algorithm may require an order of $m*n^2$ operation to decide whether a state is safe.

- **Resource Request Algorithm**

  - ✓ Let $Request_i$ be the request vector of process P_i. If $Request_i[j]==k$, then process P_i wants k instances of the resource type R_j. When a request for resources is made by process P_i the following actions are taken.

    Step 1: If $Request_i \leq Need_i$, go to step 2. Otherwise raise an error condition, since the process has exceeded its maximum claim.
    Step 2: If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since the resources are not available.
    Step 3: The system pretend to have allocated the requested resources to process P_i,then modify the state as follows.

    $Available = Available – Request_i$
    $Allocation_i = Allocation_i + Request_i$
    $Need_i = Need_i – Request_i$
  - ✓ If the resulting resource allocation state is safe, the transaction is complete and P_i is allocated its resources. If the new state is unsafe, then P_i must wait for $Request_i$ and old resource allocation state is restored.

- **An Illustrative Example**

  - ✓ To illustrate the use of the banker's algorithm, consider a system with five

Processes $P_o$through $P_4$and three resource types A, B, and C. Resource type Ahas **ten** instances, resource type B has **five** instances, and resource type C has**seven** instances. Suppose that, at time T_0, the following snapshot of the system has been taken.

a)

	Allocation	**Max**	**Available**
	A B C	**A B C**	**A B C**
P_0	0 1 0	7 5 3	3 3 2
P_1	2 00	3 2 2	
P_2	3 02	9 0 2	
P_3	2 11	2 2 2	
P_4	0 02	4 3 3	

- ✓ The content of the matrix**Need**is defined to be **Max - Allocation**and is as follows:

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Now, we have to apply**safety algorithm**to this snapshot **as shown below**,

$P_0 \rightarrow 7\ 4\ 3 \leq 3\ 3\ 2$ is false,
$P_1 \rightarrow 1\ 2\ 2 \leq 3\ 3\ 2$ is true, so work=work + allocation
work=3 3 2 + 2 0 0=5 3 2
$P_2 \rightarrow 600 \leq 5\ 3\ 2$ is false,
$P_3 \rightarrow 011 \leq 5\ 3\ 2$ is true, so work=5 3 2 + 2 1 1=7 4 3
$P_4 \rightarrow 4\ 3\ 1 \leq 74\ 3$ is true, so work=7 4 3 + 0 0 2=7 4 5
$P_2 \rightarrow 600 \leq 7\ 4\ 5$ is true, so work=7 4 5 + 3 0 2=10 4 7
$P_0 \rightarrow 7\ 4\ 3 \leq 10\ 47$ is true, so work=10 4 7 + 0 1 0=10 5 7

- ✓ We claim that the system is currently in a safe state. The sequence **<P_l, P_3, P_4, P_2, P_o>** satisfies the safety criteria.

b)Suppose now the process P_1 requests one additional instance of resource type A and two instances of resource type C, so **$Request_1$= (1,0,2).**

- ✓ To decide whether this request can be immediately granted, we first check that from Resource request algorithm,

$Request_1 \leq Need_1$, that is, **(1,0,2) ≤ (1,2,2)**, which is true then,
$Request_1 \leq$Available, that is, **(1,0,2) ≤ (3,3,2)**, which is true. Then we arrive at the following new state:

	Allocation	**Need**	**Available**
	A B C	**A B C**	**A B C**
P_0	0 1 0	7 4 3	**2 3 0**
P_1	**3 0 2**	**0 2 0**	

P_2	3 0 2	6 0 0
P_3	2 1 1	0 1 1
P_4	0 0 2	4 3 1

- ✓ Now we must determine whether this new system state is safe. We execute safety algorithm **as shown below**and find that the sequence **<P_1, P_3, P_4, P_o, P_2>**satisfies the safety requirement. Hence the request can be immediately granted.

$P_0 \rightarrow 7\ 4\ 3 \leq 2\ 3\ 0$ is false,
$P_1 \rightarrow 0\ 2\ 0 \leq 2\ 3\ 0$ is true, so work=work + allocation
work=2 3 0+ 3 0 2=5 3 2
$P_2 \rightarrow 600 \leq 5\ 3\ 2$ is false,
$P_3 \rightarrow 011 \leq 5\ 3\ 2$ is true, so work=5 3 2 + 2 1 1=7 4 3
$P_4 \rightarrow 4\ 3\ 1 \leq 74\ 3$ is true, so work=7 4 3 + 0 0 2=7 4 5
$P_0 \rightarrow 7\ 4\ 3 \leq 7\ 4\ 5$ is true, so work=7 4 5 + 0 1 0=7 5 5
$P_2 \rightarrow 6\ 0\ 0 \leq 7\ 55$ is true, so work=7 5 5 + 3 0 2=10 5 7

c) Suppose P_4 request for (3,3,0) can it be granted immediately? From Resource Request Algorithm, we must see that when the system is in this state, a request for (3,3,0) by P_4**cannot be granted**, since the resources are not available, that is, **$Request_4 \leq Need_4$,** (3,3,0)≤(4,3,1)……true
$Request_4 \leq Available$, (3,3,0) ≤ (2,3,0)……false

d) Similarly, **a request for (0,2,0) by $P_o$cannot be granted**, even though the resources are available, because the resulting state is unsafe.
that is, **$Request_4 \leq Need_1$**, (0,2,0) ≤ (7,4,3)……true
$Request_1 \leq Available_1$, (0,2,0) ≤ (2,3,0)……true

Now the snapshot changes as follows,

	Allocation	**Need**	**Available**
	A B C	**A B C**	**A B C**
P_0	0 3 0	7 2 3	2 1 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Then we are supposed to apply**safety algorithm**to this snapshot **as shown below**, but no safe sequence is generated, hence the request for (0,2,0) by $P_o$cannot be granted.

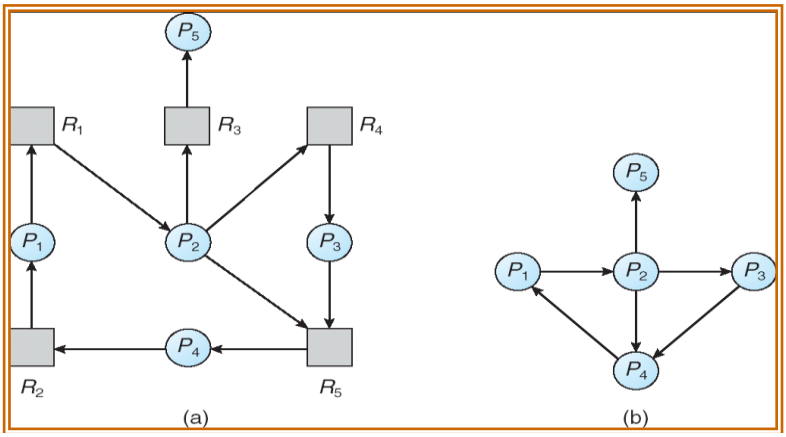
That is, $Need_i \leq Available$
$P_0 \rightarrow 7\ 2\ 3 \leq 2\ 1\ 0$ is false
$P_1 \rightarrow 0\ 2\ 0 \leq 2\ 1\ 0$ is false
$P_2 \rightarrow 600 \leq 2\ 1\ 0$ is false
$P_3 \rightarrow 011 \leq 2\ 1\ 0$ is false
$P_4 \rightarrow 4\ 3\ 1 \leq 2\ 1\ 0$ is false

3.7 Deadlock Detection

- ✓ If a system does not employ either deadlock prevention or a deadlock avoidance algorithm then a deadlock situation may occur. In this environment the system must provide,
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred.
  - An algorithm to recover from the deadlock.

- **Single Instances of each Resource Type**

  - ✓ If all the resources have only a single instance then we can define deadlock detection algorithm that uses a **variant of resource allocation graph** as shown in below **figure (a)** called a **wait-for graph**as shown in below **figure (b)**. This graph is obtained by removing the resource nodes and collapsing appropriate edges.



  - ✓ An edge from P_i to P_j in wait for graph implies that P_i is waiting for P_j to release a resource that P_i needs.
  - ✓ An edge from P_i to P_j exists in wait for graph if and only if the corresponding resource allocation graph contains the edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$.
  - ✓ Deadlock exists within the system if and only if there is a cycle. To detect deadlock the system needs an algorithm that searches for cycle in a graph.

- **Several Instances of Resource Type**

  - ✓ The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
  - ✓ The deadlock detection algorithm includes following time-varying data structures.
    - **Available**. A vector of length mindicates the number of available resources of each type.
    - **Allocation.** An n*mmatrix defines the number of resources of each type currently allocated to each process.
    - **Request**. An n*m matrix indicates the current request of each process. If Request[i][j]=k then P_i is requesting k more instances of resources type R_j.

  - ✓ The deadlock detection algorithm can be defined as follows,

Step 1. Let Work and Finish be vectors of length m and n respectively. Initialize Work= Available. For i=0,1,2…….n-1, if $allocation_i \neq 0$ then Finish[i]=false, else Finish[i]=true.

Step 2. Find an index i such that both

Finish[i]= false

$Request_i \leq Work$

If no such i exists, go to step 4.

Step 3. Work = Work + $Allocation_i$

Finish[i] = true

Go to step 2.

Step 4. If Finish[i] ==false, for some i where $0 \leq i < n$, then a system is in a deadlock state.

- ✓ To illustrate this algorithm, we consider a system with five processes P_0through $P_4$and three resource types A, B, and C. Resource type A has **seven** instances, resource type B has **two** instances, and resource type C has **six** instances. Suppose that, at time T_0, we have the following resource-allocation state:

a)

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 10	0 0 0	0 0 0
P_1	2 00	2 0 2	
P_2	3 03	0 0 0	
P_3	2 11	1 0 0	
P_4	0 02	0 0 2	

- ✓ From the above deadlock detection algorithm, the sequence **<P_0, P_2, P_3, P_1, P_4>**or **<P_0, P_2, P_3, P_4, P_1>**will result in Finish[i]== true for all i. The steps includes,

$Request_i \leq$ work, that is,

$P_0 \rightarrow 0\ 0\ 0 \leq 0\ 0\ 0$ is true, so work=work + allocation

work=0 0 0+ 0 1 0=0 1 0

$P_1 \rightarrow 2\ 0\ 2 \leq 0\ 1\ 0$ is false,

$P_2 \rightarrow 0\ 0\ 0 \leq 0\ 1\ 0$ is true, so work=0 1 0 + 3 0 3=3 1 3

$P_3 \rightarrow 1\ 0\ 0 \leq 3\ 1\ 3$ is true, so work=3 1 3 + 2 1 1=5 2 4

$P_4 \rightarrow 0\ 0\ 2 \leq 5\ 2\ 4$ is true, so work=5 2 4 + 0 0 2=5 2 6

$P_1 \rightarrow 2\ 0\ 2 \leq 5\ 2\ 6$ is true, so work=5 2 6+ 2 0 0= 7 2 6

If P_2 requests an additional instance of typeC**ie., (0, 0, 1)**, the Request matrix is modified as follows,

	Request
	A B C
P_0	0 00
P_1	2 02
P_2	**0 0 1**
P_3	1 00
P_4	0 02

From the deadlock detection algorithm,That is, **$Request_i \leq$ work**,

$P_0 \rightarrow 0\ 0\ 0 \leq 0\ 0\ 0$ is true, so work=work + allocation
work=0 0 0+ 0 1 0=0 1 0
$P_1 \rightarrow 2\ 0\ 2 \leq 0\ 1\ 0$ is false,
$P_2 \rightarrow 0\ 0\ 1 \leq 010$ is false,
$P_3 \rightarrow 1\ 0\ 0 \leq 0\ 1\ 0$ is false,
$P_4 \rightarrow 0\ 0\ 2 \leq 0\ 1\ 0$is false,

- ✓ The system is now deadlocked. Even though we can reclaim resources held by process P_0, but number of available resources is not sufficient to fulfill the requests of other processes.Thus, deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

- **Detection algorithm usage**

  - ✓ This algorithm helps to find,
    - How **often** a deadlock is likely to occur?
    - How**many**processes will be affected by deadlock when it happens?
  - ✓ We can invoke the deadlock detection algorithm when a request for allocation cannot be granted immediately.
  - ✓ If detection algorithm is invoked for every resource request, this will cause a computational time overhead. So the algorithm must be invoked less frequently.

3.8 Recovery from Deadlock

- ✓ There are **two** options for breaking a deadlock,
  - One is to abort one or more processes to break the circular wait.
  - The other is to preempt some resources from one or more of the deadlocked processes.

- **Process Termination**

  - ✓ To eliminate deadlocks by aborting a process, **one of two methods** can be used. In both methods, the system reclaims all resources allocated to the terminated processes.
    - **Abort all deadlocked processes.** This method breaks the deadlock cycle, but at great expense.
    - **Abort one process at a time until the deadlock cycle is eliminated.** This method causes overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
  - ✓ Aborting a process is not a easy task. If the process was in the middle of updating a file, terminating it will leave that file in an incorrect state.
  - ✓ If the **partial termination method** is used, then we must determine which deadlocked process (or processes) should be terminated.
  - ✓ We should abort those processes whose termination will incur the minimum cost. The following **factors** are considered to **select** the process.
    1. What the priority of the process is?
    2. How long the process has computed and how much longer the process will compute before completing its designated task?
    3. How many and what types of resources the process has used?
    4. How many more resources the process needs in order to complete?
    5. How many processes will need to be terminated?

6. Whether the process is interactive or batch?

- **Resource Preemption**

  - ✓ To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
  - ✓ The **three issues** need to be addressed are,
    - **Selecting a victim.** Which resources and which processes are to be preempted? We must determine the order of preemption to minimize cost.
    - **Rollback.** We must roll back the preempted process to some safe state and restart it from that state.
    - **Starvation.** We must ensure that a process can be picked as a victim only a small number of times. The most common solution is to include the number of rollbacks in the cost factor.

Solved Exercises (VTU QP problems)

1. Consider given chart where maximum resource available of type A, B, C and D are 3, 14, 12 and 12 respectively, and answer i) what is content of matrix need? ii) Is system safe?If yes give safe sequence. iii) If request comes from P_1 as(0,4,2,0), can it be granted ?

	Allocation	**Max**	**Available**
	ABCD	**ABCD**	**ABCD**
P_0	0012	0012	1 520
P_1	1000	1750	
P_2	1354	2356	
P_3	0632	0652	
P_4	0014	0656	

a) The content of the matrix**Need**is defined to be **Max - Allocation**and is as follows:

	Need
	A B C D
P_0	0 0 0 0
P_1	0 7 5 0
P_2	1 0 0 2
P_3	0 0 2 0
P_4	0 6 4 2

Now, we have to apply**safety algorithm**to this snapshot **as shown below**,

$P_0 \rightarrow$00 0 0≤1 5 2 0 is true, so work=work + allocation
work= 1 5 2 0 + 0 0 1 2=1 5 3 2
$P_1 \rightarrow$0 7 5 0 ≤ 1 5 3 2 is false,
$P_2 \rightarrow$1 0 0 2≤1 5 3 2 is true, so work=1 5 3 2 + 1 3 5 4=2 8 8 6
$P_3 \rightarrow$0 020 ≤ 2 8 8 6 is true, so work=2 8 8 6 + 0 6 3 2=2 14 11 8
$P_4 \rightarrow$064 2 ≤2 14 11 8 is true, so work=214 118+ 0 0 1 4=2 14 12 12
$P_1 \rightarrow$0 7 5 0≤214 1212 is true, so work=214 1212+ 1 0 0 0=314 12 12

- ✓ We claim that the system is currently in a safe state. The sequence < **P_0,P_2, P_3, P_4, P_1**> satisfies the safety criteria.

b) Suppose now the process P_1 requests for **(0,4,2,0)**. To decide whether this request can be immediately granted, we first check that,

$Request_1 \leq Need_1$, that is, **(0,4,2,0) ≤ (0,7,5,0)**which is true then,
$Request_1 \leq Available$, that is, **(0,4,2,0) ≤ (1,5,2,0)** which is true.
Then we arrive at the following new state:

	Allocation	**Available**
	ABC D	**ABCD**
P_0	001 2	1 1 0 0
P_1	1 4 2 0	
P_2	135 4	
P_3	063 2	
P_4	001 4	

	Need
	A B C D
P_0	0 0 0 0
P_1	**0 3 3 0**
P_2	1 0 0 2
P_3	0 02 0
P_4	0 64 2

- ✓ Now we must determine whether this new system state is safe. We execute safety algorithm as shown below and find that the sequence **<P_0, P_2, P_3, P_4, P_1>** satisfies the safety requirement. Hence the request can be immediately granted.

$P_0 \rightarrow$00 0 0≤1 1 0 0 is true, so work=work + allocation
work= 1 1 0 0 + 0 0 1 2=1 1 12

$P_1 \rightarrow$0 3 3 0 ≤ 1 11 2 is false,

$P_2 \rightarrow$1 0 02 ≤ 1 11 2 is true, so work=1 11 2 + 1 3 5 4=2 4 6 6

$P_3 \rightarrow$0 020 ≤2 4 6 6 is true, so work=2 4 6 6 + 0 6 3 2=2 10 9 8

$P_4 \rightarrow$064 2 ≤ 2 10 9 8 is true, so work=21098 + 0 0 1 4=2 10 10 12

$P_1 \rightarrow$0 3 3 0 ≤ 2101012 is true, so work=2101012 + 142 0=3 1412 12

2.Consider a system consisting of *m* resources of the same type beingshared by *n* processes. A process can request or release only one resourceat a time. Show that the system is deadlock free if the following twoconditions hold:

a. The maximum need of each process is between one resource and*m* resources.
b. The sum of all maximum needs is less than *m* + *n.*

Solution:

The given conditions can be written as
$Max_i \geq 1$ for all i

$$\sum_{i=1}^{n} Max_i < m + n$$

The need value can be calculated as, $Need_i = Max_i - Allocation_i$

If there exists a deadlock then,

$\sum_{i=1}^{n} Allocation_i$ = m

Therefore, $\sum Need_i + \sum Allocation_i = \sum Max_i < m+n$

We get, $\sum Need_i + m < m+n$

Hence, $\sum Need_i < n$

This implies there exist P_i such that $Need_i = 0$.
Since $Max_i \geq 1$, P_i has atleast one resource to release. Hence no deadlock.
For ex, consider m=5, n=3 and below snapshot then we can trace out with above steps and prove that no deadlock occurs.

	Max	**Allocation**	**Available**	**Need**
P_0	2	1	0	1
P_1	3	2		1
P_2	2	2		0

Also, by applying safety algorithm, we can generate a safe sequence as follows.

That is, $Need_i \leq Available$,

$P_0 \rightarrow 1 \leq 0$ is false,

$P_1 \rightarrow 1 \leq 0$ is false,

$P_2 \rightarrow 0 \leq 0$ is true, so work=work + allocation
work= 0 + 2 =2

$P_0 \rightarrow 1 \leq 2$ is true, so work=work + allocation
work= 2 + 1 =3

$P_1 \rightarrow 1 \leq 3$ is true, so work=work + allocation
work= 3 + 2 =5

Hence, the safe sequence **<P_2, P_0, P_1>**is generated and therefore no deadlock.

3.Using Banker's algorithm determine whether the following system is in a safe state.

Process	**Allocation**	**Max**	**Available**
	A B C		A B C A B C
P_0	0 02	0 0 4	10 2
P_1	1 0 0	2 0 1	

P_2	1 3 5	1 3 7
P_3	6 3 2	8 4 2
P_4	1 4 3	1 5 7

If a request from process P_2 arrives for (0 0 2), can the request be granted immediately?

a) The content of the matrix**Need**is defined to be **Max - Allocation**and is as follows:

	Need		
	A	**B**	**C**
P_0	0	0	2
P_1	1	0	1
P_2	0	0	2
P_3	2	1	0
P_4	0	1	4

Now, we have to apply**safety algorithm**to this snapshot **as shown below**,

$P_0 \rightarrow 0\ 0\ 2 \leq 10\ 2$ is true, so work=work + allocation
work =1 0 2 + 0 0 2=1 0 4,

$P_1 \rightarrow 1\ 01 \leq 1\ 0\ 4$ is true, so work=1 0 4 + 1 0 0=2 0 4,

$P_2 \rightarrow 002 \leq 2\ 0\ 4$ is true, so work= 2 0 4 + 1 3 5=3 3 9

$P_3 \rightarrow 210 \leq 3\ 3\ 9$ is true, so work=3 3 9 + 6 3 2=9 6 11

$P_4 \rightarrow 0\ 14 \leq 96\ 11$is true, so work=9 6 11 + 1 4 3=10 10 14

- ✓ We claim that the system is currently in a safe state. The sequence **<P_0, P_1, P_2, P_3, P_4>** satisfies the safety criteria.

b) Suppose now the process P_2 requests for the resources **(0,0,2)**, so **$Request_2$= (0,0,2).**

- ✓ To decide whether this request can be immediately granted or not, we have to first check from Resource request algorithm that,

  $Request_2 \leq Need_2$, that is, **(0,0,2) ≤ (0,0,2)**, which is true then,
  $Request_2 \leq$Available, that is, **(0,0,2) ≤ (1,0,2)**, which is false.

  Hence, a request for (0,0,2) by P_2**cannot be immediately granted**, since the resources are not available

MEMORY MANAGEMENT STRATEGIES

3.9 Background

- ✓ Memory management is concerned with managing the primary memory.
- ✓ Memory consists of array of bytes or words each with its own address.

✓ We can ignore howa program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

- **Basic Hardware**

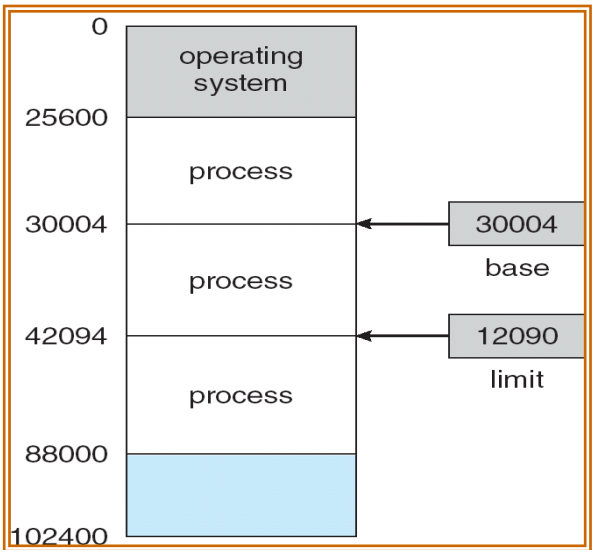
  ✓ **Main memory and the registers** in the processor are the only storage that the CPU can access directly. Hence the program and data must be brought from disk into main memory for CPU to access.

  ✓ Registers can be accessed in **one CPU** clock cycle. But main memory access can take **many CPU** clock cycles.

  ✓ A fast memory called **cache** is placed between main memory and CPU registers.

  ✓ We must ensure correct operation to **protect the operating system** from access by user processes and also to protect user processes from one another. This protection must be provided by the hardware. It can be implemented in several ways and **one such possible implementation** is,

  - We first need to make sure that each process has a separate memory space.
  - To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
  - We can provide this protection by using two registers, **a base and a limit**, as illustrated in below **figure**.

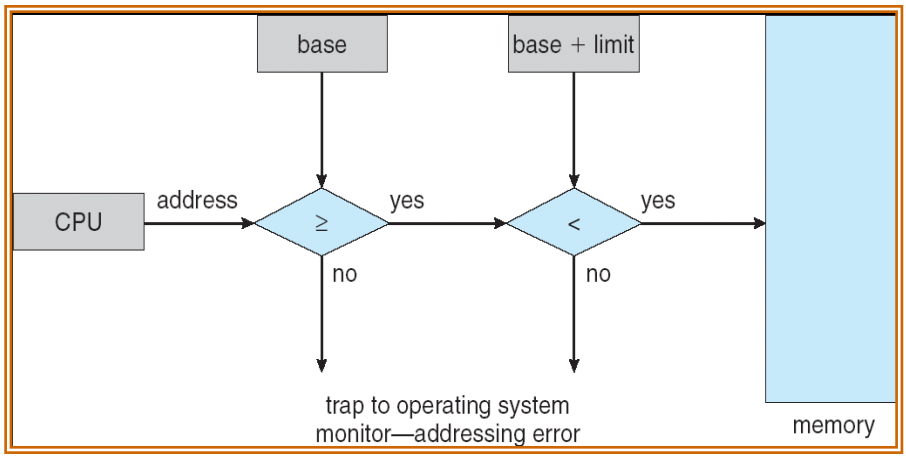


  - The **base register** holds the smallest legal **physical memory address**; the **limit register** specifies the size of **the range**. **For example**, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420940.

  ✓ **Protection of memory space** is accomplished by having the CPU hardware compare everyaddress generated in user mode with the registers.

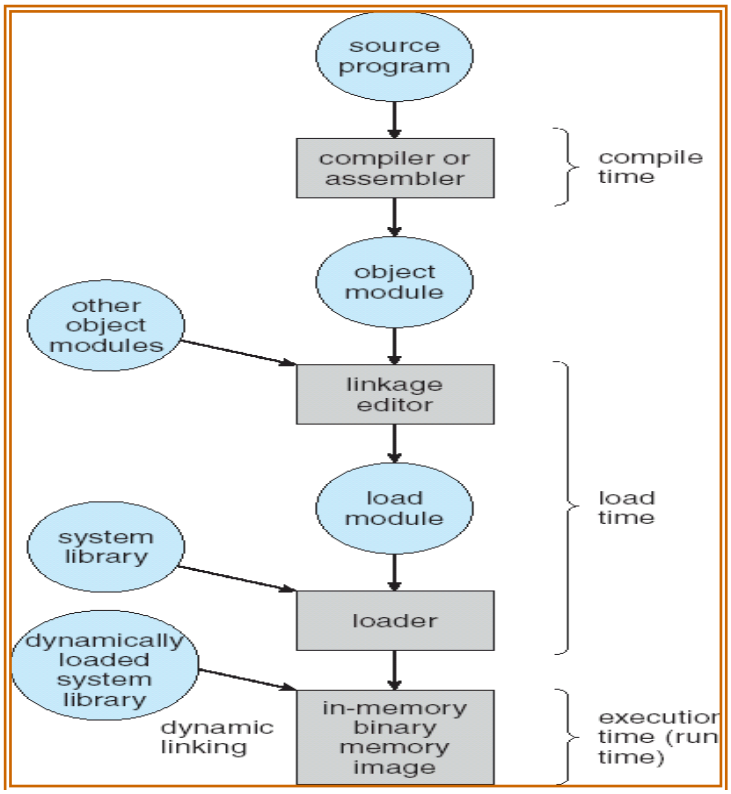
  ✓ Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a **trap to the operating system**, which treats the attempt as a **fatal error** as shown in below **figure**.

- ✓ This prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.
- ✓ The **base and limit registers** uses a special privileged instructions which can be executed only in **kernel mode**, and since only the operating system executes in kernel mode, **only the operating system can load the base and limit registers**.



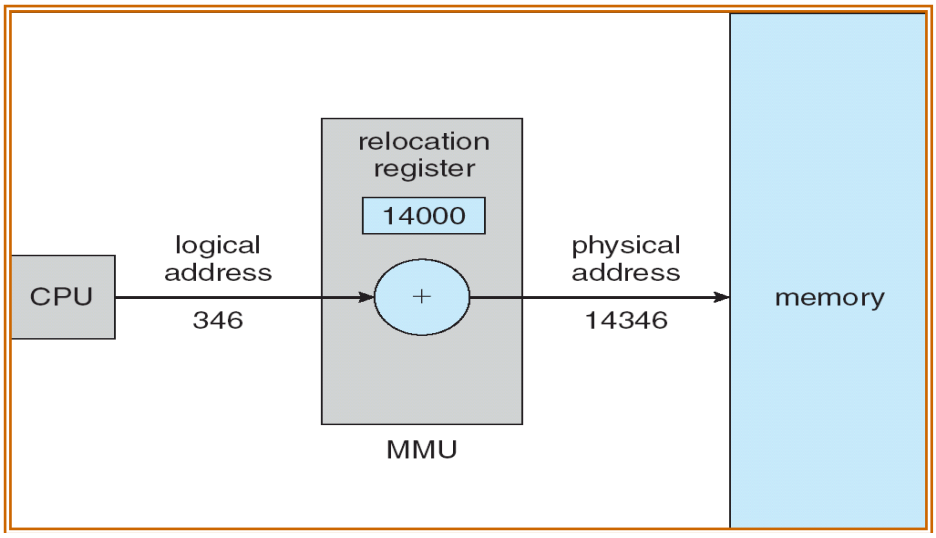
- **Address Binding**

- ✓ Programs are stored on the secondary storage disks as binary executable files.
- ✓ When the programs are to be executed they are brought in to the main memory and placed within a process.
- ✓ The collection of processes on the disk waiting to enter the main memory forms the **input queue.**
- ✓ One of the processes which are to be executed is fetched from the queue and is loaded into main memory.
- ✓ During the execution it fetches instruction and data from main memory. After the process terminates it returns back the memory space.
- ✓ During execution the process will go through several steps as shown in below **figure** and in each step the address is represented in different ways.
- ✓ In source program the address is symbolic. The compiler **binds** the symbolic address to re-locatable address.The loader will in turn bind this re-locatable address to absolute address.
- ✓ Binding of instructions and data to memory addresses can be done at **any step** along the way:
  - **Compile time:** If we know at compile time where the process resides in memory, then **absolute code** can be generated. **For example**, if we know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.
  - **Load time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case, final binding is delayed **until load time.**
  - **Execution time:** If the process is moved during its execution from one memory segment to another then the binding is delayed until run time. Special hardware is used for this. Most of the general purpose operating system uses this method.



- **Logical versus physical address**

  - ✓ The address generated by the CPU is called **logical address or virtual address**.
  - ✓ The address seen by the memory unit i.e., the one loaded in to the memory register is called the **physical address**.
  - ✓ Compile time and load time address binding methods generate **samelogical and physical address**.The execution time addressing binding generate **differentlogical and physical address**.
  - ✓ Set of logical address space generated by the programs is the **logical address space**.Set of physical address corresponding to these logical addresses is the **physical address space.**
  - ✓ The **mapping** of virtual address to physical address during run time is done by the hardware device called **Memory Management Unit (MMU)**.
  - ✓ The **base register** is now called **re-location register**.
  - ✓ Value in the re-location register is added to every address generated by the user process at the time it is sent to memory as shown in below **figure**.
  - ✓ **For example**, if the base is at **14000**, then an attempt by the user to address **location 0** is dynamically relocated to location 14000; an access to location **346** is mapped to location **14346**. The user program never sees the real physical addresses.



- **Dynamic Loading**

  - ✓ For a process to be executed it should be loaded in to the physical memory. The size of the process is limited to the size of the physical memory.Dynamic loading is used to obtain better memory utilization.
  - ✓ In dynamic loading the routine or procedure will not be loaded **until it is called**.
  - ✓ Whenever a routine is called, the calling routine first checks whether the called routine is already loaded or not. If it is not loaded it calls the loader to load the desired program in to the memory and updates the programs address table to indicate the change and control is passed to newly **invoked or called** routine.
  - ✓ The **advantages** are ,
    - Gives better memory utilization.
    - Unused routine is never loaded.
    - Do not need special operating system support.
    - Useful to handle infrequently occurring cases, such as error routines.

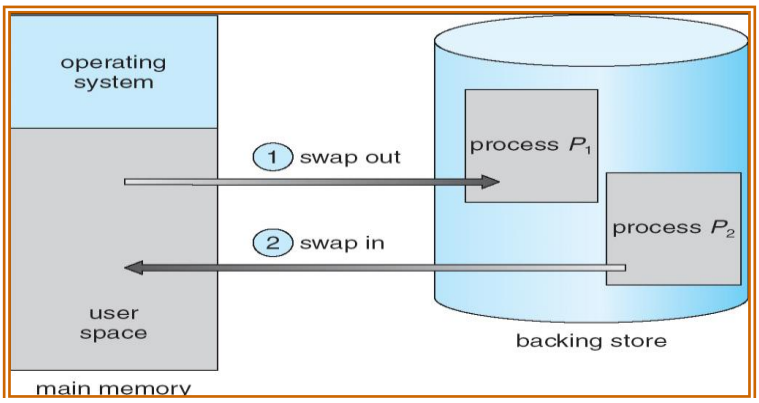
- **Dynamic linking and Shared libraries**

  - ✓ Some operating system supports only the **static linking**.
  - ✓ In dynamic linking only the main program is loaded in to the memory. If the main program requests a procedure, the procedure is loaded and the link is established at the time of references. This linking is postponed until the execution time.
  - ✓ With dynamic linking a **"stub"** is used in the image of each library referenced routine. A **"stub"** is a **piece of code** which is used to **indicate how to locate the appropriate memory resident library routine or how to load library if the routine is not already present.**
  - ✓ When "stub" is executed it checks whether the routine is present is memory or not. If not it loads the routine in to the memory.
  - ✓ This feature can be used to update libraries i.e., library is replaced by a new version and all the programs can make use of this library.
  - ✓ More than one version of the library can be loaded in memory at a time and each program uses its version of the library. Only the program that is compiled with the new version is affected by the changes incorporated in it. Other programs linked

before new version was installed will continue using older library.This system is called **"shared libraries"**.

3.10 Swapping

- ✓ A process can be **swapped** temporarily out of the memory to a **backing store** and then brought back in to the memory for continuing the execution. This process is called swapping.**Ex.** In a multi-programming environment with a round robin CPU scheduling whenever the time quantum expires then the process that has just finished is swapped out and a new process swaps in to the memory for execution as shown in below **figure**.



- ✓ A variant of this swapping policy is priority based scheduling. When a low priority is executing and if a high priority process arrives then a low priority will be swapped out and high priority is allowed for execution. This process is also called as **Roll out and Roll in**.
- ✓ Normally the process which is swapped out will be swapped back to the same memory space that is occupied previously andthis depends upon address binding.
- ✓ The system maintains a **ready queue** consisting of all the processes whose memory images are on the backing store or in memory and are ready to run.
- ✓ The **context-switch time** in a swapping system is **high**. **For ex,**assume that the user process is 10 MB in size and the backing store is a standard hard disk with a transfer rate of 40MB per second. The actual transfer of the 40MB process to or from main memory takes,

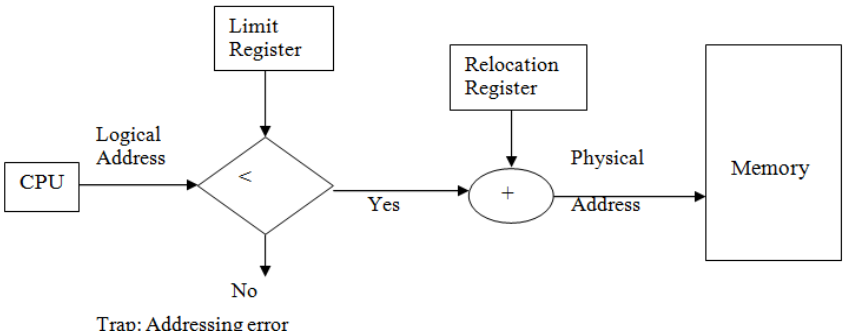
  10MB(10000KB)/40MB(40000KB) per second
  =1/4 second =250 miliseconds
- ✓ Assuming an **average latency** of **8 milliseconds**, the swap time is **258 milliseconds**. Since we must both swap out and swap in, the total swap time is about **516 milliseconds.**
- ✓ Swapping is constrained by other factors,
  - o To swap a process, it should be completely idle.
  - o If a process is waiting for an I/O operation, then the process cannot be swapped.

3.11 Contiguous Memory Allocation

- ✓ The main memory must accommodate both the operating system and the various user processes. One common **method** to allocate main memory in the most efficient way is **contiguous memory allocation**.
- ✓ The memory is divided into two partitions, **one for the resident of operating system and one for the user processes.**

- **Memory mapping and protection**

  - ✓ Relocation registers are used to protect user processes from each other, and to protect from changing OS code and data.
  - ✓ The relocation register contains the value of the smallest physical address and the limit register contains the range of logical addresses.
  - ✓ With relocation and limit registers, each logical address must be less than the limit register.
  - ✓ The MMU maps the logical address dynamicallyby adding the value in the relocation register. This mapped address is sent tomain memory as shown in below **figure**.
  - ✓ The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically.



Memory Allocation

- ✓ One of the simplest methods for memory allocation is to divide memory in to several **fixed partition**. Each partition contains exactly one process. The degree of multi-programming depends on the number of partitions.
- ✓ In **multiple partition method**, when a partition is free, process is selected from the input queue and is loaded in to free partition of memory.When process terminates, the memory partition becomes available for another process.
- ✓ The OS keeps a table indicating which part of the memory is free and is occupied.
- ✓ Initially, all memory is available for user processes and is considered one large block of available memory called a **hole.**
- ✓ When a process requests, the OS searches for large hole for this process. If the hole is too large it is **split** in to two. One part is allocated to the requesting process and other is returned to the set of holes.
- ✓ The set of holes are searched to determine which hole is best to allocate.
- ✓ **Dynamic storage allocation problem** is one which concerns about how to satisfy a request of size n from a list of free holes. There are three **strategies/solutions** to select a free hole,
  - **First bit:** Allocates first hole that is big enough. This algorithm scans memory from the beginning and selects the first available block that is large enough to hold the process.
  - **Best bit:** It chooses the hole i.e., closest in size to the request. It allocates the smallest hole i.e., big enough to hold the process.
  - **Worst fit:** It allocates the largest hole to the process request. It searches for the largest hole in the entire list.
- ✓ First fit and best fit are the most popular algorithms for dynamic memory allocation. **All these algorithms suffer from fragmentation.**

- **Fragmentation**

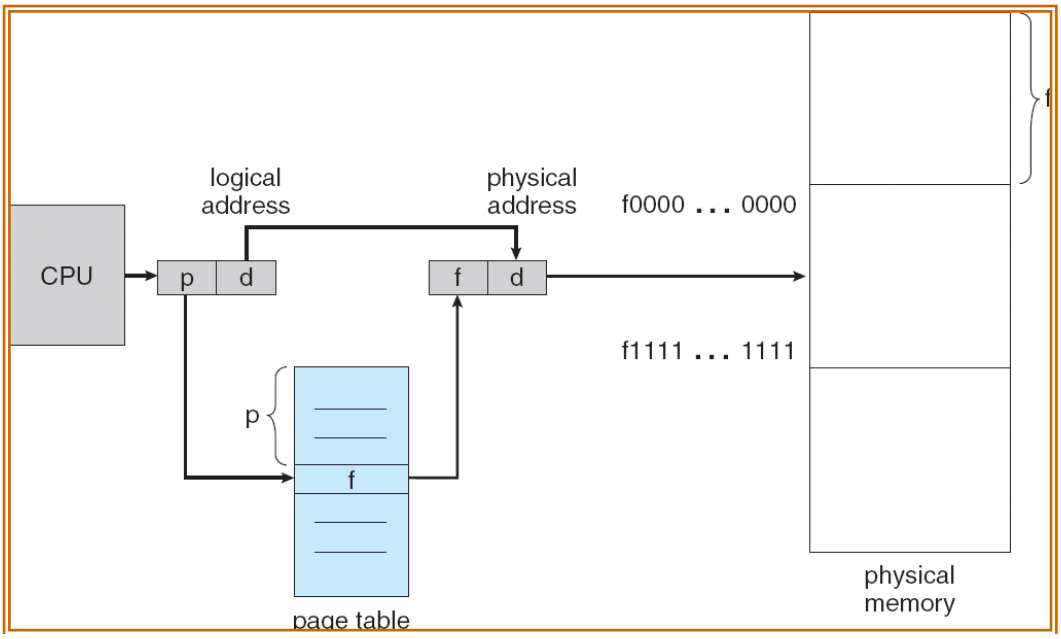
  - ✓ **External Fragmentation** exists when there is enough memory space exists to satisfy the request, but it is not contiguous.Storage is fragmented into a large number of small holes.
  - ✓ External Fragmentation may be either minor or a major problem.
  - ✓ Statistical analysis of first fit reveals that, even with some optimization, given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation. That is, one-third of memory may be unusable. This property is known as the **50-percent rule.**
  - ✓ Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes.
  - ✓ The overhead to keep track of this hole will be substantially larger than the hole itself.
  - ✓ The general approach to avoid this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory.
  - ✓ The difference between these two numbers is **internal fragmentation** that is internal to a partition.
  - ✓ One solution to over-come external fragmentation is **compaction**. The goal is to move all the free memory together to form a large block. Compaction is possible only if the re-location is dynamic and done at execution time.
  - ✓ **Another solution** to the external fragmentation problem is to permit the logical address space of a process to be non-contiguous. This can be achieved with Paging and Segmentation schemes.

3.12 Paging

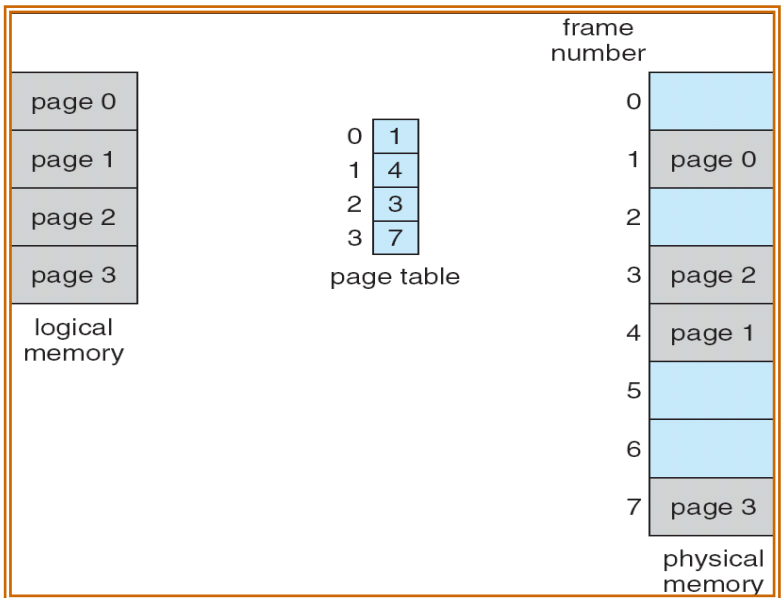
- ✓ Paging is a **memory management scheme** that permits the physical address space of a process to be **non-contiguous**. Support for paging is handled by **hardware.**
- ✓ Paging avoids the considerable problem of fitting the varying sized memory chunks on to the backing store.

- **Basic Method**

  - ✓ Physical memory is broken in to fixed sized blocks called **frames (f)** and Logical memory is broken in to blocks of same size called **pages (p).**
  - ✓ When a process is to be executed its pages are loaded in to available frames from the backing store. The backing store is also divided in to **fixed-sized blocks of same size as memory frames.**
  - ✓ The below **figure** shows paging hardware.



- ✓ Logical address generated by the CPU is divided in to **two parts: page number (p) and page offset (d)**.
- ✓ The page number (p) is used as **index** to the page table. The page table contains base address of each page in physical memory. This base address is combined with the page offset to define the physical memory i.e., sent to the memory unit. The paging model memory is shown in below **figure.**



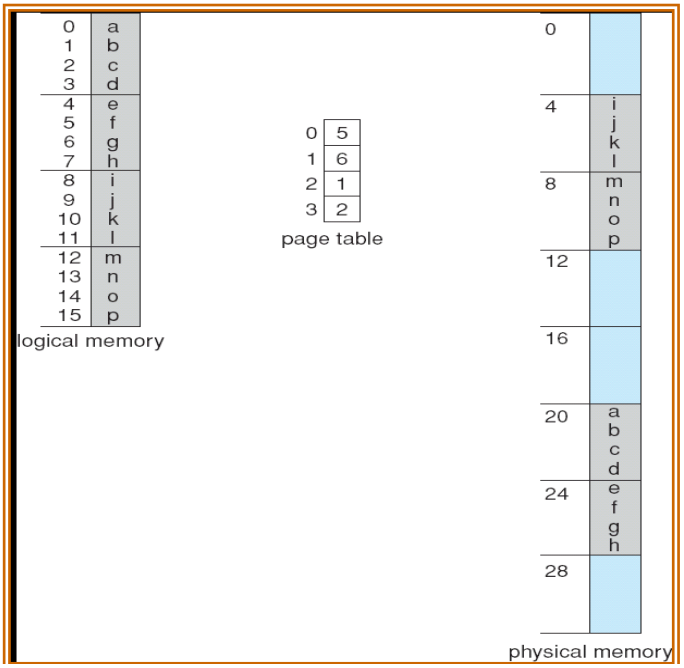
- ✓ The page size is defined by the hardware. The size is the power of 2, varying between **512 bytes and 16Mb per page.**
- ✓ If the size of logical address space is $\mathbf{2^m}$ address unit and page size is 2^n, then high order **m-n** designates the **page number** and **n** low order bits represents **page offset**. Thus logic address is as follows.

page number page offset

p	d

m-n **n**

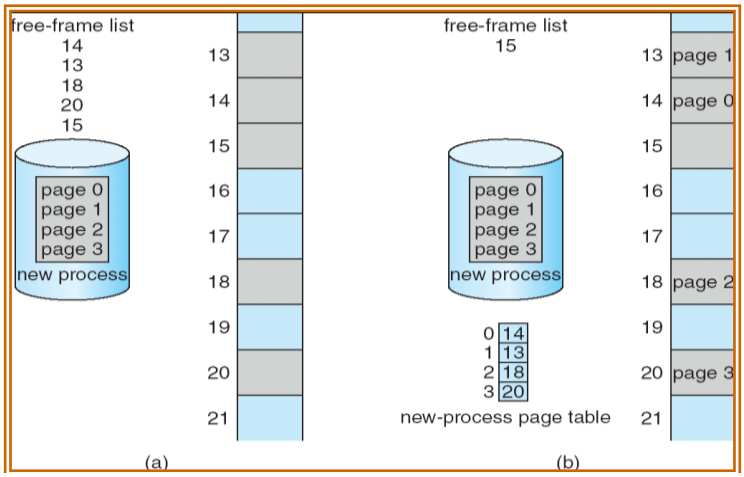
- ✓ **Ex:**To show how to map logical memory in to physical memory, consider a page size of 4 bytes and physical memory of 32 bytes (8 pages) as shown in below **figure**.
  - **a.** Logical address 0 is page 0 and offset 0 and Page 0 is in frame 5. The **logical address 0** maps to physical address **[(5*4) + 0]=20.**
  - **b. Logical address 3** is page 0 and offset 3 and Page 0 is in frame 5.The **logical address 3** maps to **physical address [(5*4) + 3]=23.**
  - c. **Logical address 4** is page 1 and offset 0 and page 1 is mapped to frame 6. So logical address 4 maps to **physical address [(6*4) + 0]=24**.
  - d. **Logical address 13** is page 3 and offset 1 and page 3 is mapped to frame 2. So logical address 13 maps to **physical address [(2*4) + 1]=9.**



- ✓ In paging scheme, we have **no external fragmentation**. Any free frame can be allocated to a process that needs it. But we may have some **internal fragmentation**.
- ✓ If the memory requirements of a process do not happen to coincide with page boundaries, the lastframe allocated may not be completely full.
- ✓ **For example**, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in **internal fragmentation** of 2,048 - 1,086= **962 bytes**.
- ✓ When a process arrives in the system to be executed, its size expressed in pages is examined. Each page of the process needs one frame. Thus, if the process requires

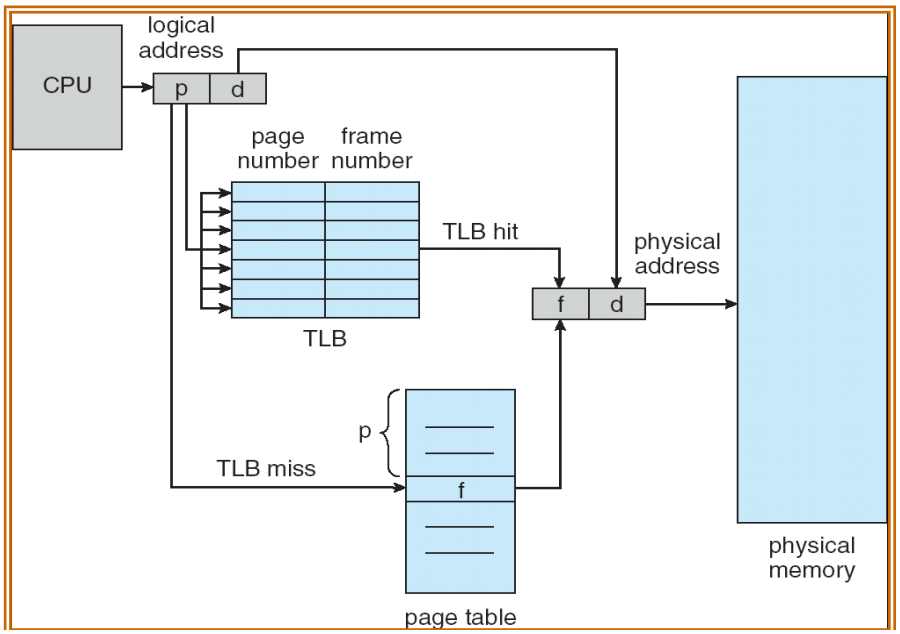
npages, at least **n**frames must be available in memory. If n frames are available, they are allocated to this arriving process.

- ✓ The first page of the process is loaded in to one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame and its frame number is put into the page table and so on, as shown in below **figure (a) before allocation, (b) after allocation**.



- **Hardware Support**

- ✓ The **hardware implementation** of the page table can be done in several ways. The simplest method is that the page table is implemented as a set of **dedicated registers.**
- ✓ The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). But most computers, allow the page table to be very large (for example, 1 million entries) and for these machines, the use of fast registers to implement the page table is not feasible.
- ✓ So the page table is kept in the main memory and a **page table base register (PTBR)** points to the page table and **page table length register (PTLR)** indicates size of page table. Here two memory accesses are needed to access a byte and thus memory access is slowed by a factor of 2.
- ✓ The only solution is to use a special, fast lookup hardware **cache** called **Translation look aside buffer(TLB)**. TLB is associative, with high speed memory. Each entry in TLB contains **two** parts, **a key and a value.** When an associative register is presented with an item, it is compared with all the key values.If found, the corresponding value field is returned. Searching is fast but hardware is expensive.
- ✓ TLB is used with the page table **as follows**,
  - o TLB contains only few page table entries.
  - o When a logical address is generated by the CPU, its page number is presented to TLB.
  - o If the page number is found, then its frame number is immediately available **(TLB hit)** andis used to access the actual memory.If the page number is not in the TLB **(TLB miss)** the memory reference to the page table must be made.
  - o When the frame number is obtained we can use it to access the memory as shown in below **figure**.The page number and frame number are added to the TLB, so that they will be found quickly on the next reference.
  - o If the TLB is full of entries, the OS must select anyone for **replacement.**
  - o Some TLBs allow entries to be **wired down** meaning that they cannot be removed from the TLB.



- ✓ Some TLBs store **Address Space Identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process.
- ✓ The percentage of time that a page number is found in the TLB is called **hit ratio**.
- ✓ **For example,** an **80-percent hit ratio** means that we find the desired page number in the TLB 80 percent of the time. If it takes **20 nanoseconds** to search the TLB and **100 nanoseconds to access memory**, then a mapped-memory access takes **120 nanoseconds when the page number is in the TLB**. If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of **220 nanoseconds**. Thus the effective access time is,

Effective Access Time (EAT)= 0.80 x 120 + 0.20 x 220
= **140 nanoseconds.**

In this example, we suffer a **40-percent slowdown** in memory-access time (from 100 to 140 nanoseconds).

- ✓ For a **98-percent hit ratio**we have

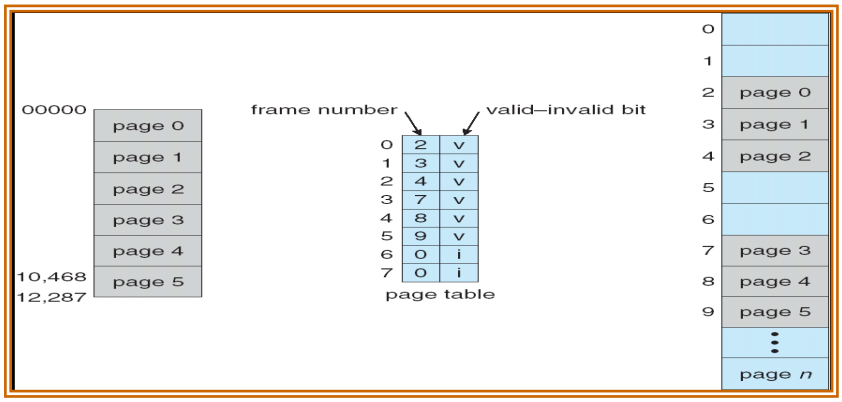
Effective Access Time (EAT) = 0.98 x 120 + 0.02 x 220
= **122 nanoseconds.**

- ✓ This increased hit rate produces only a **22 percent slowdown** in access time.

- **Protection**

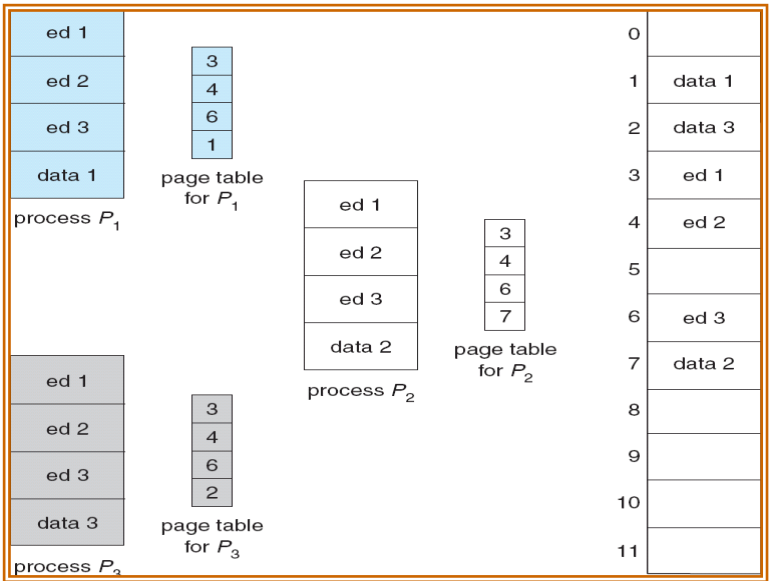
- ✓ Memory protection in paged environment is done by **protection bits** that are associated with each frame.These bits are kept in page table.
- ✓ One bit can define a page to be read-write or read-only.
- ✓ One more bit is attachedto each entry in the page table, a **valid-invalid** bit.
- ✓ A valid bit indicates that associated page is in the process's logical address space and thus it is a legal or valid page.
- ✓ If the bit is invalid, it indicates the page is not in the process's logical address space and is illegal. Illegal addresses are trapped by using the valid-invalid bit.
- ✓ The OS sets this bit for each page to allow or disallow accesses to that page.

- ✓ **For example,** in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we have the situation shown in below **figure**. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, we find that the valid -invalid bit is set to **invalid**, and the computer will trap to the operating system (**invalid page reference**).



- **Shared Pages**

- ✓ An advantage of paging is the possibility of **sharing common code**. This consideration is particularly important in a time-sharing environment.
- ✓ Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support 40 users. If the code is **reentrant code (or pure code)** it can be shared as shown in below **figure**. Here there are three-page editor-each page 50 KB in size and are being shared among three processes. Each process has its own data page.
- ✓ Reentrant code is **non-self-modifying** code**(Read only).** It never changes during execution. Thus, two or more processes can execute the same code at the same time.
- ✓ Each process has its own copy of registers and data storage to hold the data for the process's execution.
- ✓ Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2150 KB instead of 8,000 KB. **(i.e., 150+40*50= 2150 KB).**

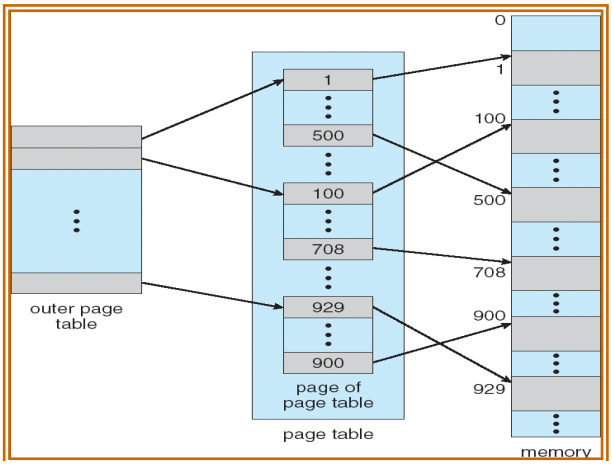


3.13 Structure of the Page Table

- **Hierarchical paging**

✓ Recent computer system support a large logical address apace from 2^{32} **to** 2^{64} and thus page table becomes large. So it is very difficult to allocate contiguous main memory for page table. One **simple solution** to this problem is to divide page table in to smaller pieces.

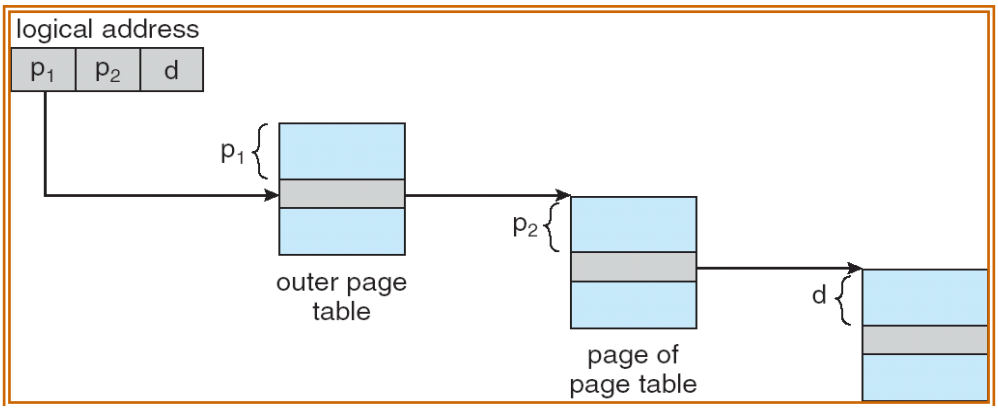
✓ One way is to use **two-level paging algorithm** in which the page table itself is also paged as shown in below **figure.**



✓ **Ex.** In a 32- bit machine with page size of 4kb, a logical address is divided in to a page number consisting of 20 bits and a page offset of 12 bit. The page table is further divided since the page table is paged, the page number is further divided in to 10 bit page number and a 10 bit offset. So the logical address is,

Page number		**page offset**
P_1	**P_2**	**d**
10	10	12

✓ **P_1** is an index into the outer page table and **P_2**is the displacement within the page of the outer page table. The **address-translation method** for this architecture is shown in below **figure**. Because address translation works from the outer page table inward, this scheme is also known as a**forward-mapped page table.**



- ✓ For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. Suppose the page size in such a system is **4 KB** the page table consists of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables can be one page long, or contain 2^{10} 4-byte entries. The addresses look like this,

Outer page	inner page	offset
P_1	P_2	d
42	10	12

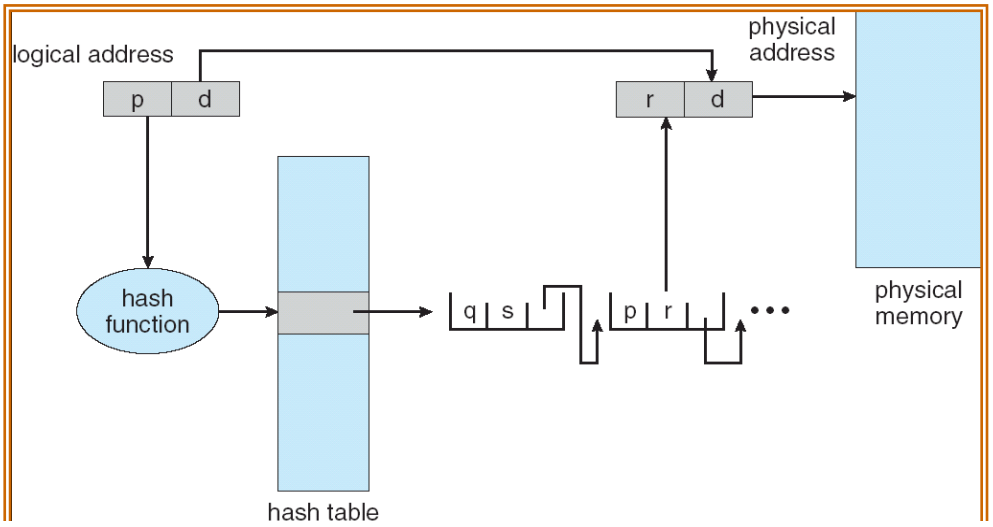
- ✓ The outer page table consists of 2^{42} entries, or 2^{44} bytes. The one way to avoid such a large table is to divide the outer page table into smaller pieces.
- ✓ We can avoid such a large table using**three-level paging scheme**.

2^{nd} outer page	outer page	inner page	offset
P_1	P_2	P_3	d
32	10	10	12

- ✓ The outer page table is still 2^{34} bytes in size. The next step would be a **four-level paging scheme.**

- **Hashed page table**

- ✓ Hashed page table handles the address space larger than 32 bit. The virtual page number is used as **hash value**. Linked list is used in the hash table which contains a list of elements that hash to the same location.
- ✓ Each element in the hash table contains the following three fields,
  - **Virtual page number**
  - **Mapped page frame value**
  - **Pointer to the next element in the linked list**
- ✓ The algorithm works as follows,
  - Virtual page number is taken from virtual/logical address space and is hashed in to the hash table.
  - Virtual page number is compared with **field 1 of** the first element in the linked list.



- If there is a match, the corresponding page frame in **field 2** is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for matching virtual/logical page number. This scheme is shown in above **figure.**
- **Clustered pages** are similar to hash table but one difference is that each entity in the hash table refer to several pages.

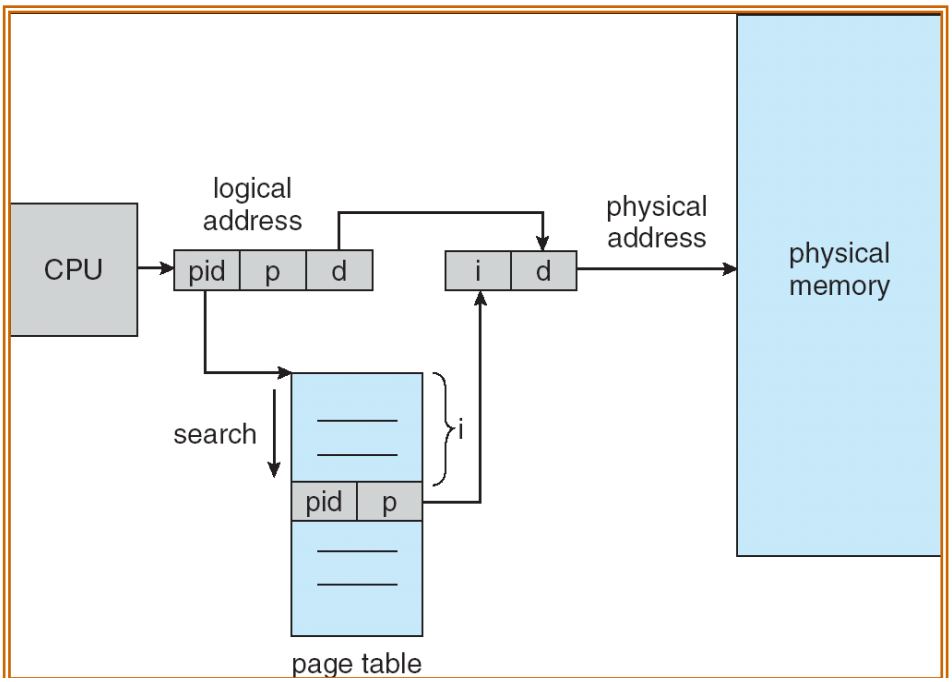
- **Inverted Page Tables**

✓ Page tables may consume large amount of physical memory just to keep track of how other physical memory is being used.

✓ To solve this problem, we can use an inverted page table that has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location with information about the process that owns the page.

✓ Thus, only one page table is in the system, and it has only one entry for each page of physical memory. The below **figure** shows the operation of an inverted page table.

✓ The inverted page table entry is a pair **<process-id, page number>**. **Where process-id assumes the role of the address-space identifier**. When a memory reference is made, the part of virtual address consisting of **<process-id, page number>**is presented to memory sub-system.

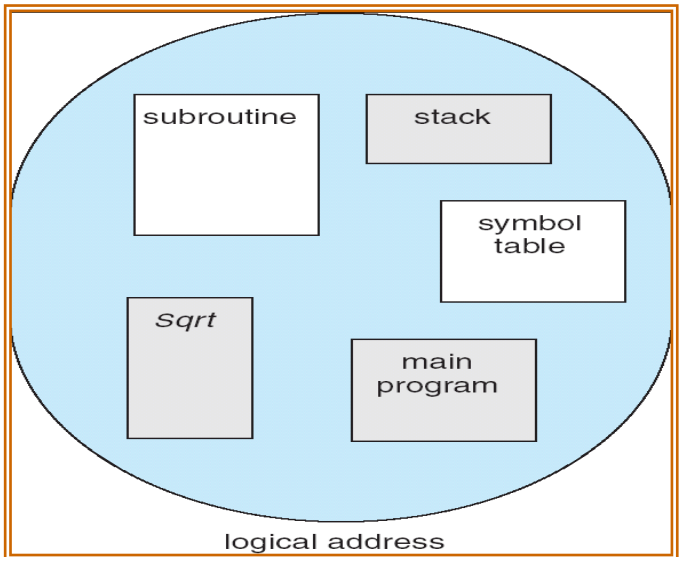


✓ The inverted page table is searched for a match. If a match is found at **entry i,** then the physical address **<i, offset>** is generated. If no match is found then an illegal address access has been attempted.

✓ This scheme **decreases the amount of memory** needed to store each page table, but increases the amount of time needed to search the table when a page reference occurs.

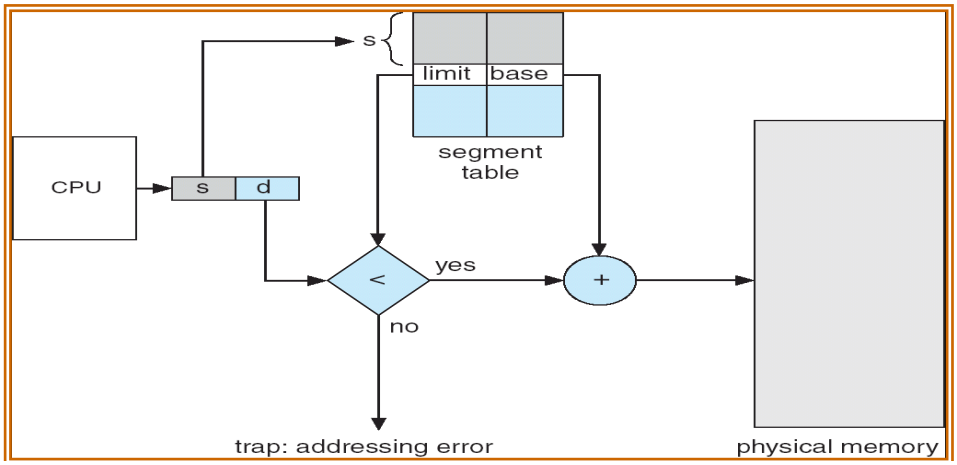
3.14 Segmentation

- **Basic method**
  - ✓ Users prefer to view memory as a collection of **variable-sized segments**, **with no ordering among segments** as shown in below **figure**.



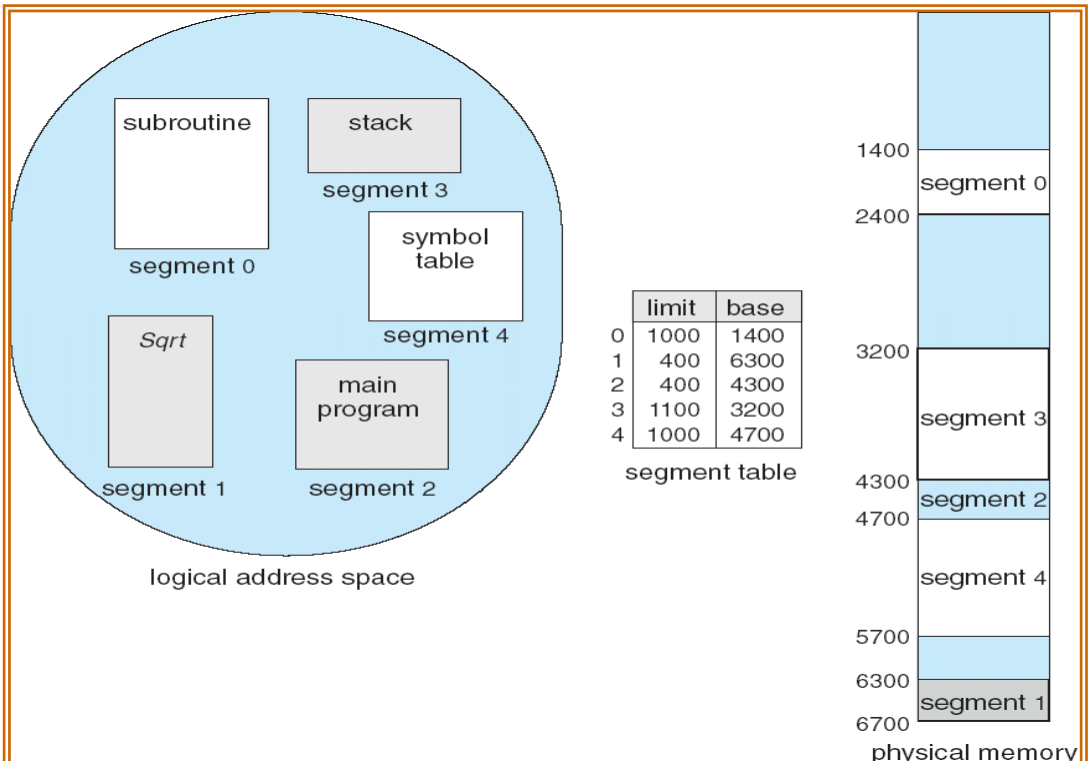
  - ✓ **Segmentation** is a memory-management scheme that supports the user view of memory.
  - ✓ A **logical address** is a collection of segments. Each segment has a **name and length.** The address specifies both the **segment name and the offset** within the segments.
  - ✓ The segments are numbered and are referred by a segment number. So the logical address consists of<**segment number, offset**>.

- **Hardware**
  - ✓ **Segment table**maps**2-Dimensional user defined address** in to **1-Dimensional physical address.**



  - ✓ Each entry in the segment table has a **segment base and segment limit**.
  - ✓ The segment **base contains the starting physical address where the segment resides and limit specifies the length of the segment**.

- ✓ The use of segment table is shown in the below figure.
- ✓ Logical address consists of two parts, **segment number s and an offset d.**
- ✓ The segment number is used as an i**ndex** to segment table.The offset must be in between **0 and limit**, if not an error is reported to OS.
- ✓ If legal the offset is added to the base to generate the actual physical address.
- ✓ The segment table is an **array of base-limit register pairs**.
- ✓ **For example**, consider the below **figure**. We have **five segments** numbered from 0 through 4. Segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 +53= 4353. A reference byte 852 of segment 3, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.



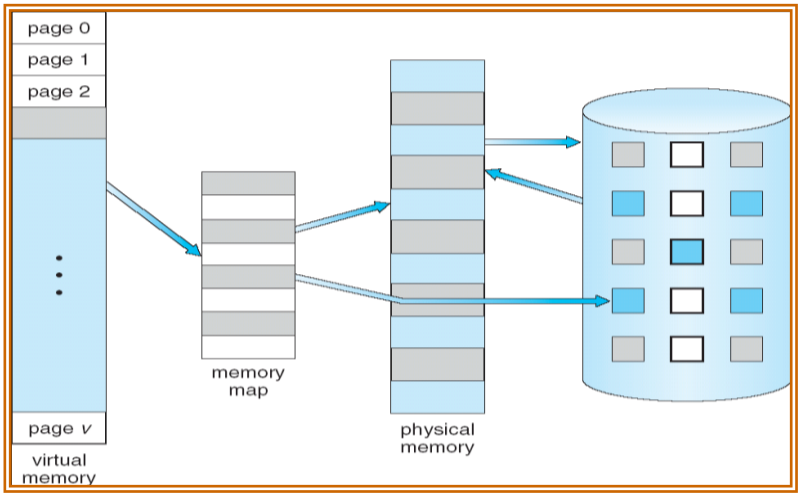
---0O0---

MODULE-4
VIRTUAL MEMORY MANAGEMENT

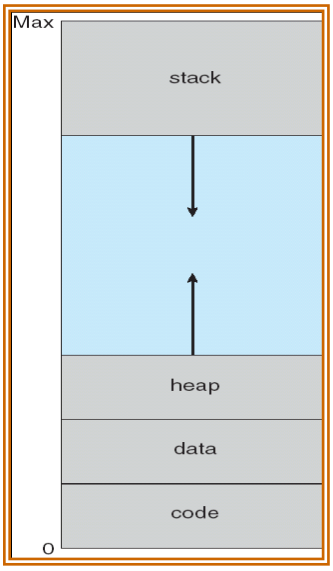
- ✓ Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- ✓ The main advantage of this scheme is that programs can be larger than physical memory.
- ✓ Virtual memory also allows processes to share files easily and to implement shared memory. It also provides an efficient mechanism for process creation.
- ✓ But virtual memory is not easy to implement.

4.1 Background

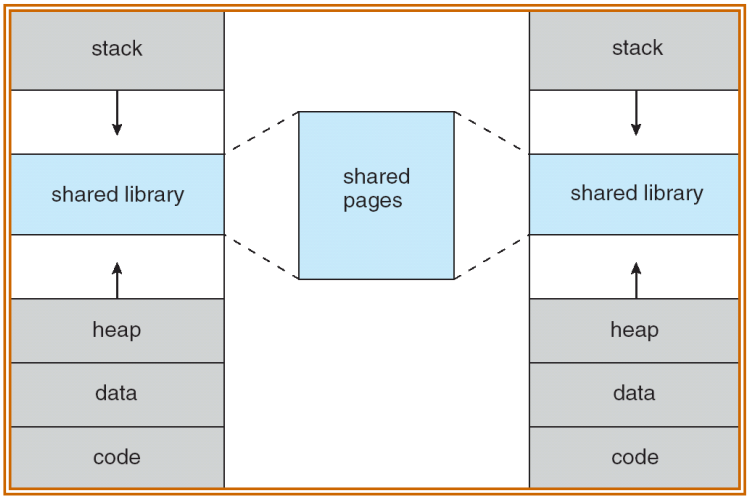
- ✓ An examination of real programs shows us that, in many cases, the entire program is not needed to be in physical memory to get executed.
- ✓ Even in those cases where the entire program is needed, it may not need all to be at the same time.
- ✓ The ability to execute a program that is only partially in memory would confer many **benefits,**
  - o A program will not be limited by the amount of physical memory that is available.
  - o More than one program can run at the same time which can increase the throughput and CPU utilization.
  - o Less **I/O** operation is needed to swap or load user program in to memory. So each user program could run faster.
- ✓ Virtual memory involves the **separation of user's logical memory from physical memory.** This separation allows an extremely large virtual memory to be provided for programmers when there is small physical memory as shown in below **figure.**



- ✓ The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory.
- ✓ In below **figure,** we allow **heap** to grow **upward** in memory as it is used for dynamic memory allocation and**stack** to grow **downward** in memory through successive function calls.
- ✓ The **large blank space (or hole)** between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.
- ✓ Virtual address spaces that include holes are known as **sparse address spaces**.



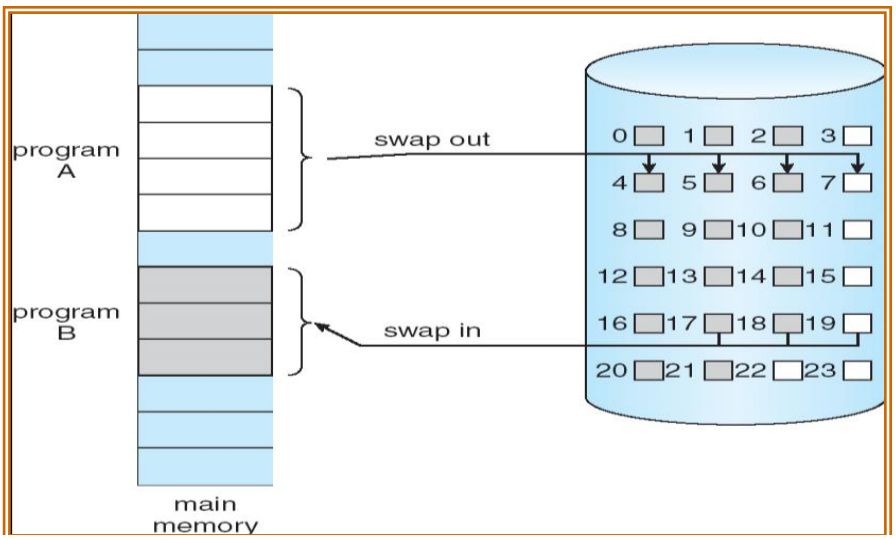
- ✓ Virtual memory allows files and memory to be shared by two or more processes through **page sharing**. This leads to the following **benefits,**
  - **System libraries** can be shared by several processes through mapping of the shared object into a virtual address space as shown in below **figure.**
  - Virtual memory allows one process to create a region of memory that it can share with another processas shown in below **figure**.
  - Virtual memory can allow pages to be shared during **process creation** with the fork() system call thus speeding up process creation.



4.2 Demand Paging

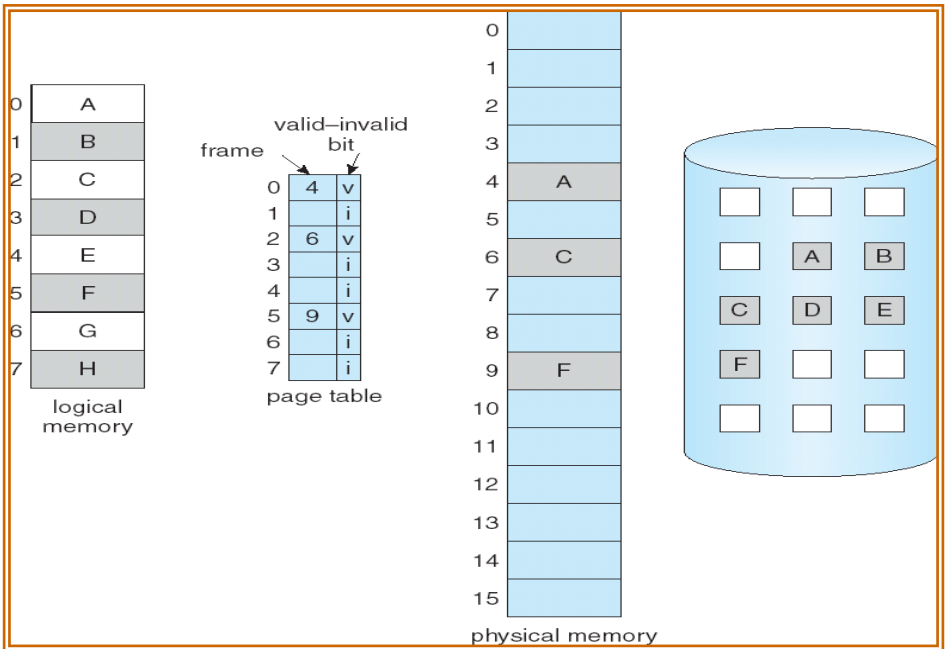
- ✓ Virtual memory is implemented using **Demand Paging**.
- ✓ A demand paging is similar to paging system with swapping as shown in below **figure**where the processes reside in secondary memory.
- ✓ When we want to execute a process we swap it in to memory.Rather than swapping the entire process into memory we use a **lazy swapper** which**never swaps** a page into memory unless that page will be needed.

- ✓ A swapper manipulates entire process, whereas a**pager** is concerned with the individual pages of a process. We thus use pagerrather than swapperin connection with demand paging.

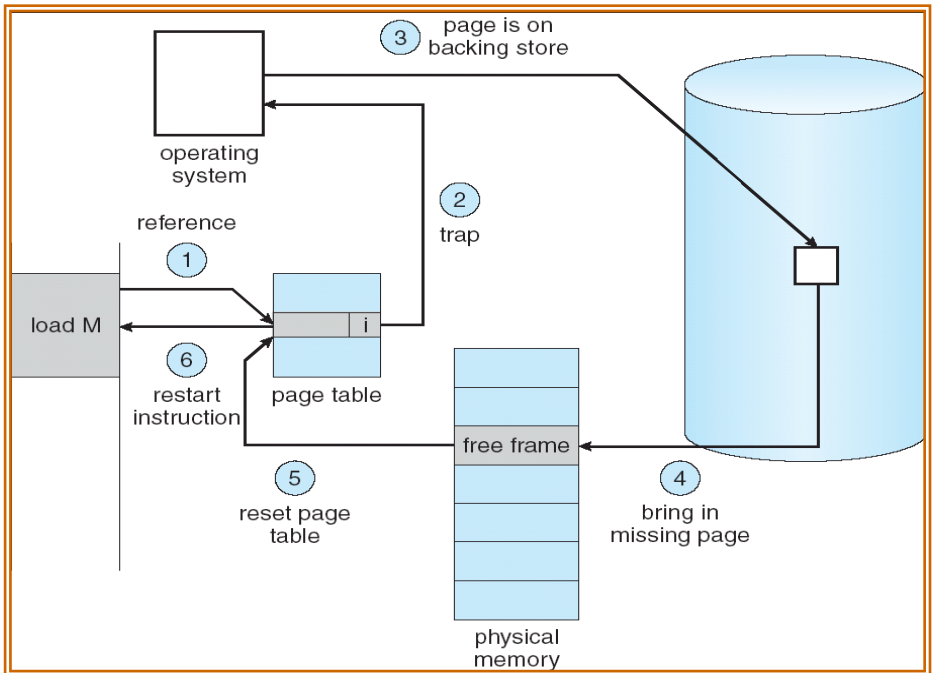


- **Basic concepts**

  - ✓ We need some form of **hardware support** to distinguish between the pages that are in memory and the pages that are on the disk.
  - ✓ The **valid-invalid bit** scheme can provide this .If the bit is valid then the page is both legal and is in memory.If the bit is invalid then either the page is not valid or is valid but is currently on the disk**.**
  - ✓ The **page-table entry** for a page that is brought into memory is set as valid but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk as shown in below **figure**.
  - ✓ Access to the page which is marked as invalid causes a **page fault trap**.



The **steps for handling** page fault is straight forward and is shown in below **figure**,



1. We check the internal table usually PCB (Process Control Block)of the process to determine whether the reference made is valid or invalid.
2. If invalid, terminate the process. If valid, then the page is not yet loaded and we now page it in.
3. We find a free frame.
4. We schedule disk operation to read the desired page in to newly allocated frame.
5. When disk read is complete, we modify the internal table kept with the process to indicate that the page is now in memory.
6. We restart the instruction which was interrupted by the trap. The process can now access the page.

- ✓ In extreme cases, we can start executing the process **without pages** in memory. When the OS sets the instruction pointer of process which is not in memory, it generates a page fault. After this, page is brought in to memory then the process continues to execute and faulting every time until every page that it needs is in memory. This scheme is known as **pure demand paging**. That is, it never brings the page in to memory until it is required.
- ✓ The **hardware support f**or demand paging is same as paging and swapping.
  - **Page table:** It has the ability to mark an entry invalid through valid-invalid bit.
  - **Secondary memory:** This holds the pages that are not present in main memory. It is a high speed disk.It is known as the **swap device,** and the section of disk used for this purpose is known as **swap space.**
- ✓ A **crucial requirement for demand paging** is the need to be able to restart any instruction after a page fault.
- ✓ A page fault may occur at any memory reference. If the page fault occurs on instruction fetch, we can restart by fetching the instruction again.
- ✓ If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.
- ✓ As an **example**, consider three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

- ✓ If we fault when we try to store in C (because C is in a page not currently in memory), we have to get the desired page, bring it into memory, correct the page table, and restart the instruction.
- ✓ The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again.

- **Performance of demand paging**

- ✓ Demand paging can have significant effect on the performance of the computer system.
- ✓ Let us compute the **effective access time** for a demand-paged memory.
- ✓ The memory-access time, denoted **ma**, ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is **equal to** the memory access time.
- ✓ If a page fault occurs, we must first read the relevant page from disk and then access the desired word.
- ✓ Let**p**be the **probability** of a page fault (0<=p<=1). The **effective access time** is then,

Effective Access Time= (1 - p) *ma + p * page fault time.

- ✓ To compute the effective access time, we must know how much **time** is needed to **service a page fault**. A page fault causes the following **sequence** to occur,

1. *Trap to the OS.*
2. *Save the user registers and process state.*
3. *Determine that the interrupt was a page fault.*
4. *Check that the page reference was legal and determine the location of the page on disk.*
5. *Issue a read from disk to a free frame.*
   a. *Wait in a queue for this device until the read request is serviced.*
   b. *Wait for the device seek and/or latency time.*
   c. *Begin the transfer of the page to a free frame.*
6. *While waiting, allocate the CPU to some other user.*
7. *Receive an interrupt from the disk I/O subsystem.*
8. *Save the registers and process statefor the other user.*
9. *Determine that the interrupt was from the disk.*
10. *Correct the page table and other table to show that the desired page is now in memory.*
11. *Wait for the CPU to be allocated to this process again.*
12. *Restore the user registers,process state and new page table, then resume the interrupted instruction.*

- ✓ The **three major components** of the **page-fault service time**,
  1. Service the page-fault interrupts.

2. Read in the page.
3. Restart the process.

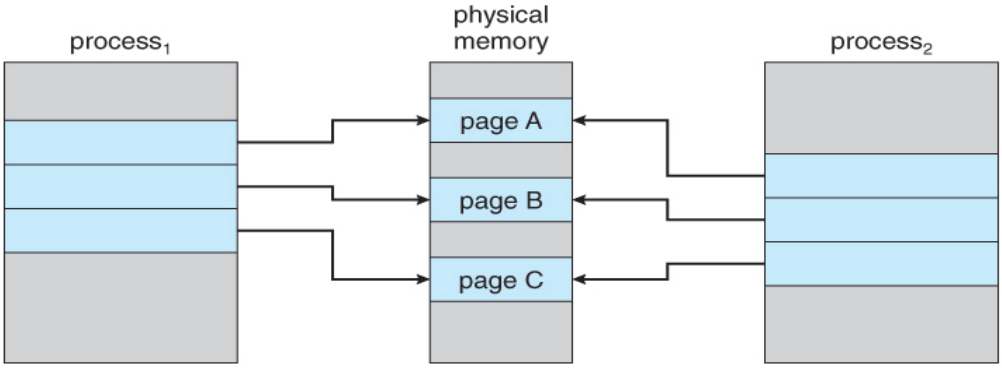
- ✓ With an **average page-fault service time** of **8 milliseconds** and a **memory access time** of **200 nanoseconds**, the effective access time in nanoseconds is

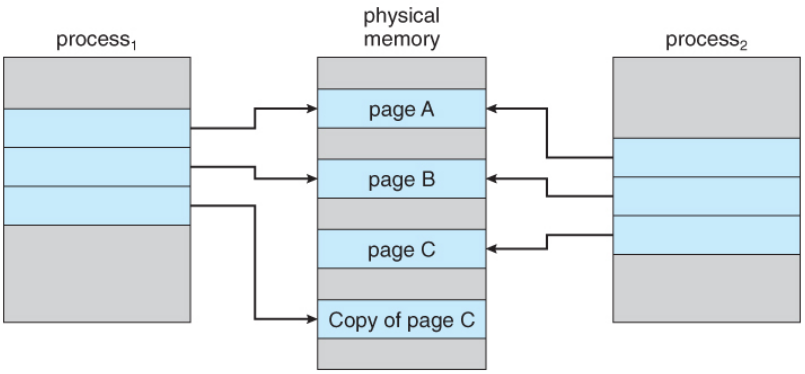
  Effective Access Time = (1 - p) * (200) + p (8 milliseconds)
  = (1 - p) * 200 + p * 8,000,000
  = 200 + 7,999,800 *p.
- ✓ The effective access time is **directly proportional** to the**page-fault rate.**
- ✓ If one access out of 1,000 causes a page fault, the effective access time is **8.2 microseconds.** The computer will be **slowed down by a factor of 40** because of demand paging. If we want **performance degradation to be less than 10 percent**, then,

  10% of 200 ns = 20, ie., 220 ns.
  So,
  220> 200 + 7,999,800 *p,
  20 >7,999,800 *p,
  P <7,999,800 ÷ 20
  p< 0.0000025 (less than 10% hike in **memory access time)**
  or
  7,999,800 ÷ 20 = 3,99,999
  From this we can conclude that, **allow one page out of 3,99,990 to fault**. So **memory access time will be** less than 10%.

4.3 Copy-on-write

- ✓ Copy-on-write technique allows both the parent and the child processes to share the same pages. These pages are marked as **copy-on-write pages** i.e., if either process writes to a shared page, a copy of shared page is created.
- ✓ Copy-on-write is illustrated in below **figures,** which shows the contents of the physical memory **before and after** process 1 modifies page C.

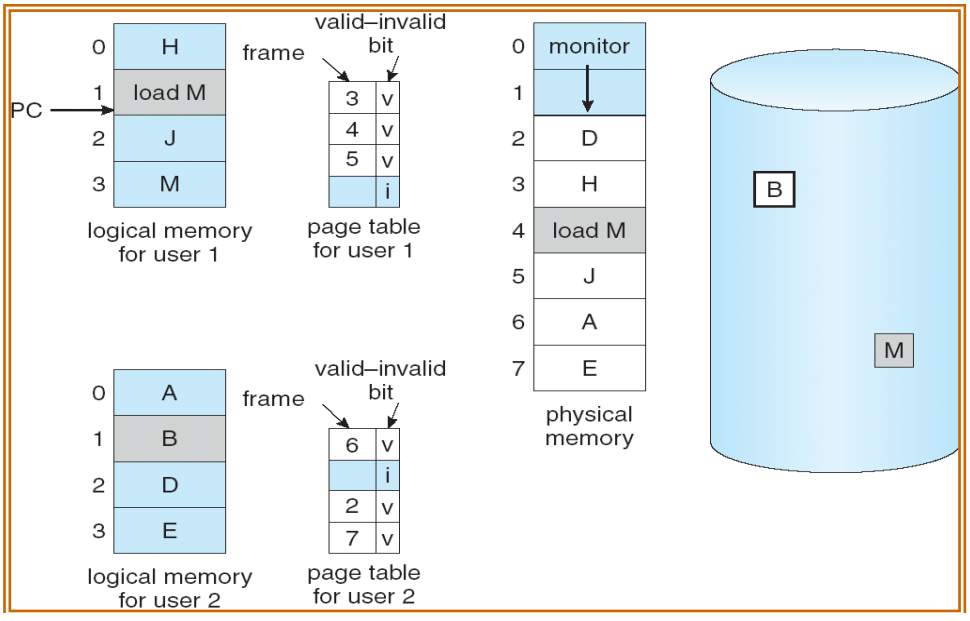




- ✓ **For Ex:** If a child process tries to modify a page containing portions of the stack, the OS recognizes them as a copy-on-write page and create a copy of this page and maps it on to the address space of the child process. So the child process will modify its copied page and not the page belonging to parent.
- ✓ The new pages are obtained from the **pool** of free pages. Operating systems allocate these pages using a **technique** known as **zero-fill-on-demand.** Zero-fill-on-demand pages have been **zeroed-out** before being allocated, thus erasing the previous contents.

4.4 Page Replacement

- ✓ If the total memory requirement exceeds physical memory,**Page replacement** policy deals with**replacing (removing) pages** frommemory to free frames for bringing inthe new pages.
- ✓ While user process is executing, a **page fault** occurs.The operating system determines where the desired page is residing on the disk, and this finds that there are no free frames on the free frame list as shown in below **figure**.

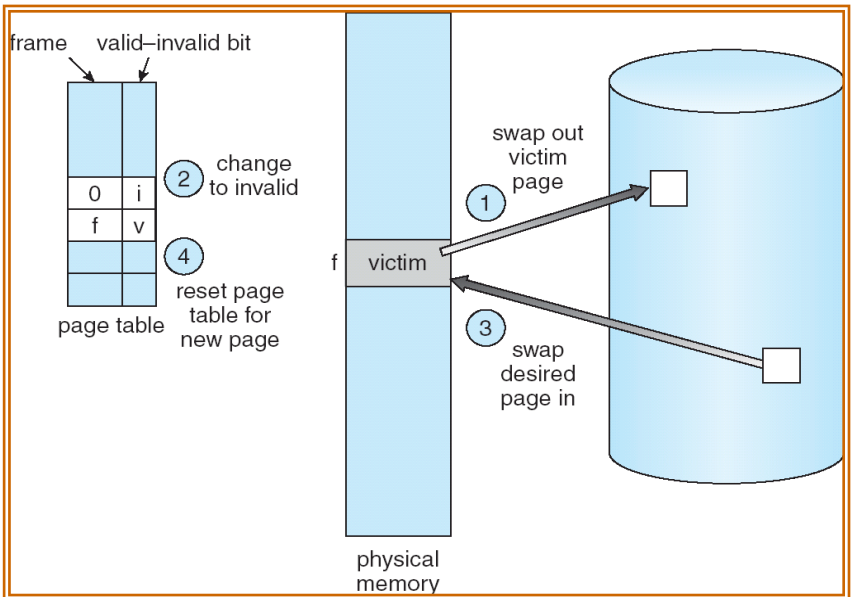


- ✓ The OS has **several options**like; it could **terminate** the user process or instead **swap out** a process, freeing all its frames and thus reduce the level of multiprogramming.

- **Basic Page Replacement**

  ✓ If frame is not free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory as shown in below **figure**. We can now use the freed frame to hold the page for which the process faulted. The **page-fault service routine**is **modified**as follows to include page replacement,

  1. Find the location of derived page on the disk.
  2. Find a free frame
     a. If there is a free frame, use it.
     b. Otherwise, use a replacement algorithm to select a victim frame.
     c. Write the victim frame to the disk; change the page and frame tables accordingly.
  3. Read the desired page into the free frame and change the page and frame tables.
  4. Restart the user process.



  ✓ If no frames are free, the **two page transfers** (one out and one in) are required. This will doubles the page-fault service time and increases the effective access time.

  ✓ This overhead can be reduced by using**modify (dirty) bit**. Each page or frame may have modify (dirty) bit associated with it. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified.

  ✓ When we select the page for replacement, we check the modify bit. If the bit is set, then the page is modified and we must write the page to the disk.

  ✓ If the bit is not set then the page has not been modified. Therefore, we can avoid writing the memory page to the disk as it is already there.

  ✓ We must solve **two major problems** to implement demand paging i.e., we must develop a **frame allocation algorithm** and a **page replacement algorithm**. If we have multiple processes in memory, we must decide how many frames to allocate

to each process and when page replacement is needed. We must select the frames that are to be replaced.

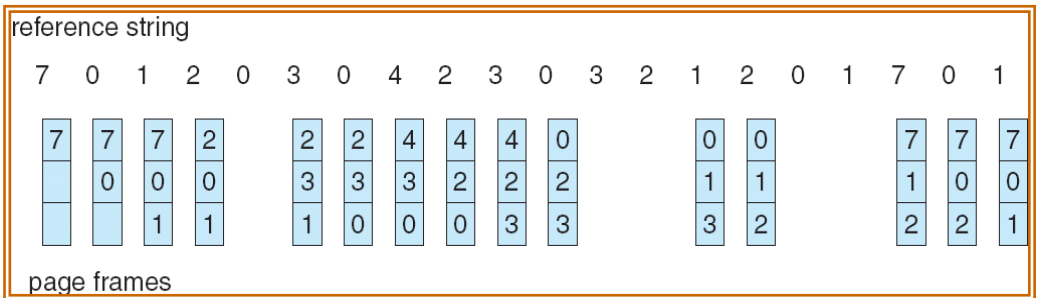
- ✓ There are **manydifferent** page-replacement algorithms. We want the one with the **lowest page-fault rate.**
- ✓ An algorithm is **evaluated** by running it on a particular string of memory references called a **reference string** and computing the **number of page faults**.

- **FIFO page replacement algorithm**

- ✓ This is the **simplest** page replacement algorithm. A FIFO replacement algorithm associates the time of each page when that page was brought into memory.
- ✓ When a page is to be replaced the oldest one is selected.
- ✓ We replace the queue at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.
- ✓ **For example**, consider the following **reference string** with **3 frames** initially empty.

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

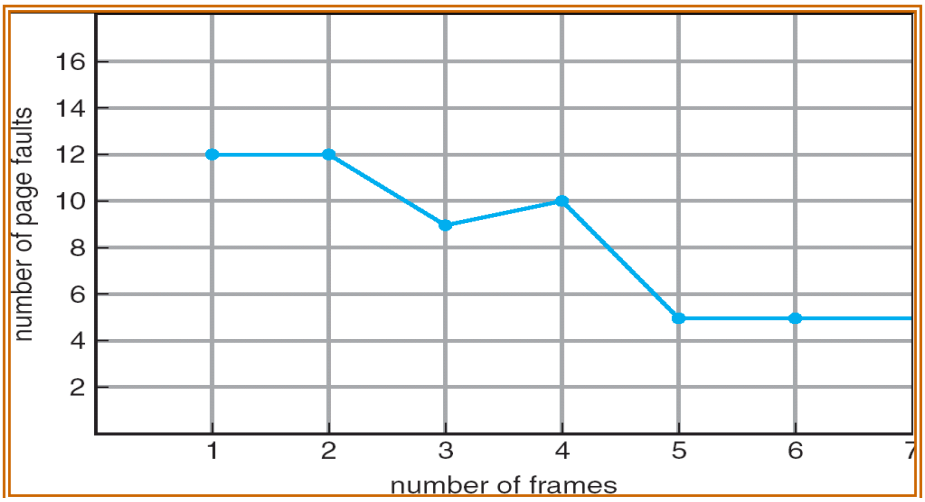
- ✓ The first three references (7,0,1) causes page faults and are brought into the empty frames.
- ✓ The next reference 2 replaces page 7 because the page 7 was brought in first.
- ✓ Since 0 is the next reference and 0 is already in memory we have no page fault for this reference.
- ✓ The next reference 3 replaces page 0 so but the next reference to 0 causer page fault. Page 1 is then replaced by page 0.
- ✓ This will continue till the end of string as shown in below **figure** and there are 15 faults all together.
- ✓



- ✓ For some page replacement algorithm, the **page fault may increase** as the number of allocated **frames increases**. This is called as **Belady'sAnamoly**. FIFO replacement algorithm may face this problem.
- ✓ To illustrate **Belady'sAnamoly** with a FIFO page-replacement algorithm, consider the following reference string.

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- ✓ The below **figure**shows the **curve of page faults** for this reference string versus the number of available frames. The number of faults for four frames (ten) is greaterthan the number of faults for three frames (nine).

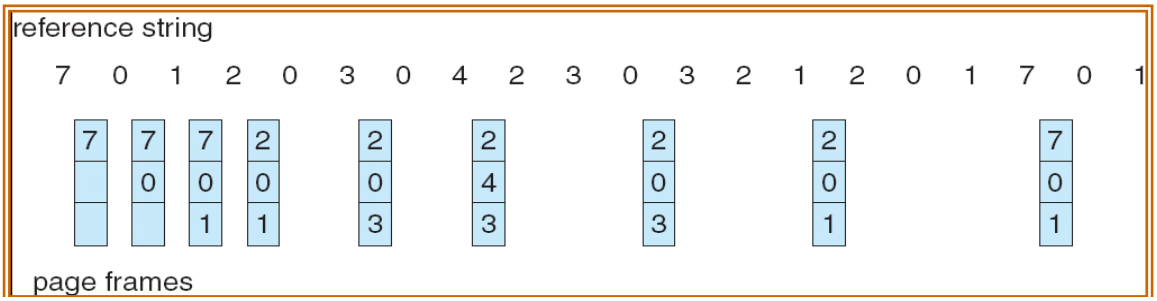


- **Optimal page replacement algorithm**

  - ✓ Optimal page replacement algorithm is mainly used to **solve** the problem of Belady'sAnamoly.
  - ✓ Optimal page replacement algorithm has the lowest page fault rate of all algorithms.
  - ✓ An optimal page replacement algorithm is also called OPT or MIN.
  - ✓ The working is simple "**Replace the page that will not be used for the longest period of time"**
  - ✓ **For example**, consider the following **reference string** with **3 frames** initially empty.

  7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

  - ✓ The first three references cause faults that fill the three empty frames.
  - ✓ The references to page 2 replaces page 7, because 7 will not be used until reference 18.
  - ✓ The page 0 will be used at 5 and page 1 at 14.This will continue till the end of the string as shown in **figure.**



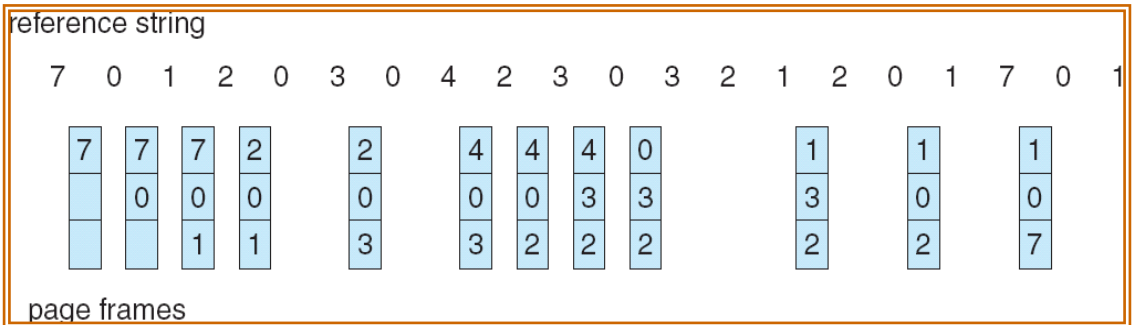
  - ✓ With only 9 page faults, optimal replacement is much better than a FIFO, which had 15 faults.
  - ✓ This algorithm is **difficult** to implement because it requires **future knowledge** of reference strings.

- **Least Recently Used (LRU) page replacement algorithm**

  - ✓ If the optimal algorithm is not feasible, an approximation to the optimal algorithm is possible.
  - ✓ The main difference between OPT and FIFO is that, FIFO algorithm uses the time when the pages was brought in and OPT uses the time when a page is to be used.
  - ✓ The LRU algorithm "**Replaces the pages that have not been used for longest period of time".**
  - ✓ The LRU associated its pages with the time of that pages last use.
  - ✓ This strategy is the optimal page replacement algorithm looking **backward in time** rather than forward.
  - ✓ **For example**, consider the following **reference string** with **3 frames** initially empty.

  7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

  - ✓ LRU replacement associates with each page the time of that page's last use.
  - ✓ When a page must be replaced LRU chooses the page that has not been used for the longest period of time.
  - ✓ The result of applying LRU replacement to our example reference string is shown in below **figure.**

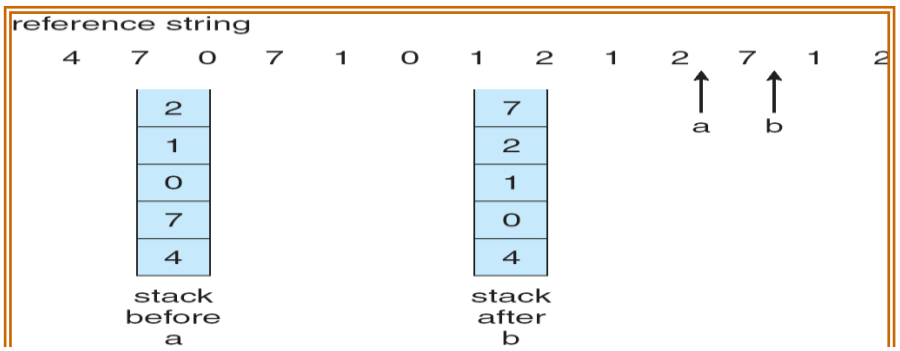


  - ✓ The first 5 faults are similar to optimal replacement.
  - ✓ When reference to page 4 occurs, LRU sees that **page 2** is used least recently. The most recently used page is page 0 and just before page 3 was used.
  - ✓ The LRU policy is often used as a page replacement algorithm and considered to be **good**.
  - ✓ Two implementations are possible,

    - **Counters:** In this we associate each page table entry a **time-of-use** field, and add to the **CPU** a logical clock or **counter**. The clock is incremented for each memory reference.When a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page table entry for that page.In this way we have the time of last reference to each page and we replace the page **with smallest time value**. The time must also be maintained when page tables are changed.

    - **Stack:** Another approach to implement LRU replacement is to keep a stack of page numbers when a page is referenced it is removed from the stack and put on to the top of stack as shown in below **figure**. In this way the top of stack is always the most recently used page and the bottom in least recently

used page. Since the entries are removed from the stack it is best implement by a doubly linked list with a head and tail pointer.Neither optimal replacement nor LRU replacement suffers from Belady'sAnamoly. These are called **stack algorithms.**



- **LRU Approximation page replacement algorithm**

  - ✓ Many systems provide hardware support in the form of a **reference bit.**
  - ✓ The reference bit for a page is **set by the hardware** whenever that page is referenced. Reference bits are associated with each entry in the page table.
  - ✓ Initially, all bits are **cleared to 0** by the operating system. As a user process executes, the bit associated with each **page is set to 1**.
  - ✓ The three **LRU Approximation page replacement algorithms** are as follows,

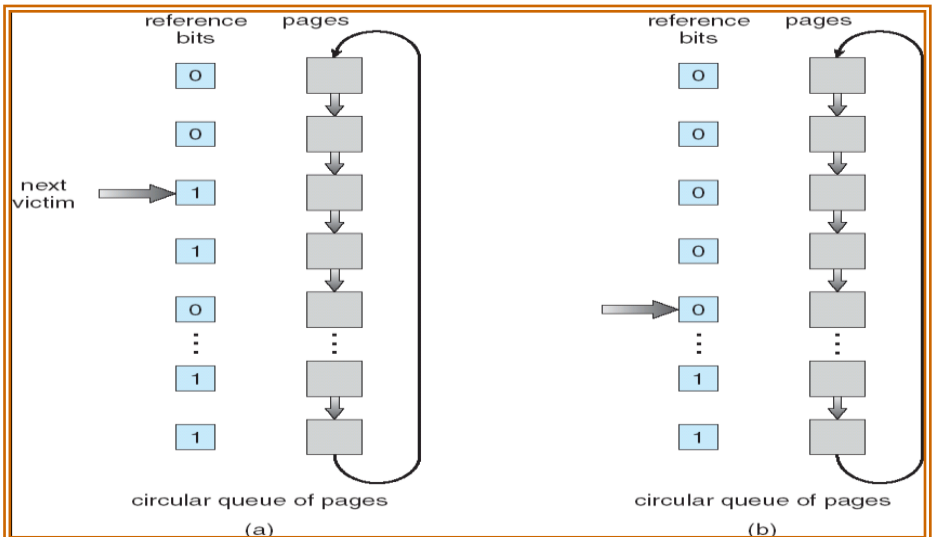
  - **Additional-Reference-Bits Algorithm**

    - ✓ We can keep an **8-bit byte** for each page in a table in memory.
    - ✓ At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit. These 8-bit shift registers contain the **history of page use for the last eight time periods**.
    - ✓ **For example**, if the shift register contains **00000000**, then the page has not been used for eight time periods; a page that is used at least once in each period has a shift register value of **11111111**. A page with a value of **11000100 has been used more recently than one with a value of 01110111** i.e., the page with the lowest number is the LRU page and it can be replaced. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the **Second-Chance Algorithm**

  - **Second-Chance Algorithm**

    - ✓ The **basic algorithm** of second-chance replacement is a **FIFO** replacement algorithm.
    - ✓ When a page has been selected we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is **set to 1,** we give the page a **second chance** and move on to select the next FIFO page.
    - ✓ When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced.

- ✓ **One way to implement** the second-chance algorithm (**clock algorithm**) is as a **circular queue**. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits as shown in below **figure**.
- ✓ Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.
- ✓ When all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.



- **Enhanced Second-Chance Algorithm**

  - ✓ We can **enhance** the second-chance algorithm by considering the **reference bit and the modify bit as an ordered pair.** With these two bits, we have four possible classes,
    - **(0, 0)**neither recently used nor modified -best page to replace.
    - **(0, 1)** not recently used but modified-not quite as good, because thepage must be written out before replacement.
    - **(1, 0)** recently used but clean-probably will be used again soon.
    - **(1, 1)** recently used and modified - probably will be used again soon, andthe page must be written out to disk before it can be replaced.
  - ✓ Each page is in one of these four classes. We replace the first page encountered in the lowest nonempty class.
  - ✓ The major **difference** between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of I/Os required.

- **Count Based Page Replacement**

  - ✓ There is many other algorithms that can be used for page replacement, we can keep a counter of the number of references that has made to a page.

    - **LFU (Least Frequently Used)**

- ✓ This causes the page with the smallest count to be replaced. The reason for this selection is that actively used page should have a large reference count.
- ✓ This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process but never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.

- **Most Frequently Used(MFU)**

  - ✓ This is based on the principle that the page with the smallest count was probably just brought in and has yet to be used.

- **Page-Buffering Algorithms**
  - ✓ Systems keep a pool of free frames and when a page fault occurs, a victim frame is chosen as before. The desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, itsframe is added to the free-frame pool.
  - ✓ An **expansion** of this idea is to maintain a list of **modified pages**. Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.
  - ✓ Another modification is to keep a pool of free frames but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it.
- **Applications and Page Replacement**
  - ✓ Applications accessing data through the operating system's virtual memory perform worse than if the operating system provided no buffering at all. An **example** is a **database**, which provides its own memory management and I/0 buffering. Applications like this understand their memory use and disk use better than an operating system that is implementing algorithms for general-purpose use.
  - ✓ In another **example, data warehouses** frequently perform massive sequential disk reads, followed by computations and writes. The LRU algorithm would be removing old pages and preserving new ones, while the application would more likely be reading older pages than newer ones. Here, MFU would be more efficient than LRU.
  - ✓ Because of such problems, some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the **raw disk**, and I/O to this array is termed **raw I/0**.

4.5 Allocation of Frames

- ✓ It is concerned with how we allocate the fixed amount of free memory among the various processes.
- ✓ The simplest case, Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. This system has 128 frames. The operating system may take 35

KB, leaving 93 frames for the user process. When the free-frame list exhaust, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminates the 93 frames would once again be placed on the free-frame list.

- **Minimum Number of Frames**

  - ✓ The strategies for the allocation of frames are constrained in various ways. We cannot allocate more than the total number of available frames. We must also allocate at least a minimum number of frames.
  - ✓ One reason for allocating at least a minimum number of frames involves performance. As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
  - ✓ Whereas the minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory.

- **Allocation Algorithms**

  - ✓ The easiest way to split **m frames** among **n processes** is to give everyone an **equal share** i.e., **m/n** frames. **For example**, if there are **93 frames** and **five processes,** each process will get **18 frames**. The **three leftover frames** can be used as a **free-frame buffer pool**. This scheme is called **equal allocation**.
  - ✓ An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a **1-KB frame size**. If a small student process of **10 KB** and an interactive database of **127 KB** are the only two processes running in a system with **62 free frames**, it does not make **much sense** to give each process **31 frames**. The student process does not need more than 10 frames, so the other 21 are wasted.
  - ✓ To **solve this problem**, we can use **proportional allocation** in which we allocate available memory to each process according to **its size**. Let the size of the virtual memory for process $\mathbf{p_i}$be $\mathbf{s_i}$, and define

$$\mathbf{S=\sum s_i}$$

  - ✓ Then, if the **total number of available frames is m**, we allocate $\mathbf{a_i}$ frames to
  - ✓ process $\mathbf{p_i}$, where $\mathbf{a_i}$ is approximately

$$\mathbf{a_i = s_i /S * m.}$$

  - ✓ We must adjust each $\mathbf{a_i}$ to be an integer that is greater than the minimum number of frames required, with a **sum not exceeding m.**
  - ✓ With **proportional allocation**, we would **split 62 frames** between **two processes**, one of 10 pages and one of 127 pages, by allocating **4 frames and 57 frames**, respectively, since

$$\mathbf{10/137 \text{ x } 62 \sim 4, \text{ and}}$$
$$\mathbf{127/137 \text{ X } 62 \sim 57.}$$

  - ✓ In this way, both processes share the available frames according to their **"needs"** rather than equally.
  - ✓ We may want to give the **high-priority process** more memory low-priority processes to speed its execution. One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the **priorities of processes** or on a **combination of size and priority.**

- **Global versus Local Allocation**

  - ✓ Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify **page-replacement algorithms** into **two broad categories**, **Global and local replacement.** The difference between them are,

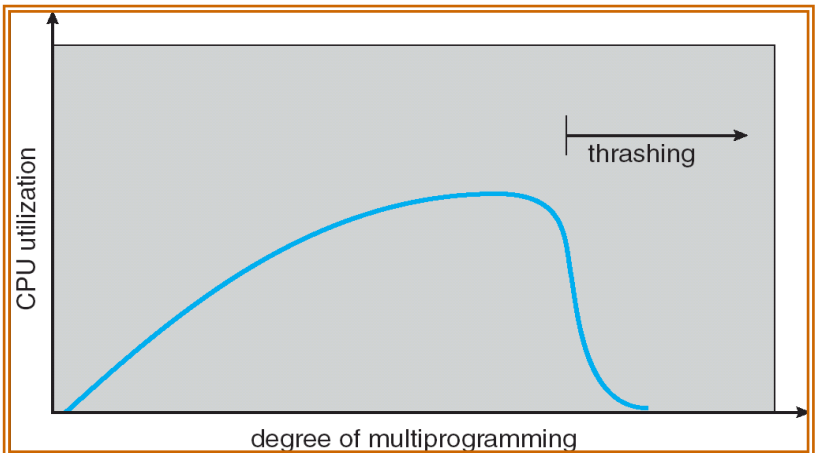
Global replacement	local replacement
Allows process to replace the frames from the set of all frames, even if that frame is allocated to other process.	Each process selects a replacement only from its own set of allocate frames.
Number of frames will change.	Number of frame does not change.
Increases system throughput	Less system throughput
Process cannot control its own page fault rate	Process can control its own page fault rate
Commonly used	Rarely used

4.6 Thrashing

- ✓ A process is **thrashing** if it is spending **more time in paging than executing**.
- ✓ If the processes do not have enough number of frames, it will quickly page fault. During this it must replace some page that is not currently in use. The process continues to fault; it quickly faults again and again, replacing pages that it must **bring back in** immediately. This high paging activity is called **thrashing.**

- **Cause of Thrashing**

  - ✓ Thrashing results in severe performance problem.
  - ✓ The operating system monitors **CPU utilization.** If it is low, we increase the degree of multiprogramming by introducing new process to the system.
  - ✓ A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong.
  - ✓ Suppose a process enters a new phase in its execution and needs more frames. It starts faulting and takes frames away from other processes. These processes need those pages, and so they also fault, taking frames from other processes.
  - ✓ These faulting processes must use the **paging device** to swap pages in and out. As they queue up for the paging device, the ready queue empties and CPU utilization decreases.
  - ✓ The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming. The new process tries to get started by taking frames from other processes, causing **more page faults and a longer queue** for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more.But this causes**thrashing** and the CPU utilization drops sharply as shown in below**figure**.



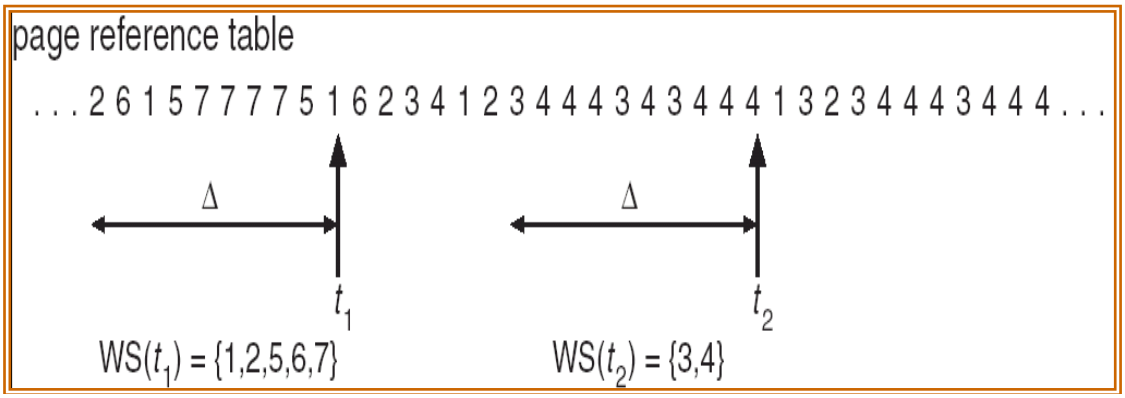
- ✓ We can limit the effect of thrashing by using a **local replacement algorithm**. To prevent thrashing, we must provide a process as many frames as it needs.
- ✓ But how do we know how many frames it "needs"? There are **several techniques.**
- ✓ The **working-set strategy** starts by looking at how many frames a process is actually using. This approach defines the locality of process execution.
- ✓ The **locality model** states that, as a process executes, it moves from locality to locality. A **locality** is a **set of pages that are actively used**.

- **Working set model**

- ✓ The **working set model** is based on the assumption of locality.
- ✓ This model uses a **parameter** Δ to define the **working set window**.
- ✓ The idea is to examine the most recent Δ **page** references. The set of pages in the most recent Δ page references is the working set as shown in below **figure**.
- ✓ If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. Thus, the working set is an approximation of the program's locality.
- ✓ **For example,** the sequence of memory references is as shown in below **figure**. If Δ = 10 memory references, then the working set at time $\mathbf{t_1}$ is {1, 2, 5, 6, 7}. By time$\mathbf{t_2}$, the working set has changed to {3, 4}.
- ✓ The accuracy of the working set depends on the selection of Δ. The most **important property** of the working set is its **size**. If we compute the working-set size,$\mathbf{WSS_i}$, for each process in the system, we can then consider that

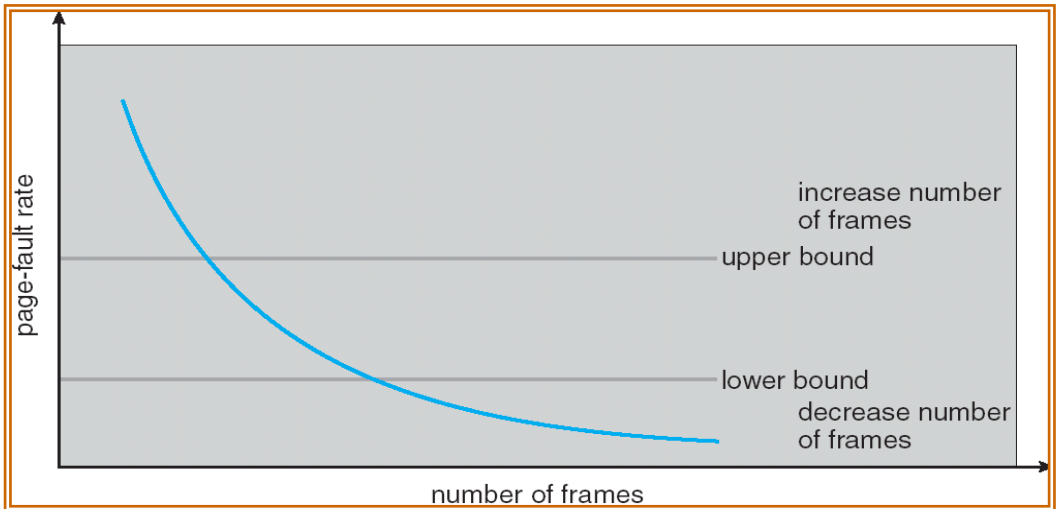
$$\mathbf{D=\sum WSS_i}$$

- ✓ Where **D** is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs $\mathbf{WSS_i}$ frames. If the total demand is greater than the total number of available frames **(D > m)**, thrashing will occur, because some processes will not have enough frames.
- ✓ Working set **prevents** thrashing by keeping the degree of multiprogramming as high as possible. Thus it optimizes the CPU utilization.
- ✓ The main **disadvantage** of this model is keeping track of the working set. The working-set window is a **moving window**. At each memory reference, a new reference appears at one end and the oldest reference drops off the other end. A page is in the working set if it is referenced anywhere in the working-set window.



- **Page-Fault Frequency**

  - ✓ The working-set model is a **clumsy** way to control thrashing. A strategy that uses **Page Fault Frequency(PFF)** takes a more **direct approach**.
  - ✓ When page fault rate is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames.
  - ✓ We can establish **upper and lower bounds** on the desired page-fault rate as shown in below **figure**. If the actual page-fault rate exceeds the upper limit, we allocate the process another frame; if the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.
  - ✓ As with the working-set strategy, we may have to **suspend a process.** If the page-fault rate increases and no free frames are available, we must select some process and suspend it. The freed frames are then distributed to processes with high page-fault rates.



4.7 File System

- ✓ The file system is the most visible aspect of an operating system. It provides the **mechanism for on-line storage** of and access to both data and programs of the operating system and all the users of the computer system.
- ✓ The file system consists of **two distinct parts,** a collection of **files** each storing related data, and a **directory structure** which organizes and provides information about all the files in the system.

4.8 File Concept

- ✓ Computers can store information on various storage media.
- ✓ Operating system provides a uniform logical view of information storage. This **logical storage unit** is called as a **file**.
- ✓ Files are **mapped by operating system** onto physical devices. These storage devices are **nonvolatile**, so contents are **persistent** through power failures and system reboots.
- ✓ **File** is a named collection of **related information** that is recorded on secondary storage.
- ✓ Files represent both the **program and the data**. Data can be numeric, alphanumeric, alphabetic or binary.
- ✓ Many different types of information like source programs, object programs, executable programs, numeric data, payroll recorder, graphic images, and sound recordings and so on can be stored on a file.
- ✓ A file has a **certain defined structures** according to its type.
  - **Text file:** Text file is a sequence of characters organized in to lines.
  - **Object file:**Object file is a sequence of bytes organized in to blocks understandable by the systems linker.
  - **Executable file:** Executable file is a series of code section that the loader can bring in to memory and execute.
  - **Source File:** Source file is a sequence of subroutine and function, each of which are further organized as declaration followed by executable statements.

- **File Attributes**

  - ✓ File attributes varies from one OS to other. The common file attributes are,
    - **Name:** The symbolic file name is the only information kept in human readable form.
    - **Identifier:** The unique tag, usually a number, identifies the file within the file system. It is the non-readable name for a file.
    - **Type:** This information is needed for systems that support different types.
    - **Location:** This information is a pointer to a device and to the location of the file on that device.
    - **Size:** The current size of the file and possibly the maximum allowed size are included in this attribute.
    - **Protection:** Access control information determines who can do reading, writing, execute and so on.

- **Time, data and User Identification:** This information must be kept for creation, last modification and last use. These data are useful for protection, security and usage monitoring.

- **File Operations**

  - ✓ File is an abstract data type. To define a file we need to consider the operation that can be performed on the file.
  - ✓ Basic operations of files are,
    - **Creating a file:Two steps** are necessary to create a file. First,**space** in the file system for file is found. Second, an **entry** for the new file must be made in the directory.
    - **Writing a file:System call** is mainly used for writing in to the file. System call specifies the **name,** of the file and the **information** to be written to the file. Given the name the system search the entire directory for the file. The system must keep a write pointer to the location in the file where the next write to be taken place.
    - **Reading a file:** To read a file, system call is used. It requires the **name** of the file and the **memory address** from where the next block of the file should be put. Again, the directory is searched for the associated directory and system must maintain a read pointer to the location in the file where next read is to take place.
    - **Delete a file:** System will search the directory for the file to be deleted. If entry is found it releases all free space. That free space can be reused by another file.
    - **Truncating the file:** User may want to erase the contents of the file but keep its attributes. Rather than forcing the user to delete a file and then recreate it, truncation allows all attributes to remain unchanged except for file length.
    - **Repositioning within a file:** The directory is searched for appropriate entry and the current file position is set to a given value. Repositioning within a file does not need to involve actual I/O. This file operation is also known as **seek.**
  - ✓ In addition to this basis 6 operations the **other two operations** include **appending** new information to the end of the file and **renaming** the existing file.
  - ✓ Most of the file operation involves searching the entire directory for the entry associated with the file. To avoid this, OS keeps a **small table**called the **open-file table** containing information about all **open files.** When a file operation is requested, the file is specified via index into this table. So searching is not required.
  - ✓ Several **piece of information** are associated with an **open file**,
    - **File pointer:** on systems that does not include offset as part of the read and write system calls, the system must track the last read-write location as current file position pointer. This pointer is unique to each process operating on a file.
    - **File open count:** It keeps the count of open files. As the files are closed, the OS must reuse its open file table entries, or it could run out of space in the table. Because multiple processes may open a file, the system must wait for the last file to close before removing the open file table entry. The counter tracks the number of copies of open and closes and reaches zero to last close.
    - **Disk location of the file:** The information needed to locate the file on the disk is kept in memory to avoid having to read it from the disk for each operation.

- **Access rights:** Each process opens a file in an access mode. This information is stored on per-process table, then OS can allow or deny subsequent I/O request.

✓ Some operating systems provide facilities for **locking an open file (or sections of a file)**. File locks allow one process to lock a file and prevent other processes from gaining access to it.
✓ File locks are useful for files that are shared by several processes.
✓ A **shared lock** is similar to a **reader lock** in that several processes can acquire the lock concurrently. An **exclusive lock** behaves like a **writer lock**which means only one process at a time can acquire such a lock.
✓ Operating systems provide both types of locks, but some systems provide only exclusive file locking.
✓ Operating systems may also provide either **mandatory or advisory** file-locking mechanisms. If a lock is mandatory, then once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file. For advisory locking, it is up to software developers to ensure that locks are appropriately acquired and released.
✓ Windows operating systems adopt mandatory locking, and UNIX systems employ advisory locks.

- **File Types**

✓ File type is included as a part of the filename. File name is split into **two parts-** a **name** and **extension** separated by a period character.
✓ The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.
✓ **For example,** only a file with a .com, .exe, or .bat extension can be executed. The **.com** and **.exe** files are two forms of binary executable files, whereas **.bat** file is a **batch file** containing commands in ASCII format to the operating system.
✓ The table shown in below **figure** gives the file type with extension and function.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

- **File Structure**

  - ✓ File types can be used to indicate the internal structure of the file. Certain files must match to a required structure that is understood by the operating system.
  - ✓ **For example,** the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is.
  - ✓ Some operating systems extend this idea into a **set of system-supported file structures,** with sets of special operations for manipulating files with those structures. This is one disadvantage where the resulting size of the operating system is cumbersome.
  - ✓ Some operating systems impose and support a **minimal number** of file structures.
  - ✓ Too few structures make programming inconvenient, whereas too many cause operating-system to expand and makes programmer to confuse.

- **Internal File Structure**

  - ✓ Locating an **offset** within a file can be complicated for the operating system.
  - ✓ Disk systems have a well-defined block size determined by the size of a sector. All disk I/0 is performed in units of one block and all blocks are the same size. It is not sure that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length.
  - ✓ **Packing** a number of logical records into physical blocks is a common solution to this problem. The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered a sequence of blocks.
  - ✓ All the basic I/O functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem.
  - ✓ Because disk space is always allocated in blocks, some portion of the last block of each file is wasted. The waste incurred to keep everything in units of blocks is **internal fragmentation.**
  - ✓ All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

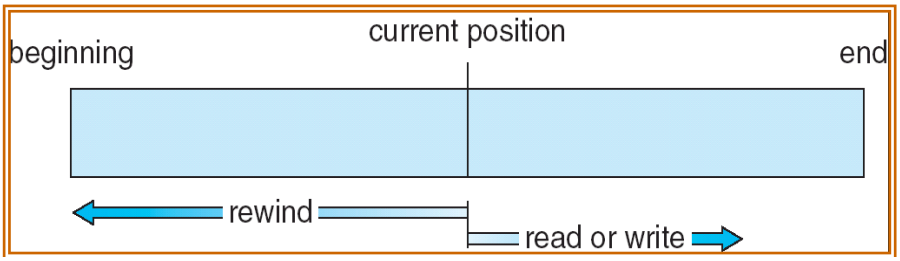
4.9 Access Methods

The information in the file can be accessed in several ways. The different file access methods are,

- **Sequential Access**

  - ✓ Sequential access is the **simplest** access method. Information in the file is processed in order, one record after another. **Editors and compilers** access the files in this fashion.
  - ✓ A read operation **read next** reads the next portion of the file and automatically advances a file pointer, which tracks next I/O location. The write operation **write next** appends to the end of the file and advances to the end of the newly written material.

- ✓ Such a file can be reset to the beginning and on some systems a program may be able to skip forward or backward **n** records for some integer n, where **n**= 1.
- ✓ Sequential access, which is depicted in below **figure** is based on a **tape model** of a file and works well on sequential-access devices as it does on random-access ones.



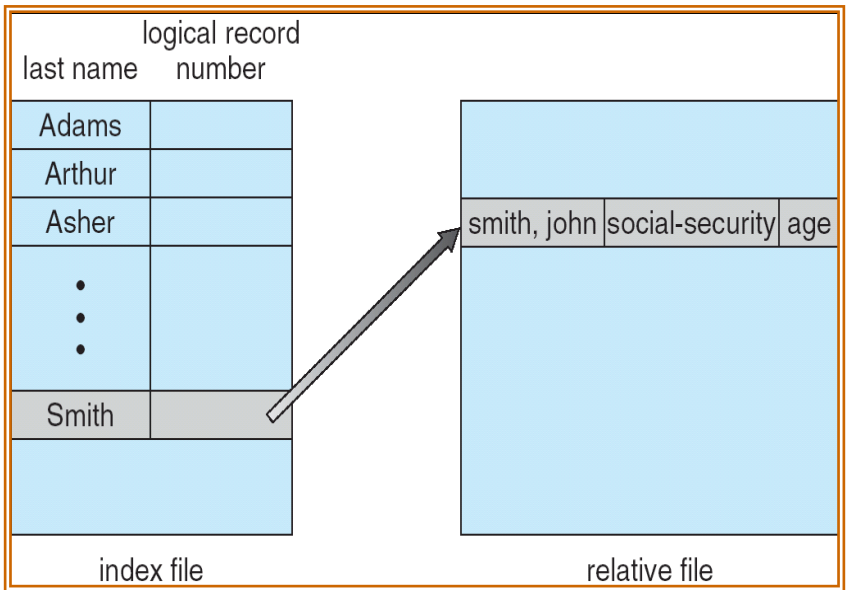
- **Direct Access (Relative Access)**

- ✓ A file is made up of fixed length logical records. It allows the program to read and write records rapidly in any order.
- ✓ Direct access allows **random access** to any file block. This method is based on **disk model** of a file.
- ✓ For direct access, the file is viewed as a **numbered sequence of blocks or records.** Thus, we may read block 14, then read block 53, and then write block 7.
- ✓ The direct access method is suitable for searching the records in large amount of information. **For example,** on an **airline-reservation system,** we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file.
- ✓ The **file operations** must be modified to include the block number as a parameter. Thus, we have **read n**, where n is the block number, rather than read next, and **write n** rather than write next.
- ✓ The block number provided by the user to the operating system is normally a **relative block number.** A relative block number is an index to the beginning of the file.
- ✓ We can easily **simulate** sequential access on a direct-access file by simply keeping a **variable cp** that defines **current position**, as shown in below **figure,** where as simulating a direct-access file on a sequential-access file is extremely inefficient and clumsy.

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

- **Other Access Methods**

  - ✓ These methods generally involve the **construction of an index** for the file.
  - ✓ The index is like an index at the end of a book which contains pointers to various blocks.
  - ✓ To find a record in a file, we search the index and then use the pointer to access the file directly and to find the desired record.
  - ✓ With large files index file itself can be very large to be kept in memory. One **solution** is to create an index for the index files itself. The primary index file would contain pointer to secondary index files which would point to the actual data items.
  - ✓ **For example**, IBM's **indexed sequential-access method (ISAM)** uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a **defined key**. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. The below **figure** shows a similar situation as implemented by VMS index and relative files.



4.10 Directory Structure

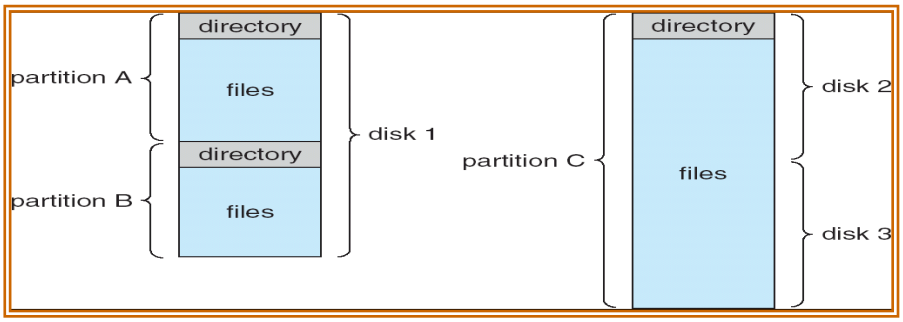
- ✓ The files systems can be very large. Some systems stores millions of files on the disk. To manage all this data we need to organize them. This organization involves the use of **directories.**

- **Storage structure**

  - ✓ A diskcan be used completely for a file system. Sometimes it is desirable to place multiple file systems on a disk or to use parts of the disk for a file system. These parts are known as **partitions, slices or minidisks**.
  - ✓ These parts can be combined to form larger structures known as **volumes.**

- ✓ Each volume contains information about files within it. This information is kept in entries in a **device directory (directory) or volume table of contents.**
- ✓ The **directory** records the information such as name, location, size, type for all files on that volume.

The below **figure** shows a typical file-system organization.



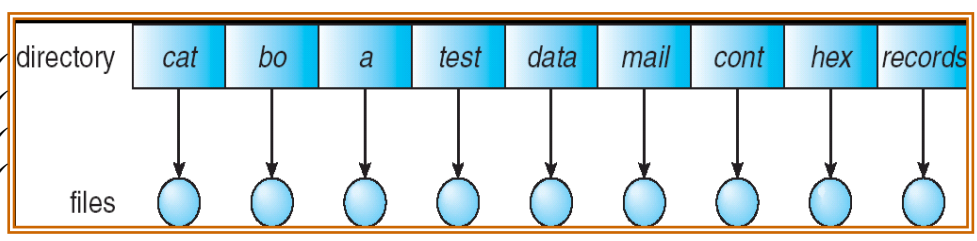
- **Directory Overview**

- ✓ The directory can be viewed as a **symbol table** that translates the file names into their directory entries. The directory itself can be organized in many ways.
- ✓ When considering a particular directory structure, several operations can be performed on a directory.
  - **Search for a file:** Directory structure is searched for finding particular file in the directory. Files have symbolic names and similar names may indicate a relationship between files, we must be able to find all the files whose name matches a particular pattern.
  - **Create a file:** New files can be created and added to the directory.
  - **Delete a file:** when a file is no longer needed, we can remove it from the directory.
  - **List a directory:** We need to be able to list the files in directory and the contents of the directory entry for each file in the list.
  - **Rename a file:** Name of the file must be changeable, when the contents of the file are changed. Renaming allows the position within the directory structure to be changed.
  - **Traverse the file system:** It should be possible to access any file in the file system. It is always good to keep the backup copy of the file so that it can be used when the system fails or when the file system is not in use.

- ✓ There are **different** types of **logical structures of a directory** as discussed below.

- **Single-level directory**

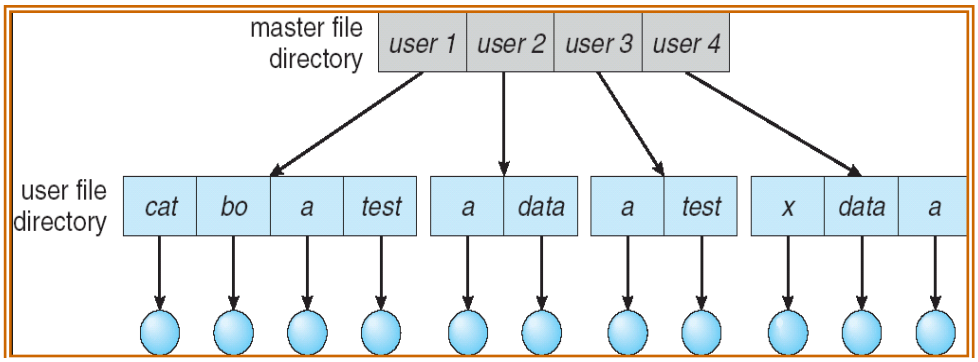
- ✓ This is the simplest directory structure. All the files are contained in the same directory which is easy to support and understand as shown in below **figure**



- ✓ The disadvantages are,
  - Not suitable for a large number of files and more than one user.
  - Because of single directory files, files require **unique file names**.
  - Difficult to remember names of all the files as the number of files increases.

- **Two-level directory**

  - ✓ A single level directory often leads to the confusion of file names between different users. The solution here is to create **separate directory for each user**.
  - ✓ In two level directories each user has its own directory. It is called **User File Directory (UFD)**. Each UFD has a similar structure, but lists only the files of a single user.
  - ✓ When a user job starts or users logs in, the system's **Master File Directory (MFD)** is searched. The MFD is indexed by the user name or account number and each entry points to the UFD for that user as shown in below **figure**
  - ✓

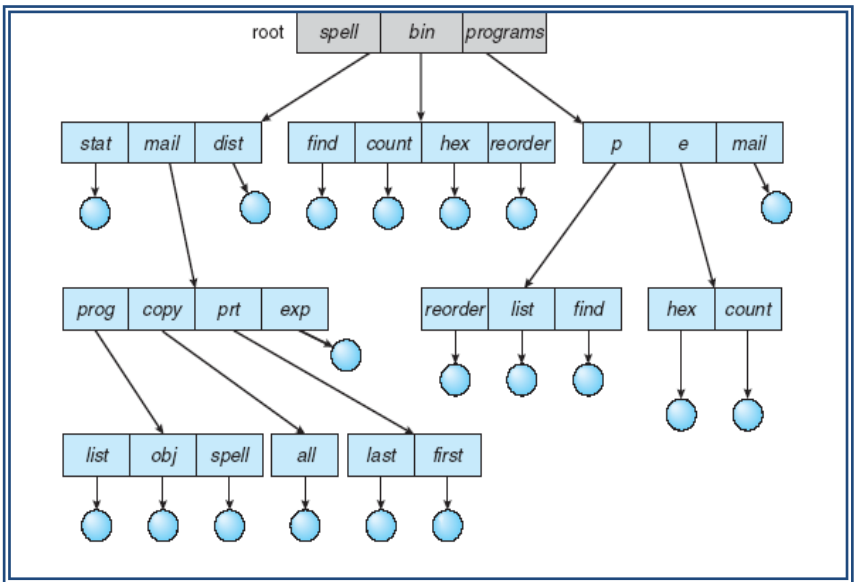


  - ✓ When a user refers to a particular file, only his own UFD is searched. Thus different users may have files with the same name.
  - ✓ To create a file for a user, OS searches only that user UFD to check whether another file of that name exists.
  - ✓ To delete a file OS checks in the local UFD, so that it cannot accidentally delete another user's file with the same name.
  - ✓ Although two-level directories **solve the name collision problem** but it still has some **disadvantages**.
  - ✓ This structure isolates one user from another and this isolation is an advantage when the users are independent, but disadvantage when some users want to co-operate on some task and to access one another's file.
  - ✓ If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file we must give both the user name and the file name.
  - ✓ A two-level directory is like a **tree, or an inverted tree of height 2.**
    The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files. The **files** are the leaves of the **tree**. A user name and a file name define a **path name.**
  - ✓ To access the **system files** the appropriate commands are given to the operating system and these files are read by the loader and executed. This file name would

be searched in the current UFD. The sequence of directories searched when a file is named is called the **search path.**

- **Tree-structured directories**

  ✓ The two level directory structures can be extended to a tree of **arbitrary height** as shown in **figure.**



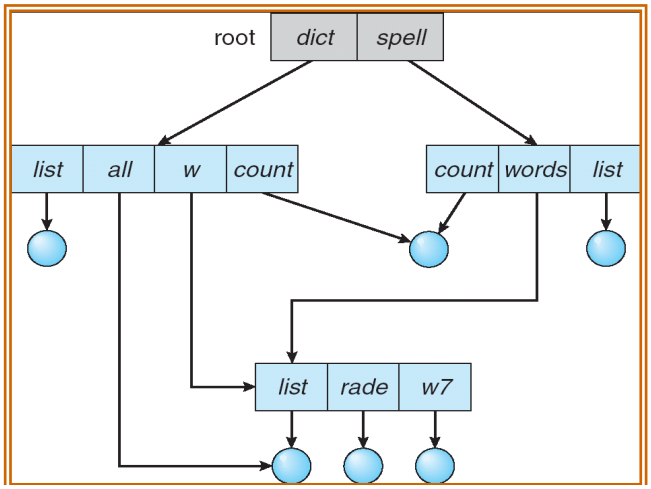
  ✓ It allows users to **create their own subdirectories** and to organize their files accordingly. A subdirectory contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way.
  ✓ The entire directory will have the same internal format. **One bit** in each entry defines the entry as a **file (0)** and as a **subdirectory (1)**. Special system calls are used to create and delete directories.
  ✓ Each process has a **current directory** and it should contain most of the files that are of the current interest to the process. When a reference is made to a file the current directory is searched. If the needed file is not in the current directory then the user must specify the path name or change the current directory.

  ✓ Path name can be of two types.
    - **Absolute path name:** Begins at the root and follows a path down to the specified file, giving the directory names on the path.
    - **Relative path name:** Defines a path from the current directory.
  ✓ One important task is how to handle the deletion of a directory. If a directory is empty, its entry can simply be deleted. If a **directory is not empty, one of the two approaches** can be used.
    - In MS-DOS, the directory is not deleted until it becomes empty.
    - In UNIX, **rm** command is used, where all directory's files and subdirectories are also deleted before deleting a directory.
  ✓ With a tree-structured directory system, users can be allowed to accesstheir files, and also the **files of other users**. Alternatively **user can change the current directory** to other user's directory and access the file by its file names.

- ✓ A **path** to a file in a tree-structured directory can be **longer** than a path in a two-level directory.

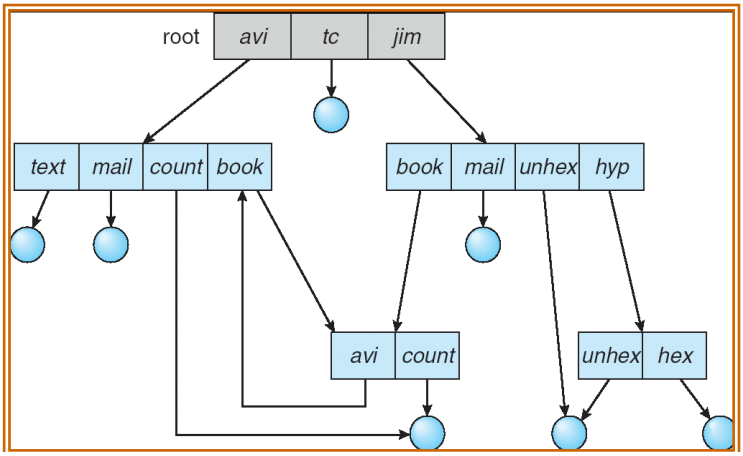
- **Acyclic graph directories**

  - ✓ A tree structure prohibits the sharing of files or directories.
  - ✓ An acyclic graph that is, a **graph with no cycles** allows directories to **share subdirectories and files** as shown in below **figure.** The same file or subdirectory may be in two different directories.



  - ✓ The acyclic graph is a natural generalization of the tree-structured directory scheme. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other.
  - ✓ **Sharing** is mainly **important for subdirectories**; a new file created by one person will automatically appear in all the shared subdirectories.
  - ✓ Shared files and subdirectories can be **implemented** by using **links.** A link is a **pointer** to another file or a subdirectory. **Another approach** to implementing shared files is simply to **duplicate all information** about them in both sharing directories. Thus, both entries are identical and equal.
  - ✓ An acyclic graph directory structure is **more flexible** than a simple tree structure but sometimes it is **more complex.**
  - ✓ Some of the **problems** are, a file may now have **multiple absolute path names** and distinct file names may refer to the same file.
  - ✓ **Another problem** involves is in **deletion**. There are **two approaches** to decide when the space allocated to a shared file can be deallocated and reused.
  - ✓ **One possibility is to remove the file** whenever anyone deletes it, but this action may leave dangling pointers to the nonexistent file.
  - ✓ **Another approach is to preserve the file** until all references to it are deleted. To **implement this approach**, we could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the **file-reference list.** When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its **file-reference list is empty**. The **trouble with this approach** is the variable and potentially large size of the file-reference list. So we need to keep only the **count of number of references,** and when the **count is 0,** the file can be deleted.
  - ✓ This is done through hard link count in UNIX OS.

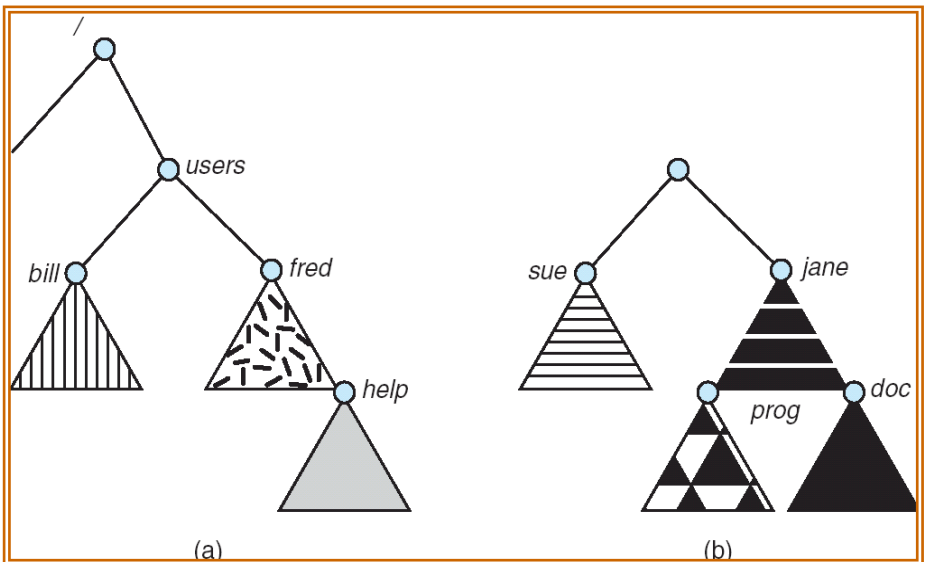
- **General Graph Directory**
  - ✓ The **problem with using an acyclic-graph structure** is ensuring that thereare no cycles. When we add links, the tree structure is destroyed, resulting in a simple graph structure as shown in below **figure.**
  - ✓ The **advantage of an acyclic graph** is the **simplicity** of the algorithms to traverse the graph and to determine when there are **no more references** to a file.



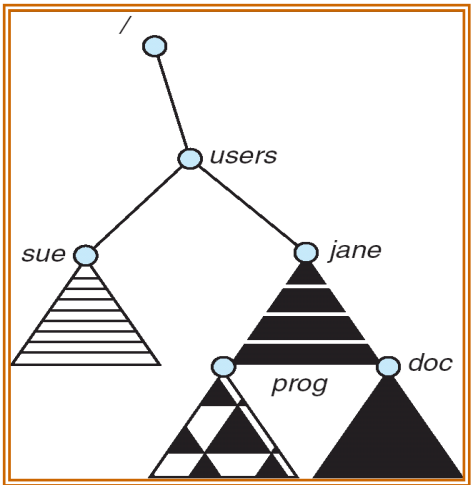
  - ✓ A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One **solution** is to limit the number of directories that will be accessed during a search.
  - ✓ A **problem exists** when we are trying to determine when a file can be **deleted.** When cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. So a**garbage-collection scheme** is used.
  - ✓ Garbage collection involves **traversing** the entire file system, **marking everything that can be accessed** but it is **time consuming**. Thus, an **acyclic-graph structure is much easier** to work than general graph structure. The **difficulty** is to **avoid cycles** as new links are added to the structure. There are algorithms to detect cycles in graphs but they are **computationally expensive**, hence **cycles must be avoided.**

4.11 File System Mounting

- ✓ The file system must be mounted before it can be available to processes on the system
- ✓ The **procedure** for mounting the file is as follows:
  - o The OS is given the name of the device and the location within the file structure at which to attach the file system (**mount point**).A mount point will be an **empty directory** at which the mounted file system will be attached.**For example:**On UNIX, a file system containing **user's home directory** might be mounted as **/home** then to access the directory structure within that file system we must precede the directory names as **/home/Jane.**
  - o Then OS verifies that the device contains this valid file system. OS uses device drivers for this verification.
  - o Finally the OS mounts the file system at the specified mount point.
- ✓ Consider the file system depicted in **figure** below, where the triangles represent sub trees of directories. **Figure(a)**shows an existing file system, while **figure (b)** shows an un-mounted volume residing on **/device/dsk**. At this point, only the files on the existing file system can be accessed.



- ✓ The below **Figure** shows the effects of mounting the volume residing on **/device/dsk** over/**users.**



- ✓ Windows operating systems automatically discover all devices and mount all located file systems at boot time. In UNIX systems, the mount commands are explicit.
- ✓ A system configuration file contains a list of devices and mount points, for automatic mounting at boot time, but other mounts may be executed manually.

4.12 File Sharing

- **Multiple Users**
  - ✓ Given a directory structure that allows files to be shared by users, the operating system must mediate the file sharing.
  - ✓ The system either can allow a user to access the files of other users by default or it may require that a user specifically grant access to the files.
  - ✓ To implement sharing and protection, the system maintain more file and directory attributes than on a single user system, most systems support the concept of file **owner and group.**

- ✓ When a user requests an operation on a file, the user ID can be compared to the owner attribute to determine if the requesting user is the owner of the file. Likewise the group ID's can be compared. The result indicates which permissions are applicable.

- **Remote File Systems**

  - ✓ Remote sharing of the file system is implemented by using **network.**
  - ✓ **Network** allows the sharing of resources. **File Transfer Protocol (FTP)** is one of the methods used for remote sharing. Other methods are **distributed file system(DFS) and World Wide Web**.
  - ✓ DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files. **This integration adds complexity as discussed in below DFSs.**

    - **Client-server Model**

      - ✓ System containing the files is the server and the system requesting access to the files is a client. Files are specified on a partition or subdirectory level. A server can serve multiple clients and a client can use multiple servers.
      - ✓ A client can be specified network name or other identifier, such as an IP address, but these can be **spoofed** or imitated. As a result of spoofing, an unauthorized client could be allowed to access the server. More secure solutions include secure authentication of the client by using **encrypted keys**.

    - **Distributed Information systems**

      - ✓ For managing client server services, distributed information system is used to provide a unified access to the information needed for remote computing. **UNIX** systems have a wide variety of distributed information methods. The **domain name system (DNS) provides host-name-to-network-address translations** for the entire internet.
      - ✓ Before DNS became widespread, files containing the same information were sent via e-mail or ftp between all networked hosts. This methodology was not scalable.
      - ✓ Sun Microsystems introduced **yellow pages** which are called as **Network Information Service (NIS).** It centralizes the storage of user names, host names, printer information, and others. But it uses unsecure authentication methods, like sending user passwords unencrypted (in clear text) and identifying hosts by IP address.
      - ✓ **NIS+** is a much more secure replacement for NIS but is also much more complicated and has not been widely adopted.
      - ✓ Microsoft introduced **Common Internet File System (CIFS).** Here network information is used in conjunction with user authentication (user name and password) to create a **network login** that the server uses to decide whether to allow or deny access to a requested file system. For this authentication to be valid, the user names must match between the machines.

- ✓ Microsoft uses **two distributed naming structures** to provide a single name space for users. The older naming technology is**domains.** The newer technology, available in Windows XP and Windows 2000, is **active directory.**
- ✓ **Light Weight Directory Access Protocol (LDAP)** introduced bySun Microsystems is a secure **distributed naming mechanism.** It is a **secure single sign-on** for users, who would enter their authentication information once for access to all computers within the organization. This **reduces system-administration efforts** by combining the information that is currently scattered in various files on each system or in different distributed information services.

- **Failure Modes**

  - ✓ **Local file systems can fail** for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure or other **disk-management information (metadata)** disk-controller failure, cable failure, and host-adapter failure.
  - ✓ User or system-administrator failure can also cause files to be lost or entire directories or volumes to be deleted. Many of these failures will cause a host to crash and an error condition to be displayed, and human intervention will be required to repair the damage.
  - ✓ **Remote file systems have even more failure modes.** Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems.
  - ✓ In the case of networks, the network can be interrupted between two hosts. Such interruptions can result from hardware failure, poor hardware configuration, or networking implementation issues.
  - ✓ To recover from failure, some kind of **state information** may be maintained on both the client and the server.
  - ✓ The **Networked File System (NFS)** takes a simple approach, implementing a stateless **Distributed File System (DFS)**. It assumes that a client request for a file read or write would not have occurred unless the file system had been **remotely mounted** and the file had been previously open. The NFS protocol carries all the information needed to locate the appropriate file and perform the requested operation.

- **Consistency Semantics**

  - ✓ **Consistency Semantics** represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system access a shared file simultaneously and they specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.
  - ✓ A **series of file accesses (**that is, reads and writes) attempted by a user to the same file is always enclosed between the open () and close () operations. The series of accesses between the open () and close () operations makes up a **file session.**

    - **examples** of consistency semantics are **UNIX Semantics**

- ✓ The UNIX file system uses the **following consistency semantics**,
  - Writes to an open file by a user are visible immediately to other users who have opened their file.
  - One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.
- ✓ In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image causes delays in user processes.

- **Session Semantics**

- ✓ The **Andrew file system (AFS)** uses the following **consistency semantics,**
  - Writes to an open file by a user are not visible immediately to other users that have the same file already opened.
  - Once a file is closed, the changes made to it are visible only in sessions starting later. Already opened instances of the file do not reflect these changes.
- ✓ According to these semantics, a file may be associated temporarily with several images at the same time. Multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay.

- **Immutable-Shared-Files Semantics**

- ✓ A unique approach here is that **immutable shared files,**once a file is declared as shared by its creator, it cannot be modified. An immutable file has **two key properties** i.e., its name may not be **reused,** and its contents may not be **altered**.

4.13 Protection

- ✓ Information must be protected from a physical damage and improper access i.e., reliability and protection.
- ✓ Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the external storage devices and locking them in a desk drawer or file cabinet.
- ✓ In a multiuser system other mechanisms are needed.

- **Types of Access**

- ✓ Protection mechanisms provide **controlled access** by limiting the types of file access that can be made. Access is **permitted or denied** depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled. They are ,
  - **Read -**Read from the file.
  - **Write -** Write or rewrite the file.
  - **Execute -** Load the file into memory and execute it.
  - **Append -** Write new information at the end of the file.

- **Delete -** Delete the file and free its space for possible reuse.
- **List -**List the name and attributes of the file.

✓ Other operations, such as renaming, copying, and editing the file, may also be controlled. For many systems these higher-level functions may be implemented by a system program that makes lower-level system calls.

- **Access Control**

✓ Different users may need different types of access to a file or directory.

✓ The most general scheme to implement identity dependent access is to associate with each file and directory an **Access Control List (ACL)** specifying user names and the types of access allowed for each user.

✓ When a user requests access to a particular file, the operating system checks the **access list** associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

✓ This approach has the **advantage** of enabling complex access methodologies.

✓ The main **problem** with access lists is their **length**. If we want to allow everyone to read a file, we must list all users with read access. This technique has **two undesirable consequences**,

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.

✓ These problems can be **resolved** by use of a **condensed version** of the access list.

✓ To condense the length of the access-control list, many systems recognize **three classifications of users** in connection with each file as follows,

- **Owner**. The user who created the file is the owner.
- **Group.** A set of users who are sharing the file and need similar access is a group, or work group.
- **Universe.** All other users in the system constitute the universe.

- **Other Protection Approaches**

✓ Another approach to the protection problem is to associate a **password** with each file. If the passwords are chosen randomly and changed often, this scheme may be effective.

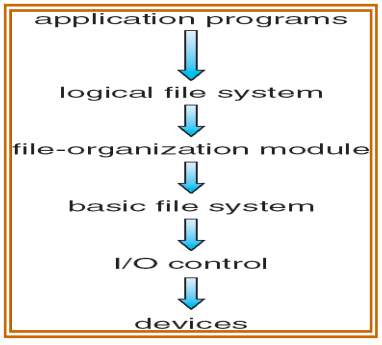
✓ The use of passwords has a few **disadvantages. First,** the number of passwords that a user needs to remember may become large. **Second,** if only one password is used for all the files, then once it is discovered, all files are accessible.

- ✓ Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to deal with this problem.
- ✓ In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories. We want to control the creation and deletion of files in a directory.

File System Implementation

4.14 File System Structure

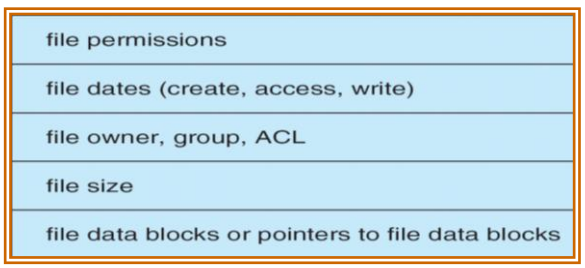
- ✓ Disks provide bulk of secondary storage on which the file system is maintained. Disks have **two characteristics:**
  - They can be rewritten in place i.e., it is possible to read a block from the disk to modify the block and to write back in to same place.
  - They can access any given block of information on the disk. Thus it is simple to access any file either sequentially or randomly and switching from one file to another.
- ✓ To provide efficient and convenient access to the disks, OS imposes one or more **file system** to allow the data to be **stored, located and retrieved easily**.
- ✓ A file system poses **two different design problems.** The **first problem** is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file and the directory structure for organizing files. The **second problem** is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.
- ✓ File system itself is composed of many different levels. Layered design is shown in below **figure.**
- ✓ Each level in the design uses the features of lower levels to create new features for use by higher levels.
- ✓ The lowest level, **the I/O control**, consists of **device drivers** and **interrupts handlers** to transfer information between the main memory and the disk system.
- ✓ The **basic file system** issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- ✓ The **file-organization module** knows about files and their logical blocks, as well as physical blocks. It also includes free-space manager.
- ✓ **Logical file system** manages **metadata information.** Metadata includes all of the file-system structure except actual data.
- ✓ **File Control Block (FCB)** contains information about the file including the ownership permission and location of the file contents. Most operating systems supports more than one type of file system.



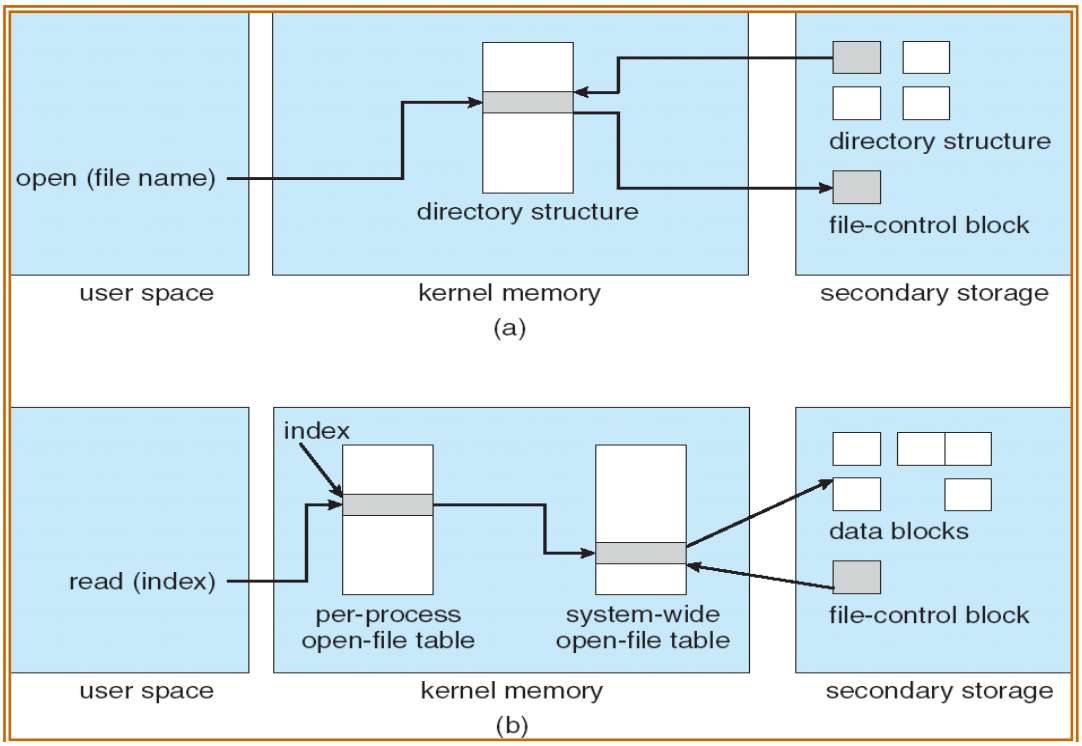
4.15 File System Implementation

- **Overview**

  - ✓ File system is implemented on the disk and the memory.
  - ✓ The implementation of the file system varies according to the OS and the file system, but there are some general principles.
  - ✓ UNIX supports UNIX File System (UFS), Windows supports File Allocation Table (FAT), FAT-32, Network Transfer File System (NTFS) and Linux supports more than 40 file systems.
  - ✓ If the file system is implemented on the disk it contains the following information:
    - **Boot Control Block** can contain information needed by the system to boot an OS from that partition. If the disk has no OS, this block is empty. It is the first block of the partition. In **UFS,** it is called **boot block** and in **NTFS** it is **partition boot sector**.
    - **Partition control Block** contains volume or partition details such as the number of blocks in partition, size of the blocks, and number of free blocks, free block pointer, free FCB count and FCB pointers. In **NTFS** it is stored in **master file tables**, In **UFS** this is called **super block**.
    - Directory structure is used to organize the files. In UFS, this includes file names and associated **inode** numbers. In NTFS, it is stored in the **master file table**.
    - An FCB contains many of the files details, including file permissions, ownership, size, location of the data blocks. In UFS this is called **inode**, In NTFS this information is actually stored within **master file table**.
  - ✓ The **In-memory information** is used for both file-system management and performance improvement via caching. The data are loaded at mount time and discarded at dismount. The structures includes,
    - An in-memory mount table containing information about each mounted information.
    - An in-memory directory structure that holds the directory information of recently accessed directories.
    - The **system wide open file table** contains a copy of the FCB of each open file as well as other information.
    - The **per-process open file table** contains a pointer to the appropriate entry in the system wide open file table as well as other information.
  - ✓ To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB. The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk. A typical **FCB** is shown in below **figure**



- ✓ File must be opened before using it for I/O. The open() call passes a file name to the file system. The open() system call searches the system-wide open-file table to find the file name given by the user. If it is opened, a per-process open-file table entry is created pointing to the existing system-wide open-file table.
- ✓ When a file is opened, the directory structure is searched for the given file name
- ✓ The open() call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are performed via this pointer.
- ✓ The name given to the entry varies. UNIX systems refer to it as a **file descriptor**, Windows refers to it as a **file handle.**
- ✓ When a process closes the file, the per-process table entry is removed, and system-wide entry's open count is decremented.
- ✓ The below **figure illustrates** the necessary file system structures provided by the operating systems.



- **Partition and Mounting**

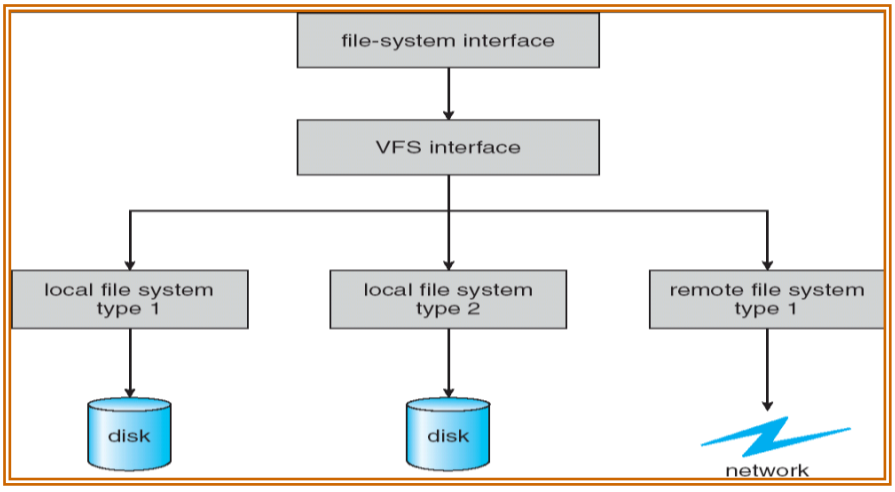
- ✓ A disk can be divided in to multiple partitions. Each partition can be either **raw** i.e., containing no file system or **cooked** i.e., containing a file system.
- ✓ Raw disk is used where no file system is appropriate. UNIX swap space can use a raw partition and do not use file system.
- ✓ Some databases uses raw disk and format the data to suit their needs. Raw disks can hold information need by disk RAID (Redundant Array of Independent Disks) system.
- ✓ Boot information can be stored in a separate partition. Boot information will have their own format. At the booting time, system does not load any device driver for the file system. Boot information is a sequential series of blocks, loaded as an image in to memory.
- ✓ **Dual booting** is also possible on some PCs where more than one OS are loaded on a system.
- ✓ A boot loader understands multiple file systems. Multiple OS can occupy the boot space once loaded and it can boot one of the OS available on the disk. The disks

can have multiple partitions each containing different types of file system and different types of OS.

- ✓ Boot partition contains the OS kernel and is mounted at a boot time.
- ✓ The operating system notes in its in-memory mount table that a file system is mounted, along with the type of the file system.Microsoft window based systems mount each partition in a separate **name space** denoted by a letter and a colon. On UNIX, file system can be mounted at any directory.

- **Virtual File Systems**

  - ✓ Modern operating systems must concurrently support multiple types of file systems.
  - ✓ Data structures and procedures are used to isolate the basic systemcall functionality from the implementation details. The file-system implementation consists of **three major layers** as shown in below **figure.**
  - ✓ The **first layer** is the file-system interface, based on the open(),read(), write(), and close() calls and on file descriptors.
  - ✓ The **second layer** is called the **Virtual file system** layer.
  - ✓ The virtual file system (VFS) layer, serves **two important functions**:
    - o It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems which are mounted locally.
    - o The VFS provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a **vnode,** that contains a numerical designator for a network-wide unique file.
  - ✓ The VFS activates file-system-specific operations to handle local requests according to their file-system types and even calls the NFS protocol procedures for remote requests. File handles are constructed from the relevant vnodes and are passed as arguments to these procedures. The layer implementing the file system type or the remote-file-system protocol is the **third layer** of the architecture.



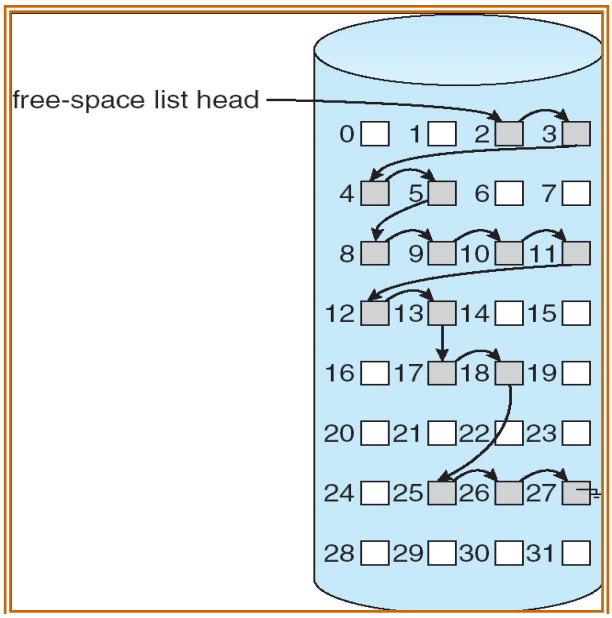
  - ✓ The **four main object** types defined by the Linux VFS are:
    - o The **inode object**, which represents an individual file.
    - o The **file object,** which represents an open file.
    - o The **superblock object**, which represents an entire file system.
    - o The **dentry object**, which represents an individual directory entry.

4.16 Directory Implementation

Directory is implemented in **two ways,**

- **Linear list**

  - ✓ Linear list is a simplest method.
  - ✓ It uses a linear list of file names with pointers to the data blocks. It uses a linear search to find a particular entry as shown in below**figure.**
  - ✓ It is **simple** for programming but **time consuming** to execute.
  - ✓ To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory. To delete a file, we search the directory for the named file and then release the space allocated to it.
  - ✓ To reuse the directory entry, we can do one of several things. We can **mark** the entry as unused or we can **attach** it to a list of free directory entries. A **third** alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory.
  - ✓ A linked list can also be used to decrease the time required to delete a file. Linear search is the main **disadvantage.**
  - ✓ An **advantage** of the sorted list is that a sorted directory listing can be produced without a separate sort step.



- **Hash table**

  - ✓ Hash table decreases the directory search time.
  - ✓ Insertion and deletion are fairly straight forward.
  - ✓ Hash table takes the value computed from that file name and it returns a pointer to the file name in the linear list.
  - ✓ Insertion and deletion are straightforward, but some provision must be made for **collision** i.e; situations in which two file names hash to the same location.
  - ✓ The **major difficulties** with a hash table are it is generally **fixed size** and the dependence of the hash function on that size.

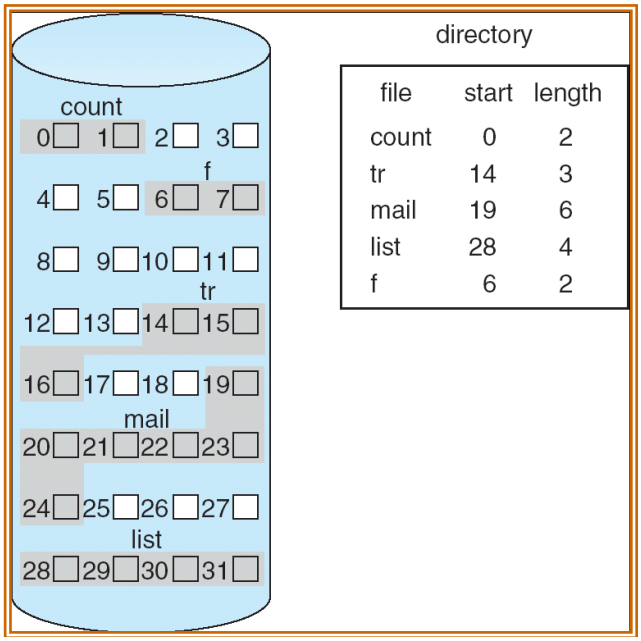
- ✓ A **chained-overflow hash table** can be used where each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

4.17 Allocation Methods(storage mechanisms available to store files)

Three major methods of allocating disk space are,

- **Contiguous Allocation**

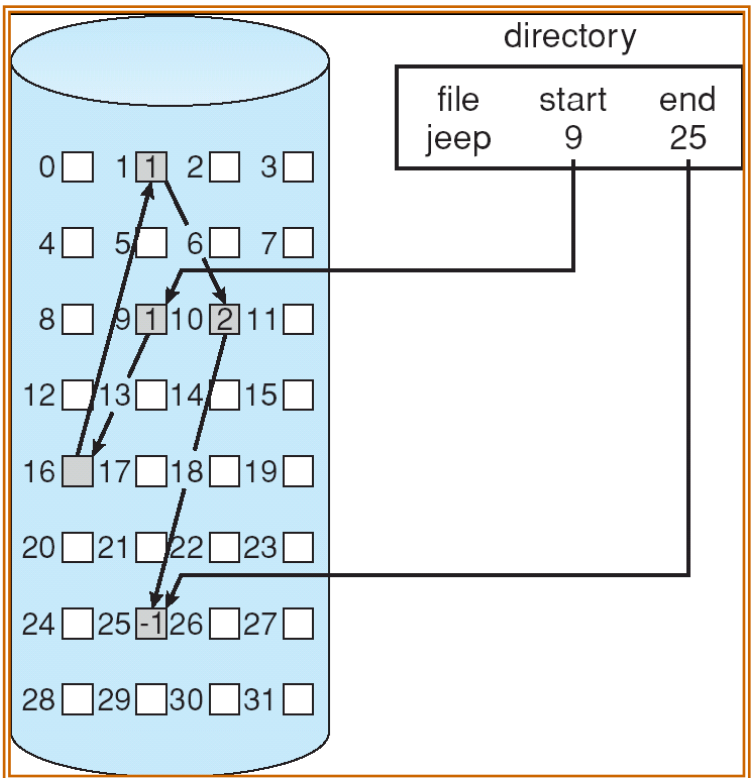
  - ✓ A **single set of blocks** is allocated to a file at the time of file creation. This is a **pre-allocation strategy** that uses portion of variable size.
  - ✓ The file allocation table needs single entry for each file, showing the **starting block and the length of the file.**
  - ✓ The below **figure** shows the contiguous allocation method.



  - ✓ Contiguous allocation algorithm suffers from **external fragmentation.**
  - ✓ **Sequential and direct access** can be supported by contiguous allocation.
  - ✓ Compaction is used to solve the problem of external fragmentation.
  - ✓ Another **problem** with contiguous allocation algorithm is **pre-allocation**, that is, it is necessary to declare the size of the file at the time of creation.
  - ✓ Even if the total amount of space needed for a file is known in advance, pre-allocation may be inefficient. A file that will grow slowly over a long period must be allocated enough space for its final size, even though much of that space will be unused for a long time. The file therefore has a large amount of **internal fragmentation.**
  - ✓ To **minimize these drawbacks**, some operating systems use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially; then, if that amount proves not to be large enough, another chunk of contiguous space, known as an **extent** is added. The location of a file's blocks is then recorded as a location and a block count, and a link to the first block of the next extent.

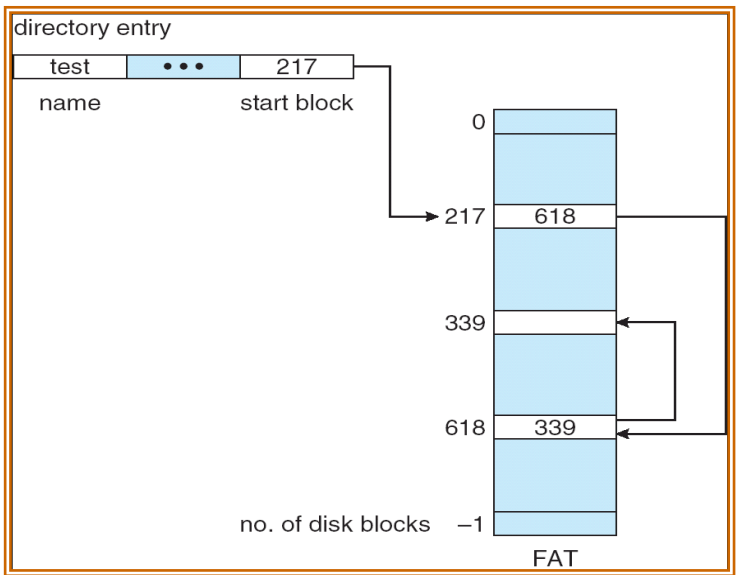
- **Linked Allocation**

  - ✓ It **solves the problem of contiguous allocation**. This allocation is on the basis of an individual block. Each block contains a **pointer** to the next block in the chain.
  - ✓ The disk block can be scattered anywhere on the disk.
  - ✓ The **directory** contains a pointer to the first and the last blocks of the file.
  - ✓ The below **figure** shows the linked allocation. To create a new file, simply create a new entry in the directory.



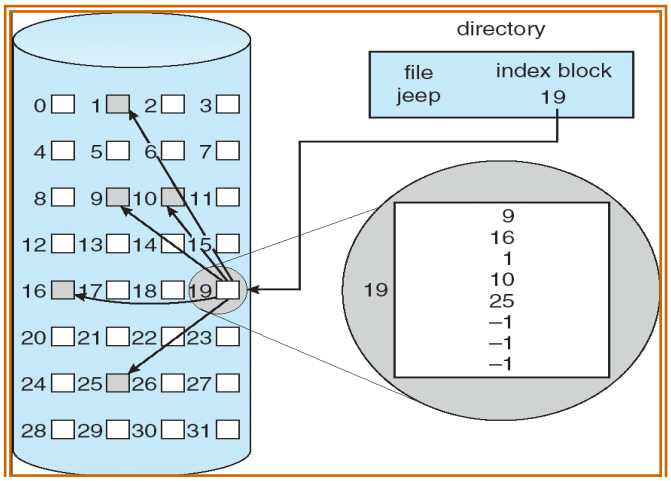
  - ✓ There is **no external fragmentation** since only one block is needed at a time.
  - ✓ The size of a file need not be declared when it is created. A file can continue to grow as long as free blocks are available.
  - ✓ The major **problem** is that it can be used effectively only for **sequential-access** files. Another disadvantage is the **space required for the pointers**. The usual **solution** to this problem is to collect blocks into multiples, called **clusters** and to allocate clusters rather than blocks.
  - ✓ Another problem of linked allocation is **reliability.** One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block.
  - ✓ An important **variation on linked allocation** is the use of a **File Allocation Table (FAT).** A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number. The FAT is used in the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block, which has a special end-of-file value as the table entry.

- ✓ An **unused block** is indicated by a table **value of 0**. An **illustrative example** is the FAT structure shown in below **figure** for a file consisting of disk blocks 217, 618, and 339.



- **Indexed Allocation**

  - ✓ The **file allocation table** contains a separate one level **index** for each file. The index has one entry for each portion allocated to the file.
  - ✓ The **ith** entry in the index block points to the **ith** block of the file. The below **figure** shows indexed allocation.



  - ✓ The indexes are not stored as a part of file allocation table but they are kept as a **separate block** and the entry in the file allocation table points to that block.
  - ✓ Allocation can be made on either fixed size blocks or variable size blocks. When the file is created all pointers in the index block are set to **nil.** When an entry is made a block is obtained from **free space manager**.
  - ✓ Indexed allocation supports both **direct access and sequential access** to the file. It supports direct access, without suffering from external fragmentation, because any

free block on the disk can satisfy a request for more space. Indexed allocation also suffer from wasted space.

- ✓ The **pointer overhead** of the index block is generally **greater than** the pointer overhead of linked allocation.
- ✓ Every file must have an index block, so we want the index block to be as small as possible. If the index block is **too small** it will not be able to hold enough pointers for a large file, and a **mechanism** will have to be available to deal with this issue.

- ✓ **Mechanisms** for this purpose include the following,
  - **Linked scheme**
    - ✓ To allow for large files, we can link together several index blocks. **For example,** an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address is is a pointer to another index block.

  - **Multilevel index**
    - ✓ A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size.

  - **Combined scheme**
    - ✓ Another alternative, used in the UFS, is to keep the First 15 pointers of the index block in the file's inode. The first 12 of these pointers point to direct blocks; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small block do not need a separate index block.
    - ✓ The **next three pointers** point to indirect blocks. The **first points to a single indirect block,** which is an index block containing not data but the addresses of blocks that do contain data. The **second points** to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The **last pointer** contains the address of a **triple indirect block**.
    - ✓ In this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the four-byte file pointers used by many operating systems.

- **Performance**

  - ✓ The allocation methods vary in their storage efficiency and data-block access times. Both are important criteria in selecting the proper methods for an operating system to implement.
  - ✓ For any type of access, **contiguous allocation** requires only one access to get a disk block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the disk address of the **i^{th} block** and read it directly.
  - ✓ For **linked allocation**, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access, but for direct access, an access to the i^{th} block might require i disk reads.

- ✓ The operating system must have appropriate data structures and algorithms to support both allocation methods.
- ✓ Files can be **converted** from one type to another by the creation of a new file of
- ✓ the desired type, into which the contents of the old file are copied. The old file may then be deleted and the new file renamed.
- ✓ **Indexed allocation** is more complex. If the index block is already in memory, then the access can be made directly. But keeping the index block in memory requires considerable space. If this memory space is not available, then we may have to read first the index block and then the desired data block. For a two-level index, two index-block reads might be necessary. For an extremely large file, accessing a block near the end of the file would require reading in all the index blocks before the needed data block finally could be read. Thus, **the performance of indexed allocation** depends on the **index structure, on the size of the file, and on the position of the block desired.**
- ✓ Some systems combine contiguous allocation with indexed allocation by using contiguous allocation for small files and automatically switching to an indexed allocation if the file grows large. Since most files are small, and contiguous allocation is efficient for small files, average performance can be quite good.

4.18 Free Space Management

- ✓ Since disk space is limited, we need to reuse the space from deleted files for new Files. To keep track of free disk space, the system maintains a **free-space list**.
- ✓ The free-space list records all free disk blocks-those not allocated to some file or directory. To **create a file**, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then **removed** from the free-space list. When a file is **deleted**, its disk space is added to the free-space list.
- ✓ There are **different methods** to manage free space.

- **Bit Vector**

  - ✓ The free-space list is implemented as **bit vector** or **bit map**.
  - ✓ Each block is represented by 1 bit. If the **block is free, the bit is 1; if the block is allocated, the bit is 0.**
  - ✓ **For example**, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be001111001111110001100000011100000 ...
  - ✓ The **main advantage** of this approach is its relative **simplicity** and its **efficiency** in finding the first free block or **nonconsecutive free blocks** on the disk.

- **Linked List**

  - ✓ Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.
  - ✓ This first block contains a pointer to the next free disk block, and so on.
  - ✓ This scheme is not efficient because to traverse the list, we must read each block, which requires substantial I/0 time.
  - ✓ The operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used.

- **Grouping**
  - ✓ A modification of the free-list approach stores the addresses of nfree blocks in the first free block. The **first n-1** of these blocks are actually free. The last block contains the addresses of other n free blocks, and so on.
  - ✓ The addresses of a large number of free blocks can be found quickly, unlike the situation when the linked-list approach is used.
- **Counting**
  - ✓ Another approach takes advantage of several contiguous blocks may be allocated or freed simultaneously.
  - ✓ Rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block.
  - ✓ Each entry in the free-space list then consists of a **disk address and a count.**

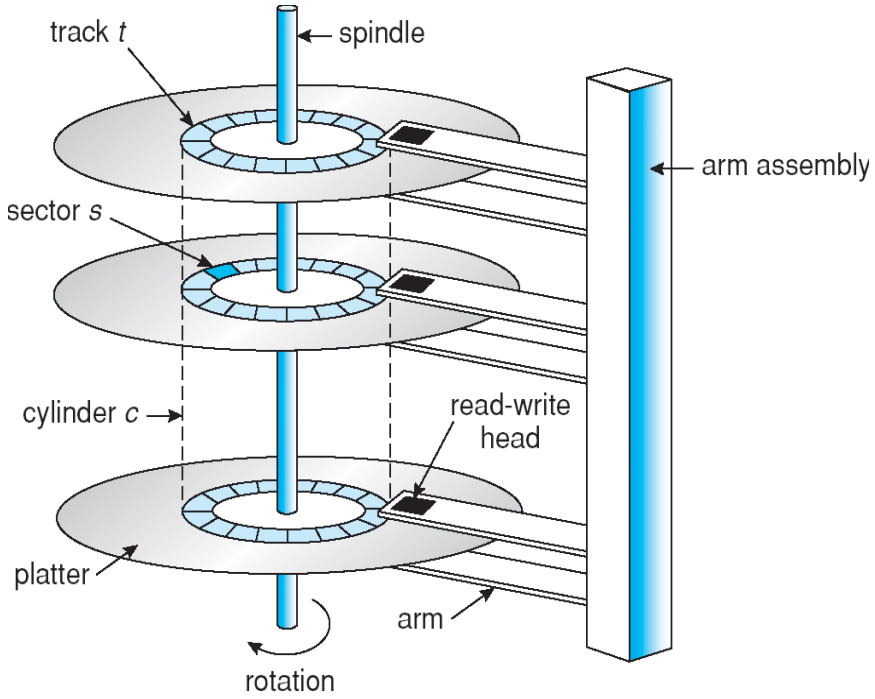
---o0o---

MODULE-5
Secondary Storage Structures, Protection

5.1 Overview of Mass Storage Structure

- **Magnetic Disks**

  - ✓ Magnetic disks provide the bulk of secondary storage for modern computer systems. Each disk platter has a flat circular shape like a CD as shown in below **figure.** The two surfaces of a platter are covered with a magnetic material.
  - ✓ We store information by recording it magnetically on the platters. A Read-write head flies just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit.
  - ✓ Surface of a platter is logically divided into circular **tracks** which are subdivided into **sectors.**
  - ✓ When the disk is in use, a drive motor spins it at high speed. Most drivers rotate 60 to 200 times per second.
  - ✓ Disk **speed has two parts -** The **Transfer rate** is the rate at which data flow between the drive and the computer. The **Positioning time** (**random-access time**) consists of the time necessary to move the disk arm to the desired cylinder, called the **seek time** and the time necessary for the desired sector to rotate and come under read-write head is called the **rotational latency.**



  - ✓ A disk can be **removable,** allowing different disks to be mounted as needed. **Ex:** floppy disks.
  - ✓ A disk drive is attached to a computer by a set of wires called an **I/O bus.** Several kinds of buses are available, including Enhanced Integrated Drive Electronics (**EIDE), Advanced Technology Attachment (ATA), Serial ATA (SATA), Universal serial Bus (USB), Fiber Channel (FC) and Small Computer System Interface (SCSI)** buses.

- ✓ The data transfers on a bus are carried out by special electronic processors called **controllers**. The **host controller** is the controller at the computer end of the bus. A **disk controller** is built into each disk drive.
- ✓ To perform a disk I/0 operations, the computer places a command into the host controller using **memory-mapped I/O ports**.

- **Magnetic Tapes**
  - ✓ Magnetic tape was used as an early secondary-storage medium. Its access time is slow when compared to main memory and magnetic disk.
  - ✓ Random access to magnetic tape is slower than disk so it is not very useful for secondary storage.
  - ✓ Tapes are used for backup and to store infrequently accessed data.

5.2 Disk Structure

- ✓ Disk drives are addressed as large one-dimensional arrays of **logical blocks** where logical block is the smallest unit of transfer.
- ✓ 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 outermost to innermost.
- ✓ By using this mapping, we can convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track.
- ✓ It is difficult to perform this translation, **for two reasons**. First, most disks have some **defective sectors**; **second, the number of sectors** per track is **not constant** on some drives.
- ✓ On media that use C**onstant Linear Velocity (CLV),** the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in **CD-ROM and DVD-ROM drives**.
- ✓ Alternatively, the disk rotation speed can stay constant, and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in **hard disks** and is known as C**onstant Angular Velocity (CAV).**

5.3 Disk Attachment

- ✓ Computers access disk storage in **two ways**
  - o Via I/O ports (host attached storage).
  - o via a remote host in a distributed file system (network attached storage)

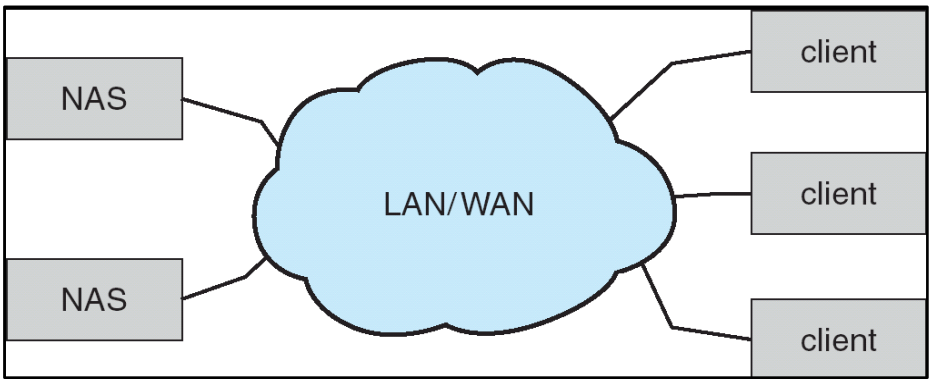
- **Host Attached Storage**

  - ✓ Host-attached storage is a storage accessed through local I/O ports.
  - ✓ High-end workstations and servers use more sophisticated I/O architectures like SCSI and Fiber channel (FC).
  - ✓ SCSI supports 16 devices on the bus.
  - ✓ Fiber-channel is a high-speed serial architecture that can operate over optical fiber and used in Storage Area Network (SAN).

- ✓ A wide variety of storage devices are suitable for use as host-attached storage.
- ✓ Ex: RAID arrays, CD, DVD and tape drives.

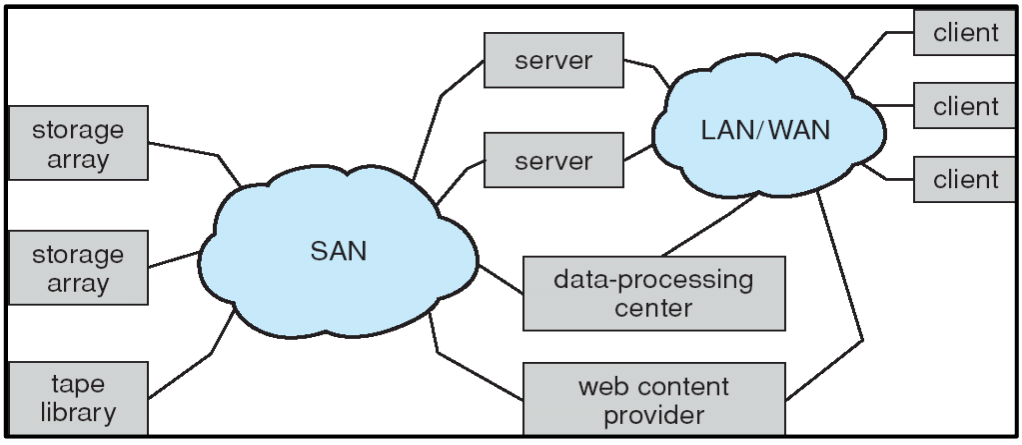
- **Network-Attached Storage**

  - ✓ A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a data network as shown in below **figure.**
  - ✓ Clients access network–attached storage via a remote-procedure-call interface such as **Network File System (NFS)** for UNIX and **Common Internet File System** (**CIFS**) for Windows.
  - ✓ The remote procedure calls are carried via TCP or UDP over all IP network.
  - ✓ **Internet Small Computer System Interface (ISCSI)** is the latest network-attached storage protocol. It uses IP network protocol to carry SCSI protocol.



- **Storage-Area Network**

  - ✓ A storage-area network (SAN) is a private network connecting servers and storage units as shown in below **figure.**
  - ✓ Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts.
  - ✓ A SAN switch allows or prohibits access between the hosts and the storage.
  - ✓ SAN's have more ports and less expensive ports than storage arrays.
  - ✓ Fiber channel is used to interconnect multiple storage area networks.

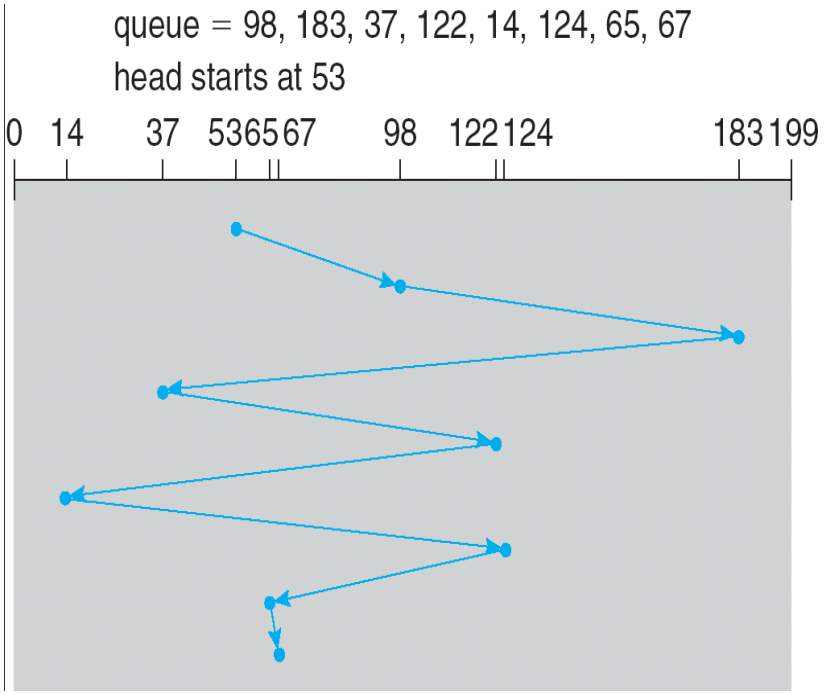


5.4 Disk Scheduling

- ✓ Disk **access time has two major components.**
  - **Seek Time:** It is the time taken by the disk arm to move the read-write head to the required cylinder that contains the desired sector.
  - **Rotational Latency:** It is the additional time for the disk to rotate the desired sector to the disk head.
  - **Disk Bandwidth:** It is the total number of bytes transferred, divided by the total time between the first request for service and the completion of last transfer.
- ✓ Whenever a process needs I/O from the disk, it issues a system call to the operating system.
- ✓ The request specifies several pieces of information,
  - Whether this operation is input or output?
  - What the disk address for the transfer is?
  - What the memory address for the transfer is?
  - What the number of sectors to be transferred is?
- ✓ If the desired disk drives available, the request can be serviced immediately. If the driver is busy, request will be placed in the queue. When one request is completed, OS chooses another pending request to service next. Several **disk scheduling** are used for this purpose.

- **FCFS Scheduling**

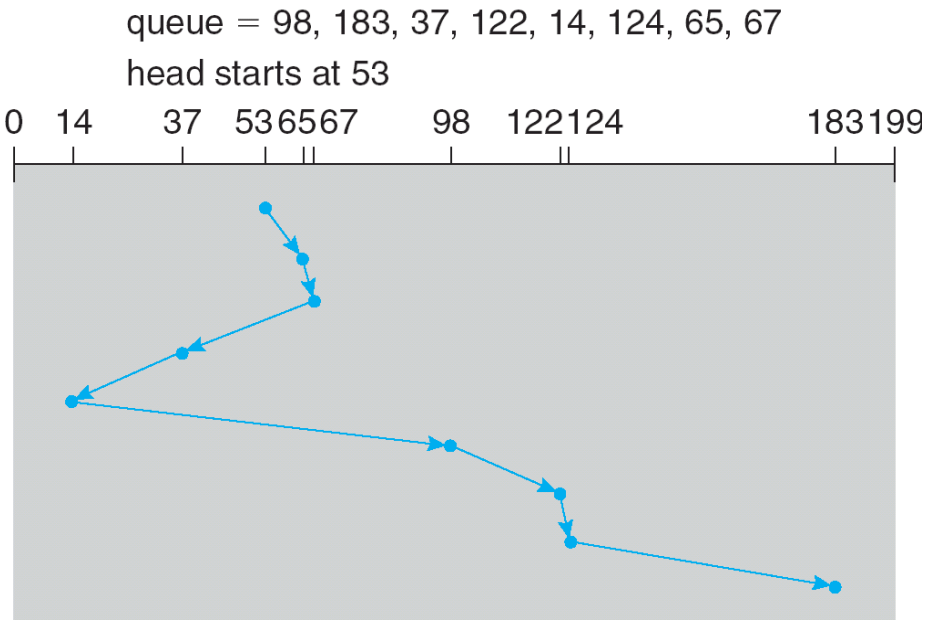
  - ✓ Simplest form of disk scheduling.
  - ✓ Generally doesn't provide fastest service.
  - ✓ **For example:** A disk queue with requests for I/O to blocks on cylinders 98, 183, 37,122,14,124,65,67 and disk head is initially at cylinder 53.
  - ✓ It will first move from 53 to 98, then to 183,37,122,14,124,65 and 67 as shown in below **figure** for a total head movement of **640 cylinders**.



|53-98| = 45
|98-183| = 85
|183-37| = 146
|37-122| = 85
|122-14| = 108
|14-124| = 110
|124-65| = 59
|65-67| = 02
640

- **SSTF Scheduling (Short-seek-Time-First)**

  - ✓ SSTF assumes that it is better to service all the requests close to the current head position before moving the head for away to service other requests.
  - ✓ SSTF choose the pending request closest to the current head position.
  - ✓ For the queue 98, 183, 37, 122, 14, 124, 65, 67 with head position = 53. Closest request to the initial head position is 65. Once we are at cylinder 65 next request served is 67, next is 37 and so on as shown in below **figure**. This scheduling results in a total head movement of only **236 cylinders**.
  - ✓ SSTF may cause starvation of some process.

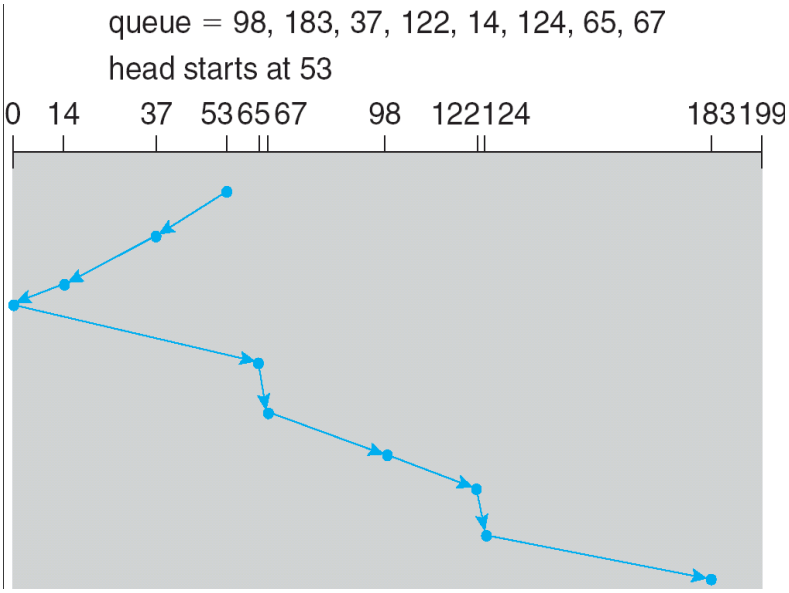


|53-65| = 45
|65-67| = 85
|67-37| = 146
|37-14| = 85
|14-98| = 108
|98-122| = 110
|122-124| = 59
|124-183| = 02

236

- **SCAN Scheduling (Elevator algorithm) (end points are not considered)**

  - ✓ In SCAN algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it get to other end of the disk. At the other end, the direction of head movement is reversed and servicing continues.
  - ✓ Head continuously scans back and forth across the disk.
  - ✓ Consider requests 98, 183, 37, 122, 14, 124, 65 and 67, head position = 53.
  - ✓ For this algorithm we need to know the direction of head movement
  - ✓ If the disk arm is moving towards 0, the head will service 37 and then 14.
  - ✓ At cylinder 0, the arm will reverse and move towards the other end servicing the requests at 65, 67, 98, 122, 124 and 183 as shown in below **figure.**



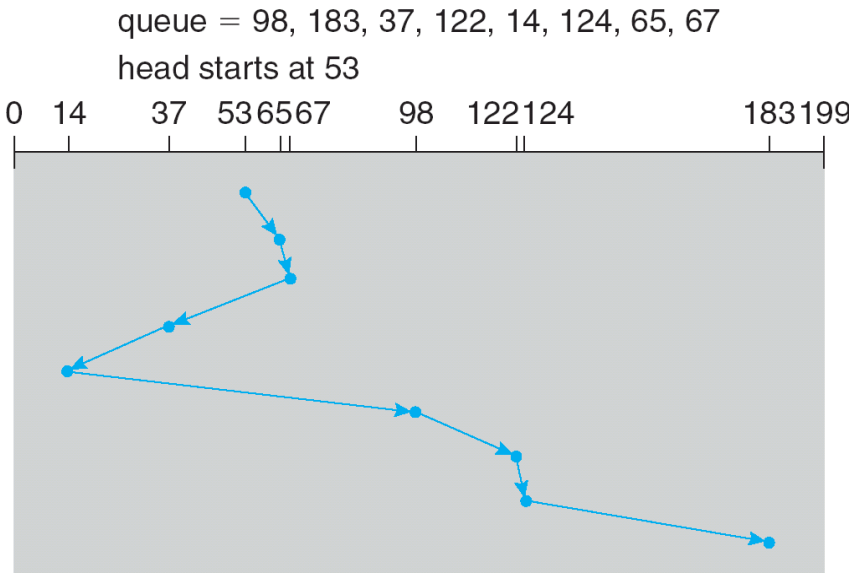
208

✓ This scheduling method results in a total head movement of only **208 cylinders.**

|53-37| = 16
|37-14| = 23
|14-65| = 51
|65-67| = 85
|67-98| = 31
|98-122| = 24
|122-124| = 02
|124-183| = 59

- **C-Scan Scheduling (end points are considered)**

  - ✓ Circular SCAN (C-SCAN) Scheduling is a variant of SCAN designed to provide a more uniform wait time.
  - ✓ C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, it immediately returns to the beginning of the disk, without servicing any requests on the return trip as shown in below **figure.**
  - ✓ The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.
  - ✓ This scheduling method results in a total head movement of only **382 cylinders.**

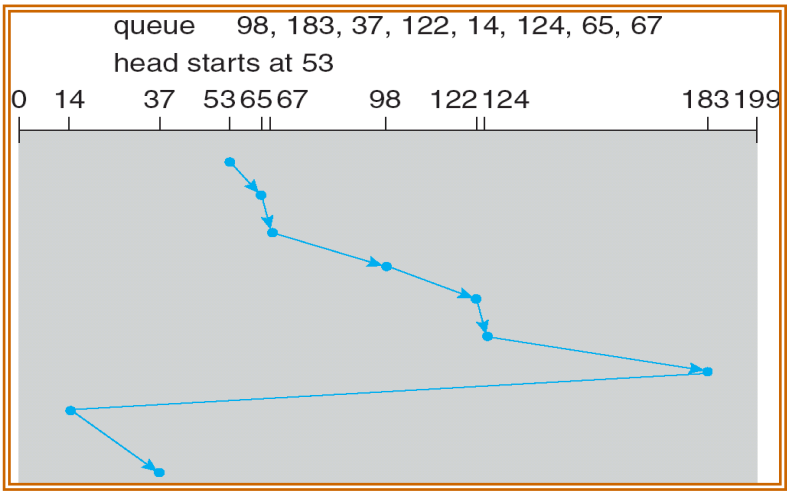


|53-65| = 12
|65-67| = 02
|67-98| = 31
|98-122| = 24
|122-124| = 02
|124-183| = 59
|183-199| = 16
|199-0| =

199
|0-14| = 14
|14-37| = 23
382

- **Look Scheduling**

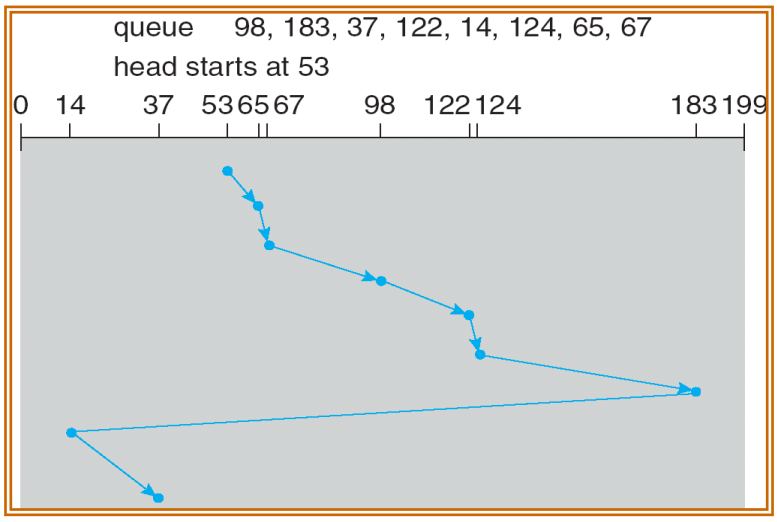
  - ✓ In this disk arm does not move across the full width of the disk.
  - ✓ The arm goes only as far as the final request in each direction. Then it reverses direction immediately, without going all the way to the end of the disk.
  - ✓ Version of SCAN and C-SCAN that follows this pattern are called LOOK and C-LOOK Scheduling because they look for a request before continuing to move in a given direction as shown in below **figure.**
  - ✓ This scheduling method results in a total head movement of only **299 cylinders.**



|53-65| = 12
|65-67| = 02
|67-98| = 31
|98-122| = 24
|122-124| = 02
|124-183| = 59
|183-37| = 146
|37-14| = 23
299

- **C-Look Scheduling (without servicing while returning)**

  - ✓ Version of C-SCAN
  - ✓ Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.
  - ✓ This scheduling method results in a total head movement of only 322 **cylinders.**



|53-65| = 12
|65-67| = 02
|67-98| = 31
|98-122| = 24
|122-124| = 02
|124-183| = 59
|183-14| = 169
|14-37| = 23
322

- **Selection of Disk Scheduling Algorithm**

  - ✓ SSTF is a common scheduling algorithm because it increases performance over FCFS.

- ✓ SCAN and C-SCAN used in the heavily loaded disk because these avoids starvation problem.
- ✓ File-allocation method must be considered while selecting scheduling algorithm.
- ✓ The location of directories and index blocks is also important.
- ✓ Caching the directories and index blocks in main memory can also help to reduce disk-arm movement particularly for read requests.
- ✓ Because of these complexities, the disk-scheduling algorithm should be written as a separate module of the operating system, so that it can be replaced with a different algorithm if necessary.

5.5 Disk Management

OS is responsible for several disk management activities like,

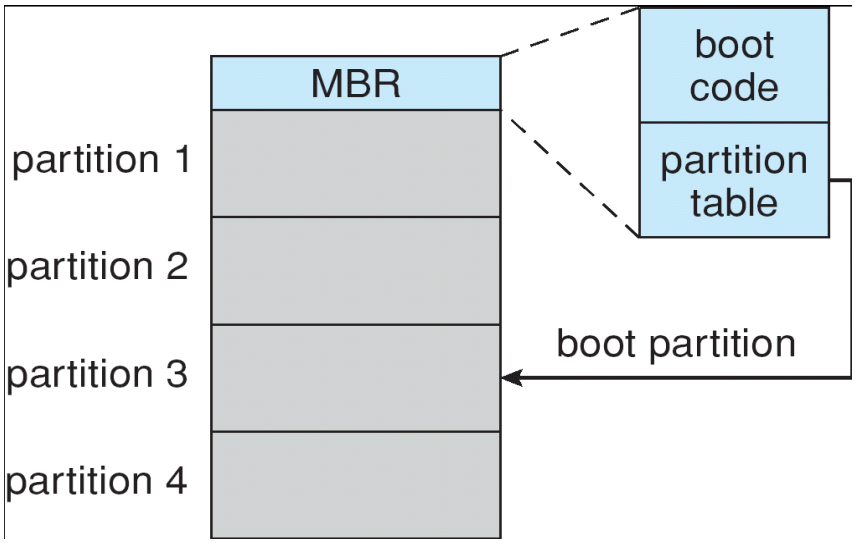
- **Disk Formatting**

  - ✓ A new magnetic disc is a blank slate, before a disk can store data, it must be divided into sectors that the disk controllers can read and write. This process is called **Low level formatting or Physical formatting.**
  - ✓ Low–level formatting fills the disk with a special data structure for each sector. It consists of a header, data area and a trailer. The header and trailer contain information used by the disk controller, such as a **sector number and an error correcting code (ECC)**.
  - ✓ To use a disk to hold files, the OS needs to record its own data structures on the disk. Two steps involved in this process are,
    - **Partition:** The disks are partitioned into one or more groups of cylinders.
    - **Logical formatting:** In this step, OS stores the initial file-system data structure onto the disk.
  - ✓ To **increase efficiency,** most file systems group the blocks together into larger chunks called as **clusters.**

- **Boot Block**

  - ✓ Initial **bootstrap** program is required for a computer to start running. It initializes the system and then starts the OS.
  - ✓ Bootstrap program finds the OS kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the OS execution.
  - ✓ It is stored in **Read Only Memory (ROM)** but the problem is that changing this bootstrap code requires changing the ROM hardware chips. So most systems store a **tiny bootstrap loader program** in the boot ROM whose job is to bring in a full bootstrap program from disk.
  - ✓ Full bootstrap program is stored in the **boot blocks** at a fixed location on the disk. A disk that has boot partition is called a **Boot disk** or **System disk.**
  - ✓ **For example**: The boot process in Windows 2000 system places its boot code in the **first sector** on the hard disk and it is termed as **master boot record (MBR).**
  - ✓ Windows 2000 allows a hard disk to be **divided into one or more partitions**; one partition is the **boot partition** and it contains the **operating system and device drivers.** The other partition contains a **table** listing the partitions for the hard disk and a **flag** indicating which partition the system is to be booted from, as shown in **figure.**

- ✓ Once the system identifies the boot partition, it reads the first sector from that partition which is called the boot sector and continues with the remainder of the boot process, which includes loading the various subsystems and system services.



- **Bad Blocks**

  - ✓ Because disks have moving parts and small tolerances, they are prone to failures. Failure may affect complete disk or it may affect one or two sectors. Most disks even come from factory with **bad blocks**.
  - ✓ On simple disks, such as some disks with IDE controllers, bad blocks are handled manually. For instance, the MS-DOS format command performs logical formatting and, as a part of the process, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block. If blocks go bad during normal operation, a special program such as **chkdsk** must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.
  - ✓ More sophisticated disks, such as the SCSI disks used in high-end PCs and most workstations and servers, are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level formatting at the factory and is updated over the life of the disk. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **Sector sparing** or **Forwarding.**
  - ✓ As an alternative to sector sparing, some controllers can be instructed to replace a bad block by sector slipping. For example,Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Then, sector slipping remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

5.6 Swap - Space Management

- ✓ **Swap - Space Management** is another low-level task of the operating system. Virtual memory uses disk space as an extension of main memory.
- ✓ Since disk access is much slower than memory access, using **swap space** significantly decreases system performance. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system.

- **Swap - Space Use**

  - ✓ Swap space is used in various ways by different operating systems, depending on the memory-management algorithms in use. For instance, systems that implement swapping may use swap space to hold an entire process image, including the code and data segments. Paging systems may simply store pages that have been pushed out of main memory.
  - ✓ The amount of swap space needed on a system can therefore vary depending on the amount of physical memory, the amount of virtual memory, and the way in which the virtual memory is used. It can range from a few megabytes of disk space to gigabytes.

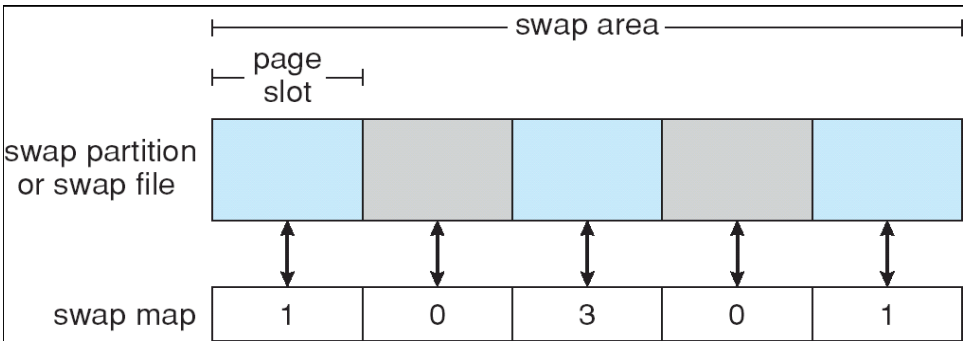
- **Swap - Space Location**

  - ✓ Swap Space can reside in one of **two places**.
    - o It can be simply a large file within the **file system** where normal file-system routines can be used to create it, name it and allocate its space. This approach is **easy to implement** but **inefficient**. Navigating the directory structure and the disk allocation data structures takes **time and extra disk accesses**.
    - o Swap space can be created in a separate **disk partition** (**raw partition).** A separate **swap space storage manager** is used to allocate and deallocate the blocks from the raw partition.
  - ✓ Some operating systems are flexible and can swap both in raw partitions and in file-system space.

- **Swap - Space Management: An Example**

  - ✓ The traditional UNIX kernel started with an implementation of swapping that copied entire processes between contiguous disk regions and memory. UNIX later evolved to a combination of swapping and paging as paging hardware became available.
  - ✓ In Solaris 1, when a process executes, text - segment pages containing code are brought in form the file system, accessed in main memory, and thrown away if select for page out. It is more efficient to reread a page from the file system than to write it to swap space and then reread it from there.
  - ✓ Swap Space is only used as a backing store for pages of **anonymous memory,** which includes memory allocated for the stack, heap and uninitialized data of a process.

- ✓ The biggest change is that Solaris now allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created. This scheme gives better performance.
- ✓ Linux allows one or more swap areas to be established. A swap area may be in either a swap file on a regular file system or a raw-swap-space partition. Each swap area consists of a series of 4-KB **page slots** which are used to hold swapped pages. An array of integer counters, each corresponding to a page slot called **swap map** is associated with a swap area. If the value of a counter is **0**, the corresponding page slot is available. Values **greater than 0** indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page.**For example,** a value of 3 indicates that the swapped page is mapped to three different processes. The **data structures for swapping** on Linux systems are shown in below **figure.**



5.7 System Protection

- ✓ The processes in an operating system must be protected from one another's activities. To provide such protection, we can use various mechanisms to ensure that only processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, and other resources of a system.
- ✓ **Protection** refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system.

5.8 Goals of Protection

- ✓ Prevention of **mischievous, intentional violation** of an access restriction by a user.
- ✓ Ensures that each program component in a system uses system resources according to stated policies.
- ✓ Protection can improve **reliability** by detecting latent errors at the interfaces between component subsystems.
- ✓ A protection - oriented system provides means to distinguish between authorized and unauthorized usage.
- ✓ Role of **protection** in a computer system is to provide a **mechanism** for the enforcement of the **policies** governing resource use.
- ✓ Mechanisms determine how something will be done and policies decide what will be done.

5.9 Principles of Protection

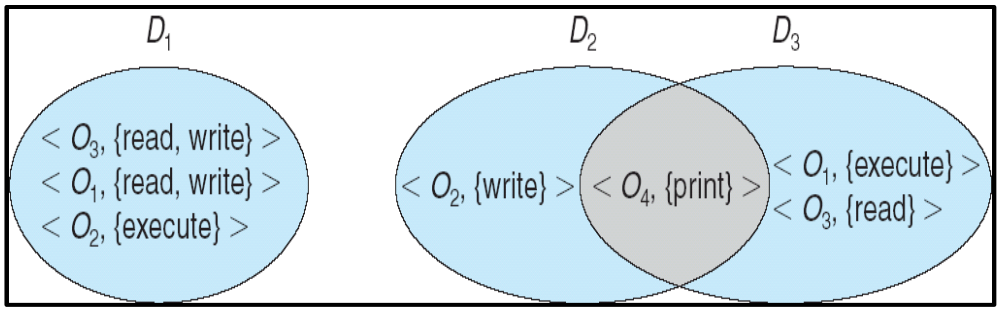
- ✓ Guiding principle for protection is the **Principle of Least Privilege**. It dictates that programs, users and even system be given just enough privileges to perform their tasks.
- ✓ Principle of least privilege implements programs, system calls in such a way that failure of a component does the minimum damage.
- ✓ It provides mechanisms to enable privileges when they are needed and to disable them when they are not needed.
- ✓ Managing users with the principle of least privilege entails creating a separate account for each user, with just the privileges that the user needs.An operator who needs to mount tapes and backup files on the system has access to just those commands and files needed to accomplish the job. Some systems implement **role-based access control (RBAC)** to provide this functionality.

5.10 Domain of Protection

- ✓ A computer system is a collection of processes and objects such as **hardware objects** like CPU, memory segments, printers, disks, and tape drives and **software objects** like files, programs, and semaphores.
- ✓ The operations that are possible may depend on the object. A process should be allowed to access only those resources for which it has authorization.
- ✓ At anytime, a process should be able to access only those resources that it currently requires to complete its task. This is referred as **Need-to-Know** principle. It limits the amount of damage caused by faulty process.

- **Domain Structure**

  - ✓ A process operates within a **Protection Domain** which specifies the resources that the process may access. Each domain defines a set of objects and the types of operations that may be invoked on each object.
  - ✓ Ability to execute an operation on an object is called **Access Right.**
  - ✓ A **domain** is a collection of access rights. It is denoted by **ordered pair - <object-name, right-set>**. **For example,** if domain D has the access right <file F, {read, write}>, then a process executing in domain D can both read and write file F andit cannot perform any other operation on that object.
  - ✓ Domains do not need to be disjoint; they may **share access rights**. **For example**, in **below figure** we have **three domains: D_1, D_2, and D_3**. The access right < 0_4, {print}> is shared by D_2 and D_3, implying that a process executing in either of these two domains can print object 0_4.



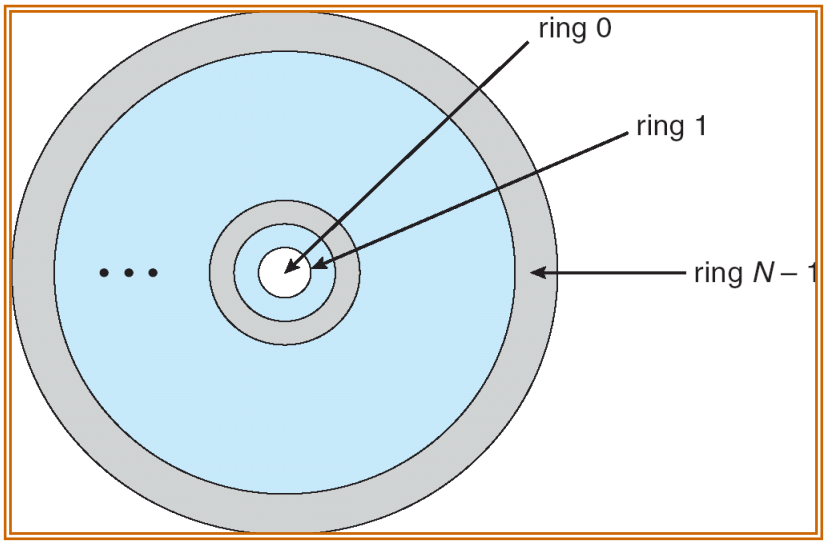
- ✓ **Association** between a process and a domain may be **static or dynamic**. Dynamic association supports **domain switching** i.e., it enables the process to switch from one domain to another. A domain can be realized in a **variety of ways:**
    - **Each user may be a domain:** In this case the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when one user logs out and another user logs in.
    - **Each process may be a domain:**In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.
    - **Each procedure may be a domain:**In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

- **An example - UNIX**

    - ✓ In UNIX Operating system, domain is related with the user. S**witching the domain** corresponds to changing the user identification temporarily.
    - ✓ Owner **identification and a domain bit (known as the setuid bit)** are associated with each file. When the **setuid bit is on**, and a user executes that file, the user ID is set to that of the owner of the file, but when the bit is **off,** the user ID does not change.
    - ✓ **Other methods** are used to change domains in operating systems in which user IDs are used for domain definition, because almost all systems need to provide such a mechanism.
    - ✓ An alternative to this method used in other operating systems is to place **privileged programs** in a special directory. The operating system would be designed to **change the user lD**of any program run from this directory, either to the equivalent of root or to the user lD of the owner of the directory.
    - ✓ Even more restrictive, and thus more protective, are systems that simply do not allow a change of user ID. In these instances, special techniques must be used to allow users access to privileged facilities. For instance, a **daemon process** may be started at boot time and run as a special user ID.
    - ✓ In any of these systems, great care must be taken in writing privileged programs.

- **An example –MULTICS**

    - ✓ Protections of domains are organized hierarchically into a **ring structure**. Rings are numbered from 0 to ring N-1. Each ring is a single domain as shown in **figure.**



- ✓ Consider any two domain rings, i.e, D_i&D_j. If value of j is less than i (j<i), then domain D_i is subset of domain D_j. The process executing in domain D_j has more privileges than the process executing in domain D_i. **Ring 0 has full privileges.**
- ✓ MULTICS has a **segmented address space**; each segment is a file, and each segment is associated with one of the rings. A segment description includes an entry that identifies the ring number and **three access bits** to control reading, writing, and execution.
- ✓ A current-ring-number counter is associated with each process, identifying the ring in which the process is executing currently. When a process is executing in ring i, it cannot access a segment associated with ring j (j <i). It can access a segment associated with ring k (k>= i). The type of access is restricted according to the access bits associated with that segment.
- ✓ **Domain switching** in MULTICS occurs when a process crosses from one ring to another by calling a procedure in a different ring. This switch must be done in a controlled manner; otherwise, a process could start executing in ring 0, and no protection would be provided.
- ✓ To allow **controlled domain switching,** we modify the ring field of the segment descriptor to include the following:
  - o **Access bracket**. A pair of integers, bland b2, such that bl<= b2.
  - o **Limit.** An integer b3 such that b3 >b2.
  - o **List of gates.** Identifies the **entry points** (or **gates)** at which the segments may be called.
- ✓ If a process executing in ring i calls a procedure (or segment) with access bracket (bl,b2), then the call is allowed if **bl<= i <= b2,** and the current ring number of the process remains i. Otherwise, a **trap** to the operating system occurs, and the situation is **handled as follows**:
  - o If i <bl, then the call is allowed to occur, because we have a transfer to a ring (or domain) with fewer privileges. If parameters are passed that refer to segments in a lower ring then these segments must be copied into an area that can be accessed by the called procedure.
  - o If i >b2, then the call is allowed to occur only if b3 is greater than or equal to i and the call has been directed to one of the designated entry points in the list of gates. This scheme allows processes with limited access rights to call procedures in lower rings that have more access rights, but only in a carefully controlled manner.

- ✓ The main **disadvantage** of the ring structure is that it does not allow us to enforce the **need-to-know principle.** The MULTICS protection system is generally more **complex and less efficient**.

5.11 Access Matrix

- ✓ The model of protection can be viewed abstractly as a matrix, called an **access matrix.**
- ✓ The **rows** of the access matrix represent **domains**, and the **columns** represent **objects**. Each entry in the matrix consists of a set of access rights.
- ✓ The entry **access(i,j)** defines the set of operations that a process executing in domain D_i can invoke on object O_j.
- ✓ **For Example,** consider the access matrix shown in below **figure**
- ✓ The access matrix consists of four domains, four objects, three files and one printer. The **summary of access matrix** is as follows:
  - Process in domain D_1 can read file F_1 and file F_3.
  - Process in domain D_2 can only use printer.
  - Process in domain D_3 can read file F_2 and execute file F_3.
  - Process in domain D_4 can read and write file F_1 and file F_3.

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- ✓ Access matrix scheme provides us with the mechanism for specifying a variety of policies. We must ensure that a process executing in domain D_i, can access only those objects specified in row, and then only as allowed bythe access-matrix entries. When a user creates a new object O_j, the column O_j, is added to the access matrix. Blank entries indicate no access rights. A process is switched from one domain to another domain by executing **switch operation** on the object.
- ✓ Each entry in the access matrix may be modified individually. Domain switch is only possible if and only if the access right **switch € access (i, j)**. The below **figure (1)** shows the access matrix with domains as objects. Process can change domain as follows,
  - Process in domain D_2 can switch to domain D_3 and domain D_4.
  - Process in domain D_4 can switch to domain D_1.
  - Process in domain D_1 can switch to domain D_2.

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

figure (1)

- ✓ Access matrix is inefficient for storage of access rights in computer system because they tend to be large and sparse.
- ✓ Allowing controlled change in the contents of the access-matrix entries requires **three additional operations: copy, owner, and control.**
- ✓ The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an **asterisk (*)** appended to the access right.
- ✓ The **copy** right allows the access right to be copied only within the column for which the right is defined.
- ✓ **For example,** as shown in below **figure(a)**, a process executing in domain D_2 can copy the read operation into any entry associated with file F_2. Hence, the access matrix of **figure(a)** can be modified to the access matrix shown in **figure (b).**

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

- ✓ This scheme has **two variants:**

- A right is copied from access(i, j) to access(k, j); it is then removed from access(i, j). This action is a transfer of a right, rather than a copy.
- Propagation of the copy right may be limited. That is, when the right R* is copied from **access(i,j) to access(k,j), only the right R (not R*)** is created. A process executing in domain $D_k$cannot further copy the right R.

✓ A system may select only one of these three copy rights, or it may provide all three by identifying them as separate rights: copy, transfer, and limited copy.

✓ The **owner right controls these operations.** If access(i, j) includes the owner right, then a process executing in domain $D_i$can add and remove any right in any entry in column j.

✓ **For example,** as shown in below **figure (a),** domain D_1 is the **owner of F_1** and thus can add and delete any valid right in column F_1. Similarly, domain D_2 is the owner of F_2 and F_3 and thus can add and remove any valid right within these two columns. Thus, the access matrix of **figure (a)** can be modified to the access matrix as shown in **figure (b).**

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

✓ The copy and owner rights allow a process to **change the entries** in a column.

✓ A mechanism is also needed to change the entries in a row. The **control** right is applicable only to domain objects. If access(i, j) includes the **control**right, then a process executing in domain D_i can remove any access right from row j.

✓ **For example,** in **figure (1)** we include the control right in access (D_2, D_4). Then, a process executing in domain D_2 could modify domain D_4, as shown in below **figure.**

✓ The **problem** of guaranteeing that no information initially held in an object can migrate outside of its execution environment is called the **confinement problem**. This problem is in **general unsolvable.**

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

5.12 Implementing Access matrix

- ✓ It is implemented in several ways. Methods for implementing access matrix are,
  - Global table.
  - Access lists for objects.
  - Capability list for domain.
  - A lock key mechanism.

- **Global Table**

  - ✓ It is the simplest method for implementation of access matrix. Global table consists of domain, object and right set. The order of syntax is **<domain, object, right-set >**
  - ✓ If **operation M** is executed on an object O_j within domain D_i, the global table is searched for a triple- $< D_i, O_j, R_k >$ with M € R_k. If the above triple is found, then operation is allowed to continue. If suppose triple is not found then an exception error condition occurs.
  - ✓ **Limitations of Global table are,** Global table is large and it cannot be kept in memory and additional Input/ Output is required.

- **Access list for objects**

  - ✓ Each column in the access matrix can be implemented as an access list for one object. The empty entries can be discarded.
  - ✓ The resulting list for each object consists of **ordered pairs <domain, rights-set>,** which define all domains with a **nonempty** set of access rights for that object.
  - ✓ This approach can be extended easily to define a list plus a default set of access rights. When an operation M on an object O_i is attempted in domain D_i, we search the access list for object O_i, looking for an entry $< D_i, R_k >$ with M € R_k. If the entry is found, we allow the operation; if it is not, we check the default set. If M is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs.

- **Capability list for domains**

  - ✓ Each row is associated with its domain.
  - ✓ A **capability list** for a domain is a list of objects together with the operations allowed on those objects.

- ✓ An object is often represented by its **physical name or address, called a Capability.**
- ✓ Process executes operation M by specifying the capability (or pointer) for object O_j as a parameter.
- ✓ Capabilities are **distinguished** from other data in **two ways**-
  - Each object has a **tag** to denote its type as either a capability or as accessible data.
  - The **address space** associated with a program can be split into two parts. One part is accessible to the program and contains the programs normal data and instructions. The other part containing the capability list is accessible only by the operating system.

- **A Lock –Key Mechanism**

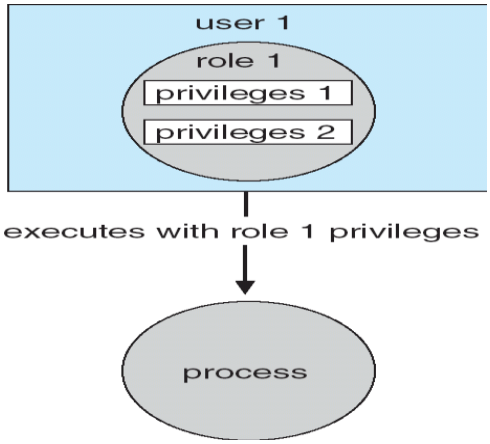
  - ✓ The lock key scheme is a compromise between access list and capability list.
  - ✓ Each object has a list of **unique bit** patterns called **locks** and each domain has a list of unique bit patterns called **keys**.
  - ✓ A process executing in a domain can access an object only if the domain has a key that matches one of the locks of the object.
  - ✓ Users are not allowed to examine or to modify the list of keys directly.

    - **Comparison of methods**

      - ✓ Global table is simple but table can be quite large and cannot take advantage of special groupings of objects or domains.
      - ✓ Access lists corresponds directly to the needs of users. But determining the set of access rights of a particular domain is difficult.
      - ✓ Capability lists do not correspond directly to the needs of users. They are useful for localizing information for a given process.
      - ✓ Lock-Key mechanism is a compromise between access lists and capability lists. The mechanism can be effective and flexible depending on the length of the keys.

5.13 Access Control

- ✓ **Role-based Access control (RBAC)** facility revolves around privileges.
- ✓ A privilege is the right to execute a system call or to use an option within that system call. Privileges can be assigned to **process or roles**.
- ✓ Users are assigned roles or can take roles based on passwords to the roles.
- ✓ In this way a user can take a role that enables a privilege, allowing the user to run a program to accomplish a specific task as shown in the below **figure**.



5.14 Revocation of Access Rights

- ✓ Revocation of access rights to objects in shared environment is possible. Various questions about revocation may arise as follows,
  - **Immediate versus delayed.** Does revocation occur immediately, or is it delayed? If revocation is delayed, can we find out when it will take place?
  - **Selective versus general.** When an access right to an object is revoked, does it affect all the users who have an access right to that object, or can we specify a **select group of users whose access rights should be revoked?**
  - **Partial versus total.** Can a subset of the rights associated with an object be revoked, or must we revoke all access rights for this object?
  - **Temporary versus permanent.** Can access be revoked permanently (that is, the revoked access right will never again be available), or can access be revoked and later be obtained again?
- ✓ Revocation is **easy for access list and complex for capabilities list.**The access list is searched for any access rights to be revoked, and they are deleted from the list.
- ✓ **Schemes** that implement revocation for capabilities include the following:
  - **Reacquisition.** Periodically, capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.
  - **Back-pointers.** A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, change the capabilities as necessary. This scheme was adopted in the MULTICS system. It is quite general, but its implementation is costly.
  - **Indirection.** The capabilities point indirectly to the objects. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it. Then, when an access is attempted, the capability is found to point to an illegal table entry. Table entries can be reused for other capabilities without difficulty, since both the capability and the table entry contain the unique name of the object. The object for a capability and its table entry must match. This scheme was adopted in the CAL system. It does not allow selective revocation.
  - **Keys.** A key is a unique bit pattern that can be associated with a capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process that owns the capability. A **master key** is associated with each object; it can be defined or replaced with the **set-key** operation. When a

capability is created, the current value of the master key is associated with the capability. When the capability is exercised, its key is compared with the master key. If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised. If we associate a list of keys with each object, then selective revocation can be implemented. Finally, we can group all keys into one **global table** of keys. A capability is valid only if its key matches some key in the global table. In key-based schemes, the operations of defining keys, inserting them into lists, and deleting them from lists should not be available to all users.

5.15 Capability Based Systems

Capability based protection systems are of two types,

- **An example - Hydra**
  - ✓ Hydra provides a fixed set of possible access rights that are known to and interpreted by the system. These rights include such basic forms of access as the right to read, write or execute a memory segment.
  - ✓ Operations an object are defined procedurally. The procedures that implement such operations are themselves a form of object and they are accessed indirectly by capabilities. When the definition of an object is made known to hydra, the names of operations on the type become **auxiliary right.**
  - ✓ Hydra also provides **rights amplification.** This scheme allows certification of a procedure as trust worthy to act on an formal parameter of a specified type, on behalf of any process that holds a right to execute the procedure.
- **An example - Cambridge Cap System**
  - ✓ CAP system is simpler and superficially less powerful than that of hydra. CAP has two kinds of capabilities-
    - o **Data capability** can be used to provide access to objects, but the only rights provided are the standard read, write and execute of the individual storage segments associated with the object. Data capabilities are interpreted by microcode in the CAP machine.
    - o **Software capability** is protected, but not interpreted by the CAP microcode. It is interpreted by a protected procedure, which may be written by an application programmer as part of a subsystem. The interpretation of a software capability is left completely to the subsystem, through the protected procedures it contains. This scheme allows a variety of protection policies to

The Linux System

5.16 Linux History

- ✓ The Linux system has grown to include much UNIX functionality. Linux development revolved largely around the central operating-system kernel.
- ✓ The linux kernel is an entirely original piece of software developed from scratch by the Linux community. The **linux system**, as we know it today, includes *a* multitude of components, some written from scratch, others borrowed from other development projects, and still others created in collaboration with other teams.
- ✓ A **linux distribution** includes all the standard components of the Linux system, plus *a* set of administrative tools to simplify the initial installation and subsequent upgrading of Linux and to manage installation and removal of other packages on the system. A modern distribution also typically includes tools for management of file systems, creation and management of user accounts, administration of networks, Web browsers, word processors, and so on.

- **The Linux Kernel**

  - ✓ The first Linux kernel released to the public was Version 0.01, dated May 14, 1991. It had no networking, ran only on 80386-compatible Intel processors and PC hardware, and had extremely limited device-driver support.
  - ✓ The virtual memory subsystem was also fairly basic and included no support for memory-mapped files; however, even this early incarnation supported shared pages with copy-on-write.
  - ✓ The only file system supported was the Minix file system.
  - ✓ The next milestone version, Linux 1.0, was released on March 14, 1994.
  - ✓ The single biggest new feature was networking: 1.0 included support for UNIX's standard TCP */IP* networking protocols, as well as a BSD-compatible socket interface for networking programming.
  - ✓ Device-driver support was added for running IP over an Ethernet or (using PPP or SLIP protocols) over serial lines or modems.
  - ✓ The 1.0 kernel also included a new, much enhanced file system.
  - ✓ The developers extended the virtual memory subsystem to support paging to swap files and memory mapping of arbitrary files.
  - ✓ A range of extra hardware support was also included in this release.
  - ✓ System V UNIX-style **interprocess communication(IPC)** including shared memory, semaphores, and message queues, was implemented.
  - ✓ Kernels with an odd minor-version number, such as 1.1, 1.3, and2.1, are **development kernels**; even numbered minor-version numbers are stable **production kernels**. In March 1995, the 1.2 kernel was released.
  - ✓ support a much wider variety of hardware, including the new PCI hardware bus architecture.
  - ✓ They also updated the networking stack to provide support for the IPX protocol and made the IP implementation more complete by including accounting and firewalling functionality.
  - ✓ The 1.2 kernel was the final PC-only Linux kernel.
  - ✓ The source distribution for Linux 1.2 included partially implemented support for SPARC, Alpha, and MIPS CPUs.

- ✓ This work was finally released as Linux 2.0 in Jmce 1996. This release was given a major version-number increment on account of two major new capabilities.
- ✓ support for multiple architectures, including a 64-bit native Alpha port, and support for multiprocessor architectures.
- ✓ The memory-management code was substantially improved to provide a unified cache for file-system data independent of the caching of block devices.
- ✓ file-system and virtual memory performance. For the first time, file-system caching was extended to networked file systems, and writable memory-mapped regions also were supported.
- ✓ The 2.0 kernel also included much improved TCP /IP performance, and a number of new networking protocols were added, including Apple Talk, AX.25 an'lateur radio networking, and ISDN support.
- ✓ The ability to mount remote netware and SMB (Microsoft LanManager) network volumes was added.
- ✓ Other major improvements in 2.0 were support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand.
- ✓ Improvements continued with the release of Linux 2.2 in January 1999.
- ✓ A port for Ultra SPARC systems was added.
- ✓ Networking was enhanced with more flexible firewalling, better routing and traffic management, and support for TCP large window and selective acks.
- ✓ Acorn, Apple, and NT disks could now be read, and NFS was enhanced and a kernel-mode NFS daemon added.
- ✓ Signal handling, interrupts, and some I/0 were locked at a finer level than before to improve symmetric multiprocessor (SMP) performance.
- ✓ Advances in the 2.4 and 2.6 releases of the kernel include increased support for SMP systems, journaling file systems, and enhancements to the memory-management system.
- ✓ The process scheduler was modified in Version 2.6, providing an efficient 0(1) scheduling algorithm.
- ✓ In addition, the Linux 2.6 kernel is now preemptive, allowing a process to be preempted while running in kernel mode.

- **The Linux System**

- ✓ The Linux kernel forms the core of the Linux project, but other components make up the complete Linux operating system.
- ✓ Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project.
- ✓ The **GNU C compiler(gcc)**, were alreadyof sufficiently high quality to be used directly in Linux.
- ✓ The networking administration tools under Linux were derived from code first developed for 4.3 BSD, but more recent BSD derivatives, such as FreeBSD, have borrowed code from Linux in return.
- ✓ Examples include the Intel floating-point-emulation math library and the PC sound-hardware device drivers.
- ✓ The Linux system as a whole is maintained by a loose network ofdevelopers collaborating over the Internet.

- ✓ The **file system hierarchy standard** document is also maintained by the Linux community as a means of ensuring compatibility across the various system components.

- **Linux Distributions**

  - ✓ Distributions, include much more than just the basic Linux system.
  - ✓ They typically include extra system-installation and management utilities, as well as precompiled and ready-to-install packages of many of the common UNIX tools, such as news servers, Web browsers, text-processing and editing tools, and even games.
  - ✓ Linux distributions include a package-tracking database that allows packages to be installed, upgraded, or removed painlessly.
  - ✓ The SLS distribution, was the first collection of Linux packages that was recognizable as a complete distribution.
  - ✓ Although it could be installed as a single entity, SLS lacked the package-management tools.
  - ✓ The **slackware** distribution represented a great improvement in overall quality, even though it also had poor package management; in fact, it is still one of the most widely installed distributions in the Linux community.
  - ✓ **Red Hat** and**Debian** are particularly popular distributions; the first comes from a commercial Linux support company and the second from the free-software Linux community.
  - ✓ Other commercially supported versions of Linux include distributions from **Caldera, craftworks** and **Workgroup solutions**.

- **Linux Licensing**

  - ✓ The Linux kernel is distributed under the GNU general public license (GPL), the terms of which are set out the Free Software Foundation.
  - ✓ Linux is not public-domain software .**Public domine** implies that the authors have waived copyright rights in the software, but copyright rights in Linux code are still held by the code's various authors.
  - ✓ Linux is *free* software, however, in the sense that people can copy it, modify it, use it in any manner they want, and give away their own copies, without any restrictions.
  - ✓ The main implications of Linux's licensing terms are that nobody using linux, or creating a derivative of Linux (a legitimate exercise), can make the derived product proprietary.
  - ✓ Software released under the GPL cannot be redistributed as a binary-only product.

5.17 Design Principles

- ✓ Linux resembles any other traditional, no nmicro kernel UNIX implementation. It is a multiuser, multitasking system with a full set of UNIX-compatible tools.
- ✓ Linux's file system adheres to traditional UNIX semantics, and the standard UNIX networking model is implemented fully.

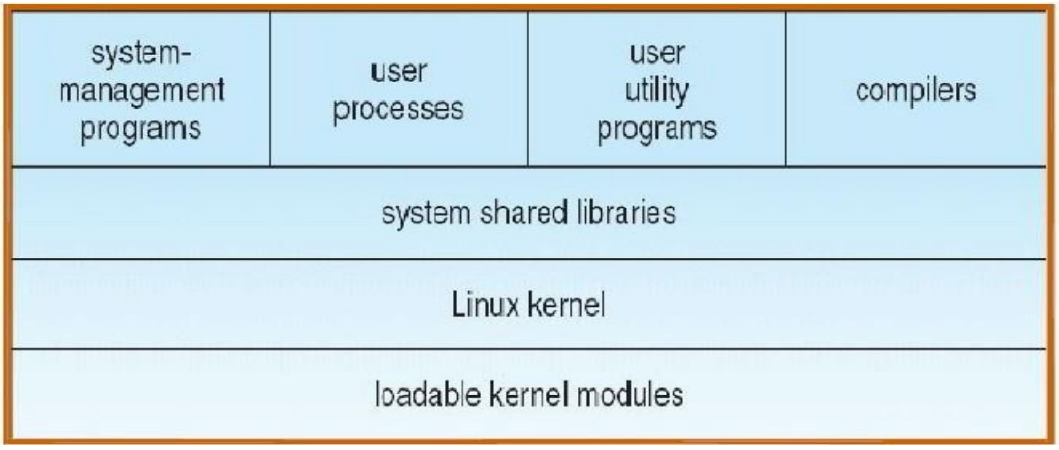
- ✓ Linux can run happily on a multiprocessor machine with hundreds of megabytes of main memory and many gigabytes of disk space, but it is still capable of operating usefully in under 4 MB of RAM.
- ✓ Speed and efficiency are still important design goals, but much recent and current work on Linux has concentrated on a third major design goal: standardization.
- ✓ There are POSIX documents for common operating-system functionality and for extensions such as process threads and real-time operations.
- ✓ the Linux programming interface adheres to SVR4 UNIX semantics.
- ✓ A separate set of libraries is available to implement BSD semantics in places where the two behaviors differ significantly.
- ✓ Linux currently supports the POSIX threading extensions-Pthreads -and a subset of the POSIX extensions for real-time process control.

- **Components of a Linux System**

  - ✓ The Linux system is composed of three main bodies of code, in line with mosttraditional UNIX implementations:
    - **Kernel-** The kernel is responsible for maintaining all the important abstractions of the operating system, including such things as virtual memory and processes.
    - **System libraries**- The system libraries define a standard set of functions through which applications can interact with the kernel. These functions implement much of the operating-system functionality that does not need the full privileges of kernel code.
    - **System utilities-**The system utilities are programs that perform individual, specialized management tasks. Some system utilities may be invoked just once to initialize and configure some aspect of the system; others known as *daemons* in UNIX terminology -may run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.

  Figure illustrates the various components that make up a full Linux system.

    - The most important distinction here is between the kernel and everything else.
    - All the kernel code executes in the processor's privileged mode with full access to all the physical resources of the computer.
    - Linux refers to this privileged mode as **kernel mode.**
    - The kernel is created as a single, monolithic binary. The main reason is to improve performance.
    - Because all kernel code and data structures are kept in a single address space, no context switches are necessary when a process calls an operating-system function or when a hardware interrupt is delivered.
    - Not only the core scheduling and virtual memory code but *all* kernel code, including all device drivers, file systems, andnetworking code.
  - ✓ The kernel does not necessarily need to know in advance which modules may be loaded-they are truly independent loadable components.
  - ✓ The Linux kernel forms the core of the Linux operating system.
  - ✓ It provides all the functionality necessary to run processes, and it provides system services to give arbitrated and protected access to hardware resources.



- ✓ The system libraries provide many types of functionality.
- ✓ At the simplest level, they allow applications to make kernel-system service requests.
- ✓ Making a system call involves transferring control from unprivileged user mode to privileged kernel mode.
- ✓ The libraries take care of collecting the system-call arguments and, if necessary, arranging those arguments in the special form necessary to make the system call.
- ✓ The libraries may also provide more complex versions of the basic system calls. The libraries also provide routines that do not correspond to system calls at all, such as sorting algorithms, mathematical functions, and string-manipulation routines.
- ✓ All the functions necessary to support the running of UNIX or POSIX applications are implemented here in the system libraries.
- ✓ The Linux system includes a wide variety of user-mode programs-both system utilities and user utilities.
- ✓ The system utilities include all the programs necessary to initialize the system, such as those to configure network devices and to load kernel modules.
- ✓ Continually running server programs also com1.t as system utilities; such programs handle user login requests, incoming network connections, and the printer queues.

5.18 Kernel Modules

- ✓ The Linux kernel has the ability to load and unload arbitrary sections of kernel code on demand.
- ✓ These loadable kernel modules run in privileged kernel mode and as a consequence have full access to all the hardware capabilities of the machine on which they run.
- ✓ Kernel modules are convenient for several reasons.
- ✓ Linux' s source code is free, so anybody wanting to write kernel code is able to compile a modified kernel and to reboot to load that new functionality.
- ✓ If you use kernel modules, the driver can be compiled on its own and loaded into the already-running kernel.
- ✓ Once a new driver is written, it can be distributed as a module so that other users can benefit from it without having to rebuilt their kernels.
- ✓ The kernel's module interface allows third parties to write and distribute, on their own terms, device drivers or file systems.
- ✓ Kernel modules allow a Linux system to be set up with a standard minimal Kernel, without any extra device drivers built in.

- ✓ Any device drivers that the user needs can be either loaded explicitly by the system at startup or loaded automatically by the system on demand and unloaded when not in use.
- ✓ The module support under Linux has three components:
  - The **module management** allows modules to be loaded into memory and to talk to the rest of the kernel.
  - The **driver registration** allows modules to tell the rest of the kernel that a new driver has become available.
  - A **conflict-resolution mechanism** allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

- **Module Management**

  - ✓ Loading a module requires more than just loading its binary contents into kernel memory.
  - ✓ The system must also make sure that any references the module makes to kernel symbols or entry points are updated to point to the correct locations in the kernel's address space.
  - ✓ Linux deals with this reference updating by splitting the job of module loading into two separate sections: the management of sections of module code in kernel memory and the handling of symbols that modules are allowed to reference.
  - ✓ Linux maintains an internal symbol table in the kernel.
  - ✓ This symbol table does not contain the full set of symbols defined in the kernel during the latter's compilation; rather, a symbol must be exported explicitly by the kernel.
  - ✓ The set of exported symbols constitutes a well-defined interface by which a module can interact with the kernel.
  - ✓ When a module is to be loaded into the kernel, a system utility first scans the module for these unresolved references.
  - ✓ All symbols that still need to be resolved are looked up in the kernel's symbol table, and the correct addresses of those symbols in the currently running kernel are substituted into the module's code.
  - ✓ Only then is the module passed to the kernel for loading. If the system utility cannot resolve any references in the module by looking them up in the kernel's symbol table, then the module is rejected.
  - ✓ The loading of the module is performed in two stages.
  - ✓ First, the module loader utility asks the kernel to reserve a continuous area of virtual kernel memory for the module.
  - ✓ The kernel returns the address of the memory allocated, and the loader utility can use this address to relocate the module's machine code to the correct loading address.
  - ✓ A second system call then passes the module, plus any symbol table that the new module wants to export, to the kernel.
  - ✓ The module itself is now copied verbatim into the previously allocated space, and the kernel's symbol table is updated with the new symbols for possible use by other modules not yet loaded.
  - ✓ The final module-management component is the module requestor.
  - ✓ The kernel defines a communication interface to which a module-management program can connect.

- ✓ With this connection established, the kernel will inform the management process whenever a process requests a device driver, file system, or network service that is not currently loaded and will give the manager the opportunity to load that service.
- ✓ The original service request will complete once the module is loaded.

- **Driver Registration**

- ✓ Once a module is loaded, it remains no more than an isolated region of memory until it lets the rest of the kernel know what new functionality it provides.
- ✓ The kernel maintains dynamic tables of all known drivers and provides a set of routines to allow drivers to be added to or removed from these tables at any time.
- ✓ The kernel makes sure that it calls a module's startup routine when that module is loaded and calls the module's cleanup routine before that module is unloaded: these routines are responsible for registering the module's functionality.
- ✓ A module may register many types of drivers and may register more than one driver if it wishes.
- ✓ Registration tables include the following items:
  - o **Device drivers**- These drivers include character devices (such as printers/terminals/ and mice) *1* block devices (including all disk drives) *1* and network interface devices.
  - o **File systems**- The file system may be anything that implements Linux's virtual-file-system calling routines. It might implement a format for storing files on a disk, but it might equally well be a network file system, such as NFS1 or a virtual file system whose contents are generated on demand/ such as Linux's /proc file system.
  - o **Network protocols**- A module may implement an entire networking protocol such as IPX1 or simply a new set of packet-filtering rules for a network firewall.
  - o **Binary format**- This format specifies a way of recognizing/ and loading/ a new type of executable file.

- **Conflict Resolution**

- ✓ Commercial UNIX implementations are usually sold to run on a vendor/s own hardware.
- ✓ One advantage of a single-supplier solution is that the software vendor has a good idea about what hardware configurations arepossible.
- ✓ The problem of managing the hardware configuration becomes more severe when modular device drivers are supported/ since the currently active set of devices becomes dynamically variable.
- ✓ Linux provides a central conflict-resolution mechanism to help arbitrate access to certain hardware resources.
- ✓ Its aims are as follows:
  - o To prevent modules from clashing over access to hardware resources.
  - o To prevent auto probes-device-driver probes that auto-detect device configuration-from interfering with existing device drivers.

- To resolve conflicts among multiple drivers trying to access the same hardware-for example, as when both the parallel printer driver and the parallel-line IP (PUP) network driver try to talk to the parallel printer port.

- ✓ To these ends/ the kernel maintains lists of allocated hardware resources.
- ✓ The PC has a limited number of possible I/0 ports (addresses in its hardware I/0 address space), interrupt lines/ and DMA channels.
- ✓ When any device driver wants to access such a resource, it is expected to reserve the resource with the kernel database first.
- ✓ This requirement incidentally allows the system administrator to determine exactly which resources have been allocated by which driver at any given point.
- ✓ A module is expected to use this mechanism to reserve in advance any hardware resources that it expects to use.
- ✓ If the reservation is rejected because the resource is not present or is already in use, then it is up to the module to decide how to proceed.
- ✓ It may fail its initialization and request that it be unloaded if it cannot continue, or it may carry on, using alternative hardware resources.

5.19 Process management

- ✓ A process is the basic context within which all user-requested activity is serviced within the operating system.
- ✓ To be compatible with other UNIX systems, Linux must use a process model similar to those of other versions of UNIX.
- ✓ The traditional UNIX process model.

- **The fork() and exec() Process Model**

  - ✓ The basic principle of UNIX process management is to separate two operations: the creation of a process and the running of a new program.
  - ✓ A new process is created by the fork() system call, and a new program is run after a call to exec().
  - ✓ These are two distinctly separate functions.
  - ✓ A new process may be created with fork() without a new program being run-the new sub process simply continues to execute exactly the same program that the first (parent) process was running.
  - ✓ Equally, running a new program does not require that a new process be created first: any process may call exec() at any time.
  - ✓ The currently rumung program is immediately terminated, and the new program starts executing in the context of the existing process.
  - ✓ This model has the advantage of great simplicity.
  - ✓ It is not necessary to specify every detail of the environment of a new program in the system call that runs that program; the new program simply runs in its existing environment.
  - ✓ If a parent process wishes to modify the environment in which a new program is to be run, it can fork and then, still running the original program in a child process, make any system calls it requires to modify that child process before finally executing the new program.
  - ✓ process properties fall into three groups: the process identity, environment, and context.

- **Process Identity**

  - ✓ A process identity consists mainly of the following items:
    - **Process ID (PID)**- Each process has a Lmique identifier. The PID is used to specify the process to the operating system when an application makes a system call to signal, modify, or wait for the process. Additional identifiers associate the process with a process group (typically, a tree of processes forked by a single user command) and login session.
    - **Credentials**- Each process must have an associated user ID and one or more group IDs (user groups are discussed in Section 10.6.2) that determine the rights of a process to access system resources and files.
    - **Personality**- Process personalities are not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls. Personalities are primarily used by emulation libraries to request that system calls be compatible with certain varieties of UNIX.

- **Process Environment**

  - ✓ A process's environment is inherited from its parent and is composed of two null-terminated vectors: the argument vector and the environment vector.
  - ✓ The argument vector simply lists the command-line arguments used to invoke the running program; it conventionally starts with the name of the program itself.
  - ✓ The environment vector is a list of "NAME= VALUE" pairs that associates named environment variables with arbitrary textual values.
  - ✓ The environment is not held in kernel memory but is stored in the process's own user-mode address space as the first datum at the top of the process's stack.
  - ✓ The argument and environment vectors are not altered when a new process is created.
  - ✓ A completely new environment is set up when a new program is invoked.
  - ✓ On calling exec (),a process must supply the environment for the new program.
  - ✓ The kernel passes these environment variables to the next program, replacing the process's current environment.
  - ✓ The kernel otherwise leaves the environment and command-line vectors alone.
  - ✓ The passing of environment variables from one process to the next and the inheriting of these variables by the children of a process provide flexible ways to pass information to components of the user-mode system software.
  - ✓ Various important environment variables have conventional meanings to related parts of the system software.
  - ✓ For example, the TERM variable is set up to name the type of terminal connected to a user's login session.

- **Process Context**

  - ✓ process context is the state of the running program at any one time; it changes constantly.
  - ✓ Process context includes the following parts:
    - **Scheduling context** The most important part of the process context is its scheduling context-the information that the scheduler needs to suspend and

restart the process. This information includes saved copies of all the process's registers. Floating-point registers are stored separately and are restored only when needed.

- **Accounting** The kernel maintains accounting information about the resources currently being consumed by each process and the total resources consumed by the process in its entire lifetime so far.
- **File table** The file table is an array of pointers to kernel file structures. When making file-I/O system calls, processes refer to files by their index into this table.
- **File-system context** Whereas the file table lists the existing open files, the file-system context applies to requests to open new files. The current root and default directories to be used for new file searches are stored here.
- **Signal-handler table** UNIX systems can deliver asynchronous signals to a process in response to various external events. The signal-handler table defines the routine in the process's address space to be called when aspecific signals arrive.
- **Virtual memory context** The virtual memory context describes the full contents of a process's private address space; we discuss it in Section 21.6.

- **Processes and Threads**

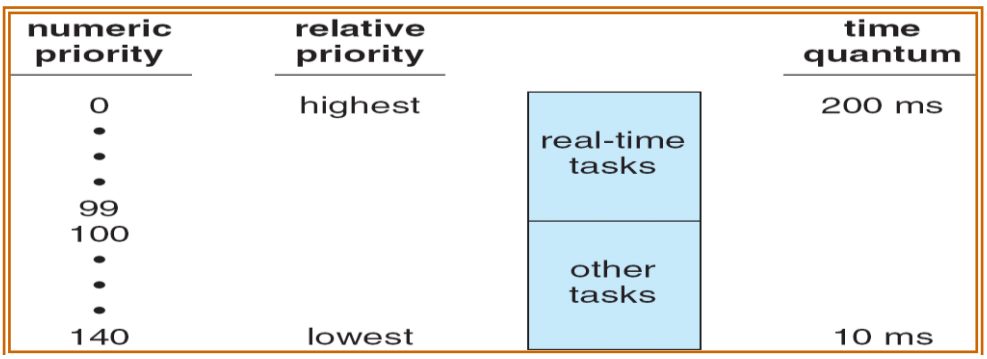
  - ✓ Linux provides the fork() system call with the traditional functionality of duplicating a process.
  - ✓ Linux also provides the ability to create threads using the clone() system call.
  - ✓ However, Linux does not distinguish between processes and threads. Linux uses the term task.
  - ✓ When clone()is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks.
  - ✓ Some of these flags are:
    - CLONE_FS-file system information is shared.
    - CLONE_VM-the same memory space is shared.
    - CLONE_SIGHAND-Signals handlers are shared.
    - CLONE_FILES-the set of open files are shared.
  - ✓ Thus, if clone() is passed the flags CLONE_FS, CLONE_VM, CLONE_SIGHAND, and CLONE_FILES, the parent and child tasks will share the same file-system information (such as the current working directory), the same memory space, the same signal handlers, and the same set of open files.
  - ✓ If none of these flags is set when clone () is invoked, no sharing takes place, resulting in functionality similar to the fork() system call.
  - ✓ The lack of distinction between processes and threads is possible because Linux does not hold a process's entire context within the main process data structure; rather, it holds the context within independent sub contexts.
  - ✓ Thus, a process's file-system context, file-descriptor table, signal-handler table, and virtual memory context are held in separate data structures.
  - ✓ The process data structure simply contains pointers to these other structures, so any number of processes can easily share a sub context by pointing to the same sub context.
  - ✓ The arguments to the clone () system call tell it which sub contexts to copy, and which to share, when it creates a new process.

- ✓ The new process always is given a new identity and a new scheduling context according to the arguments passed.

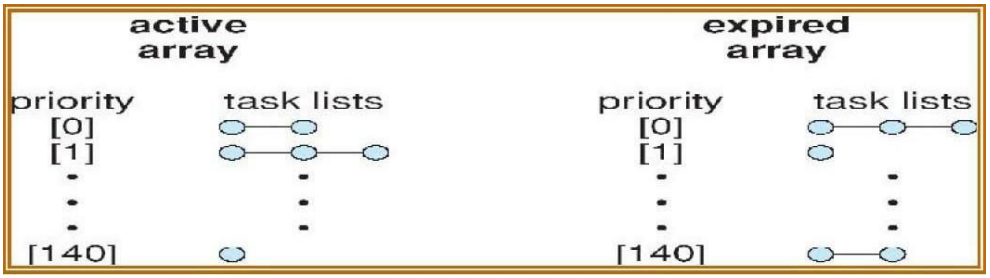
5.20 Scheduling

- ✓ Scheduling is the job of allocating CPU time to different tasks within an operating system.
- ✓ Aspect of scheduling to Linux: the running of the various kernel tasks.
- ✓ Kernel tasks encompass both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver.

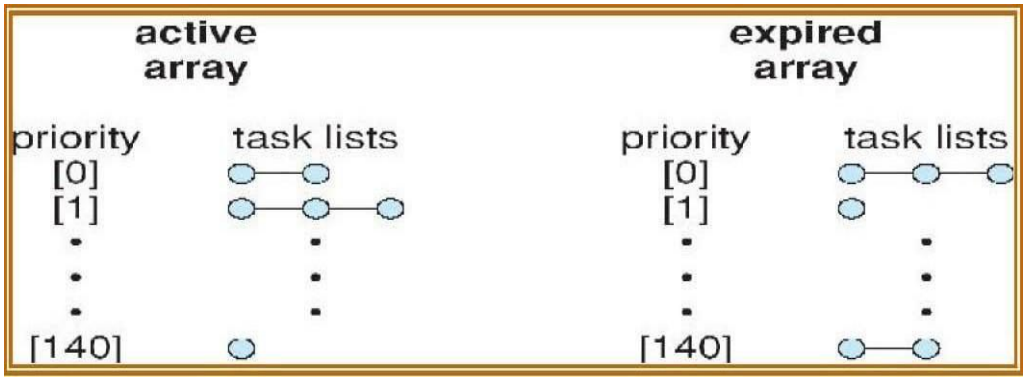
- **Process Scheduling**
  - ✓ Linux has two separate process-scheduling algorithms.
  - ✓ One is a time-sharing algorithm for fair, preemptive scheduling among multiple processes; the other is designed for real-time tasks, where absolute priorities are more important than fairness.
  - ✓ Problems with the traditional UNIX scheduling algorithm, which does not provide adequate support for SMP systems and does not scale well as the number of tasks on the system grows.
  - ✓ Version 2.5 of the kernel provides a scheduling algorithm that runs in constant time-known as 0(1)-regardless of the number of tasks on the system.
  - ✓ The new scheduler also provides increased support for SMP, including processor affinity and load balancing, as well as maintaining fairness and support for interactive tasks.
  - ✓ The Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: a real-time range from 0 to 99 and a nice value ranging from 100 to 140.
  - ✓ These two ranges map into a global priority scheme whereby numerically lower values indicate higher priorities.
  - ✓ The Linux scheduler assigns higher-priority tasks longer time quanta and lower-priority tasks shorter time quanta and vise-versa.



The relationship between priorities and time scheduling



- ✓ The relationship between priorities and time-slice length is shown in Figure 21.2.
- ✓ When a task has exhausted its time slice, it is considered expired and is not eligible for execution again until all other tasks have also exhausted their time quanta.
- ✓ The kernel maintains a list of all runnable tasks in a runqueue data structure.
- ✓ Because of its support for SMP, each processor maintains its own run queue and schedules itself independently. Each run queue contains two priority arrays-active and expired.
- ✓ The active array contains all tasks with time remaining in their time slices, and the expired array contains all expired tasks. Each of these priority arrays includes a list of tasks indexed according to priority (Figure 21.3).
- ✓ When all tasks have exhausted their time slices (that is, the active array is empty), the two priority arrays are exchanged as the expired array becomes the active array and vice-versa.



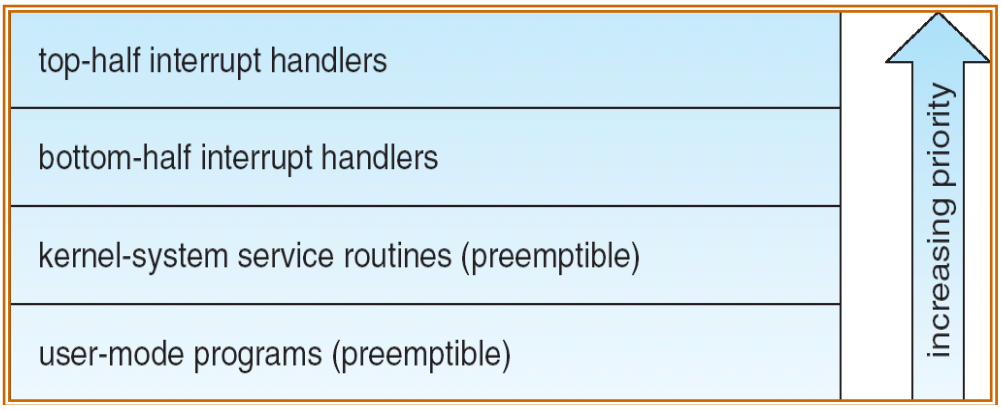
list of tasks indexed according to priority

- ✓ Tasks are assigned dynamic priorities that are based on the *nice* value plus or minus a value up to the value 5 based upon the interactivity of the task.
- ✓ A task's interactivity is determined by how long it has been sleeping while waiting for I/0. Linux' s real-time scheduling is simpler.
- ✓ Linux implements the two real time scheduling classes required by POSIX.lb: first-come, first-served (FCFS) and round-robin.
- ✓ Processes with different priorities can compete with one another to some extent in time-sharing scheduling; in real-time scheduling, however, the scheduler always runs the process with the highest priority. Among processes of equal priority, it runs the process that has been waiting longest.
- ✓ The only difference between FCFS and round-robin scheduling is that FCFS processes continue to run until they either exit or block, whereas a round-robin process will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin processes of equal priority will automatically time-share among themselves.
- ✓ Unlike routine time-sharing tasks, real-time tasks are assigned static priorities.

- **Kernel Synchronization**

- ✓ The way the kernel schedules its own operations is fundamentally different from the way it schedules processes.
- ✓ A request for kernel-mode execution can occur in two ways.

- ✓ A running program may request an operating-system service, either explicitly via a system call or implicitly-for example, when a page fault occurs. Alternatively, a device controller may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.
- ✓ The problem posed to the kernel is that all these tasks may try to access the same internal data structures.
- ✓ If one kernel task is in the middle of accessing some data structure when an interrupt service routine executes, then that service routine cannot access or modify the same data without risking data corruption.
- ✓ This fact relates to the idea of critical sections-portions of code that access shared data and that must not be allowed to execute concurrently.
- ✓ As a result, kernel synchronization involves much more than just process scheduling.
- ✓ A framework is required that allows kernel tasks to run without violating the integrity of shared data.
- ✓ With Version 2.6, the Linux kernel became fully preemptive; so a task can now be preempted when it is running in the kernel.
- ✓ The Linux kernel provides spinlocks and semaphores (as well as reader writer versions of these two locks) for locking in the kernel.
- ✓ On SMP machines, the fundamental locking mechanism is a spinlock; the kernel is designed so that the spinlock is held only for short durations.
- ✓ This pattern is summarized below:
- ✓ The kernel is not pre-emptible if a kernel-mode task is holding a lock.
- ✓ To enforce this rule, each task in the system has a thread-info structure that includes the field preempt_count, which is a counter indicating the number of locks being held by the task.
- ✓ The counter is incremented when a lock is acquired and decremented when a lock is released.
- ✓ If the value of preempt_count for the task currently running is greater than zero, it is not safe to preempt the kernel as this task currently holds a lock.
- ✓ If the count is zero, the kernel can safely be interrupted, assuming there are no outstanding calls to preempt_disable ().
- ✓ When the lock is held for short durations. When a lock must be held for longer periods, semaphores are used.
- ✓ The second protection technique used by Linux applies to critical sections that occur in interrupt service routines.
- ✓ The basic tool is the processor's interrupt-control hardware.
- ✓ By disabling interrupts (or using spinlocks) during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access to shared data structures.
- ✓ The Linux kernel uses a synchronization architecture that allows long critical sections to run for their entire duration without having interrupts disabled.
- ✓ This ability is especially useful in the networking code.



interrupt protection levels

- ✓ Linux implements this architecture by separating interrupt service routines into two sections: the top half and the bottom half.
- ✓ The **Top half** is a normal interrupt service routine that runs with recursive interrupts disabled; interrupts of a higher priority may interrupt the routine, but interrupts of the same or lower priority are disabled.
- ✓ The **Bottom half** of a service routine is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves.
- ✓ The bottom-half scheduler is invoked automatically whenever an interrupt service routine exits.
- ✓ This separation means that the kernel can complete any complex processing that has to be done in response to an interrupt without worrying about being interrupted itself.
- ✓ If another interrupt occurs while a bottom half is executing, then that interrupt can request that the same bottom half execute, but the execution will be deferred until the one currently running completes.
- ✓ Each execution of the bottom half can be interrupted by a top half but can never be interrupted by a similar bottom half.
- ✓ Figure 21.4 summarizes the various levels of interrupt protection within the kernel.

▪ **Symmetric Multiprocessing**

- ✓ The Linux 2.0 kernel was the first stable Linux kernel to support **Symmetric Multiprocessor(SMP)** hardware, allowing separate processes to execute in parallel on separate processors.
- ✓ In Version 2.2 of the kernel, a single kernel spinlock (sometimes termed BKL for "big kernel lock") was created to allow multiple processes (running on different processors) to be active in the kernel concurrently.
- ✓ However, the BKL provided a very coarse level of locking granularity. Later releases of the kernel made the SMP implementation more scalable by splitting this single kernel spinlock into multiple locks, each of which protects only a small subset of the kernel's data structures.

5.21 Memory Management

- ✓ Memory management under Linux has two components.
- ✓ The first deals with allocating and freeing physical memory.
- ✓ The second handles virtual memory, which is memory mapped into the address space of running processes.

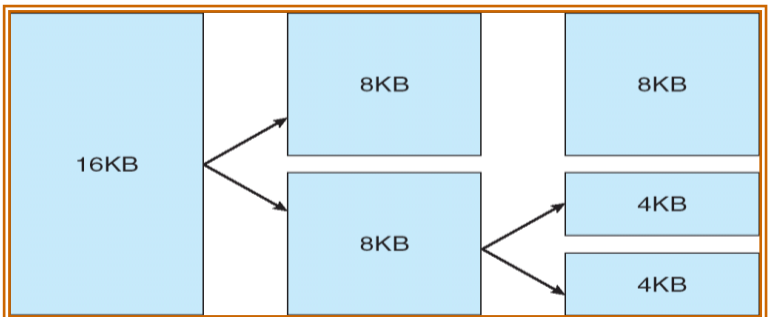
- **Management of Physical Memory**

  - ✓ Linux separates physical memory into three different zones, or regions:
    - ZONE_DMA
    - ZONE_NORMAL
    - ZONE_HIGHMEM
  - ✓ These zones are architecture specific.The relationship of zones and physical addresses on the Intel80x86 architecture is shown in Figure 21.5.
  - ✓ The kernel maintains a list of free pages for each zone. When a request for physical memory arrives, the kernel satisfies the request using the appropriate zone.
  - ✓ The primary physical-memory manager in the Linux kernel is the **page allocator.** Each zone has its own allocator, which is responsible for allocating and freeing all physical pages for the zone and is capable of allocating ranges of physically contiguous pages on request.
  - ✓ The allocator uses a buddy system to keep track of available physical pages. In this scheme, adjacent units of allocatable memory are paired together (hence its name). Each allocatable memory region has an adjacent partner (or buddy).

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

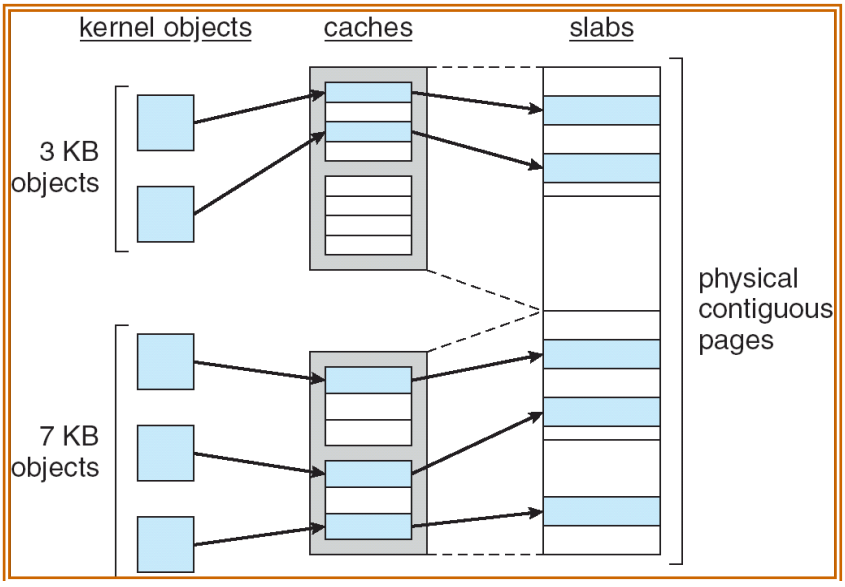
*relationship of zones and physical addresses on the intel 80*86*

  - ✓ Whenever two allocated partner regions are freed up, they are combined to form a larger region-a *buddy heap.*
  - ✓ That larger region also has a partner, with which it can combine to form a still larger free region.
  - ✓ Conversely, if a small memory request cannot be satisfied by allocation of an existing small free region, then a large free region will be subdivided into two partners to satisfy the request.
  - ✓ Separate linked lists are used to record the free memory regions of each allowable size;
  - ✓ Figure shows an example of buddy-heap allocation. A 4-KB region is being allocated, but the smallest available region is 16 KB. The region is broken up recursively until a piece of the desired size is available.
  - ✓ The kmalloc () variable-length allocator; the slab allocator, used for allocating memory for kernel data structures; and the page cache, used for caching pages belonging to files.



Splitting of memory in the buddy system

- ✓ Many components of the Linux operating system need to allocate entire pages on request, but often smaller blocks of memory are required.
- ✓ The kernel provides an additional allocator for arbitrary-sized requests, where the size of a request is not known in advance and may be only a few bytes.
- ✓ kmalloc () service allocates entire pages on demand but then splits them into smaller pieces.
- ✓ The kernel maintains lists of pages in use by the kmalloc () service.
- ✓ Another strategy adopted by Linux for allocating kernel memory is known as slab allocation.
- ✓ A **slab** is used for allocating memory for kernel data structures and is made up of one or more physically contiguous pages.
- ✓ A **Cache** consists of one or more slabs.
- ✓ There is a single cache for each unique kernel data structure.
- ✓ Each cache is populated with that are instantiations of the kernel data structure the cache represents.
- ✓ The relationship among slabs, caches, and objects is shown in Figure.



Slab allocator in Linux

- ✓ The figure shows two kernel objects 3 KB in size and three objects 7 KB in size.
- ✓ These objects are stored in the respective caches for 3-KB and 7-KB objects.

- ✓ The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects are allocated to the cache. The number of objects in the cache depends on the size of the associated slab.
- ✓ Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor.
- ✓ In Linux, a slab may be in one of three possible states:
  - o Full- All objects in the slab are marked as used.
  - o Empty-All objects in the slab are marked as free.
  - o Partial-The slab consists of both used and free objects.
- ✓ The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exist, a free object is assigned from an empty slab.
- ✓ If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.
- ✓ Two other main subsystems in Linux do their own management of physical pages: the page cache and the virtual memory system.
- ✓ These systems are closely related to one another.
- ✓ The page cache is the kernel's main cache for block devices and memory-mapped files and is the main mechanism through which I/0 to these devices is performed.

- **Virtual Memory**

- ✓ The Linux virtual memory system is responsible for maintaining the address space visible to each process.
- ✓ It creates pages of virtual memory on demand and manages loading those pages from disk and swapping them back out to disk as required.
- ✓ Under Linux, the virtual memory manager maintains two separate views of a process's address space: as a set of separate regions and as a set of pages.
- ✓ The first view of an address space is the logical view, describing instructions that the virtual memory system has received concerning the layout of the address space.
- ✓ In this view, the address space consists of a set of non overlapping regions, each region representing a continuous, page-aligned subset of the address space.
- ✓ Each region is described internally by a single vm_area_struct structure that defines the properties of the region, including process's read, write, and execute permissions in the region as well as information about any files associated with the region.
- ✓ The regions for each 824 Chapter 21 address space are linked into a balanced binary tree to allow fast lookLlp of the region corresponding to any virtual address.
- ✓ The kernel also n"laintains a second, physical view of each address space.
- ✓ This view is stored in the hardware page tables for the process.
- ✓ The pagetable entries identify the exact current location of each page of virtual mernory, whether it is on disk or in physical memory.
- ✓ The physical view is managed by a set of routines, which are invoked from the kernel's software-interrupt handlers whenever a process tries to access a page that is not currently present in the page tables.
- ✓ Each vm_area_struct in the address-space description contains a field that points to a table of functions that implement the key page-management functions for any given virtual memory region.
- ✓ All requests to read or write an unavailable page are eventually dispatched to the appropriate handler in the function table for the vm_area_struct, so that the

central memory management routines do not have to know the details of managing each possible type of memory region.

- **Virtual Memory Regions**

  - ✓ Linux implements several types of virtual memory regions.
  - ✓ One property that characterizes virtual memory is the backing store for the region, which describes where the pages for the region come from.
  - ✓ Most memory regions are backed either by a file or by nothing.
  - ✓ A region backed by nothing is the simplest type of virtual memory region.
  - ✓ Such a region represents demand-zero memory: when a process tries to read a page in such a region, it is simply given back a page of memory filled with zeros.
  - ✓ A region backed by a file acts as a viewport onto a section of that file.
  - ✓ Whenever the process tries to access a page within that region, the page table is filled with the address of a page within the kernel's page cache corresponding to the appropriate offset in the file.
  - ✓ The same page of physical memory is used by both the page cache and the process's page tables, so any changes made to the file by the file system are immediately visible to any processes that have mapped that file into their address space.
  - ✓ Any number of processes can map the same region of the same file, and they will all end up using the same page of physical memory for the purpose.
  - ✓ A virtual memory region is also defined by its reaction to writes.
  - ✓ The mapping of a region into the process's address space can be either *private* or *shared.*
  - ✓ If a process writes to a privately mapped region, then the pager detects that a copy-on-write is necessary to keep the changes local to the process.
  - ✓ In contrast, writes to a shared region result in updating of the object mapped into that region, so that the change will be visible immediately to any other process that is mapping that object.

- **Lifetime of a Virtual Address Space**

  - ✓ The kernel will create a new virtual address space in two situations: when a process runs a new program with the exec() system call and when a new process is created by the fork() system call.
  - ✓ The first case is easy.
  - ✓ When a new program is executed, the process is given a new, completely empty virtual address space.
  - ✓ It is up to the routines for loading the program.
  - ✓ to populate the address space with virtual memory regions.
  - ✓ The second case, creating a new process with fork(), involves creating a concplete copy of the existing process's virtual address space.
  - ✓ The kernel copies the parent process's vm_area_struct descriptors, then creates a new set of page tables for the child.
  - ✓ The parent's page tables are copied directly into the child's, and the reference count of each page covered is incremented; thus, after the fork, the parent and child share the same physical pages of memory in their address spaces.

- ✓ A special case occurs when the copying operation reaches a virtual memory region that is mapped privately.
- ✓ Any pages to which the parent process has written within such a region are private, and subsequent changes to these pages by either the parent or the child must not update the page in the other process's address space.
- ✓ When the page-table entries for such regions are copied, they are set to be read only and are marked for copy-on-write.
- ✓ As long as neither process modifies these pages, the two processes share the same page of physical memory.
- ✓ However, if either process tries to modify a copy-on-write page, the reference count on the page is checked.
- ✓ If the page is still shared, then the process copies the page's contents to a brand-new page of physical memory and uses its copy instead.
- ✓ This mechanism ensures that private data pages are shared between processes whenever possible; copies are made only when absolutely necessary.

- **Swapping and Paging**

- ✓ An important task for a virtual memory system is to relocate pages of memory from physical memory out to disk when that in memory is needed.
- ✓ Early UNIX systems performed this relocation by swapping out the contents of entire processes at once, but modern versions of UNIX rely more on paging-the movement of individual pages of virtual memory between physical memory and disk.
- ✓ The paging system can be divided into two sections. First, it decides which to write out to disk and when to write them. Second, it carries out the transfer and pages data back into physical memory when they are needed again.
- ✓ Linux's pageout policy uses a modified version of the standard clock (or second-chance) algorithm.
- ✓ amultiplepass clock is used, and every page has an *age* that is adjusted on each pass of the clock.
- ✓ The age is more precisely a measure of the page's youthfulness, or how much activity the page has seen recently.
- ✓ Frequently accessed pages will attain a higher age value, but the age of infrequently accessed pages will drop toward zero with each pass.
- ✓ This age valuing allows the pager to select pages to page out based on a least frequently used (LFU) policy.
- ✓ The paging mechanism supports paging both to dedicated swap devices and partitions and to normal files, although swapping to a file is significantly slower due to the extra overhead incurred by the file system.
- ✓ Blocks are allocated from the swap devices according to a bitmap of used blocks, which is maintained in physical memory at all times.
- ✓ The allocator uses a next-fit algorithm to try to write out pages to continuous runs of disk blocks for improved performance.
- ✓ The allocator records the fact that a page has been paged out to disk by using a feature of the page tables on modern processors i.e, the page-table entry's page-not-present bit is set, allowing the rest of the page table entry to be filled with an index identifying where the page has been written.

- **Kernel Virtual Memory**

  - ✓ Linux reserves for its own internal use a constant, architecture-dependent region of the virtual address space of every process.
  - ✓ The page-table entries that map to these kernel pages are marked as protected, so that the pages are not visible or modifiable when the processor is running in user mode. This kernel virtual memory area contains two regions.
  - ✓ The first is a static area that contains page-table references to every available physical page of memory in the system, so that a simple translation from physical to virtual addresses occurs when kernel code is run.
  - ✓ The remainder of the kernel's reserved section of address space is not reserved for any specific purpose.
  - ✓ Page-table entries in this address range can be modified by the kernel to point to any other areas of memory.
  - ✓ The kernel provides a pair of facilities that allow processes to use this virtual memory.
  - ✓ The vmalloc () function allocates an arbitrary number of physical pages of memory that may not be physically contiguous into a single region of virtually contiguous kernel memory.
  - ✓ The vremap () function maps a sequence of virtual addresses to point to an area of memory used by a device driver form memory-mapped I/0.

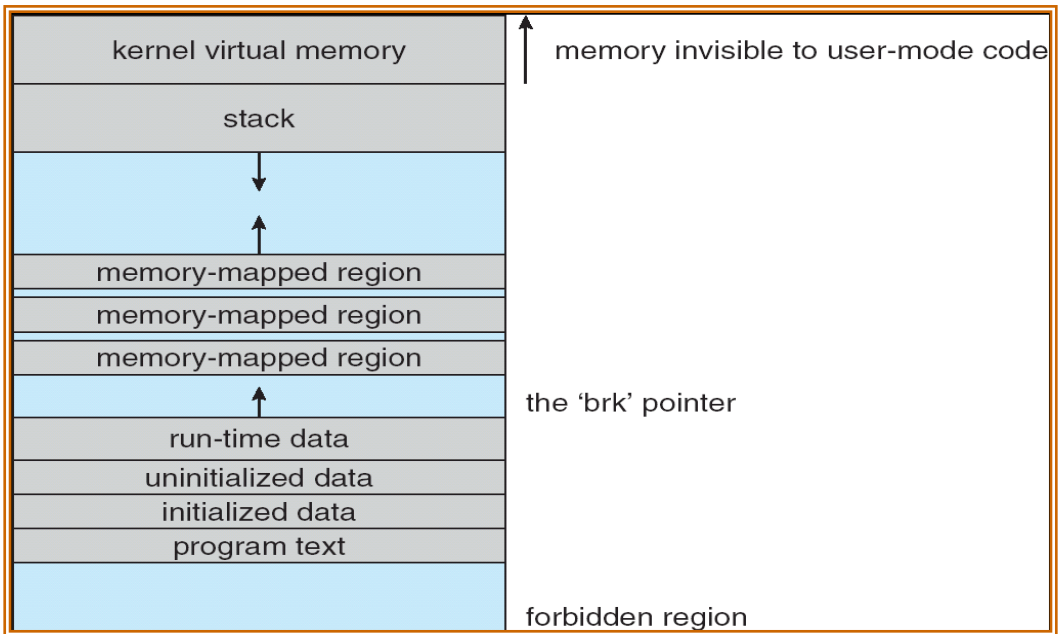
- **Execution and Loading of User Programs**

  - ✓ The Linux kernel's execution of user programs is triggered by a call to the exec() system call.
  - ✓ This exec() call commands the kernel to run a new program within the current process, completely overwriting the current execution context with the initial context of the new program.
  - ✓ The first job of this system service is to verify that the calling process has permission rights to the file being executed.
  - ✓ Once that matter has been checked, the kernel invokes a loader routine to start running the program. The loader does not necessarily load the contents of the program file into physical memory, but it does at least set up the mapping of the program into virtual memory.
  - ✓ There is no single routine in Linux for loadil1.g a new program. Instead, Linux maintains a table of possible loader functions, and it gives each such function the opportunity to try loading the given file when an exec() system call is made.
  - ✓ Newer Linux systems use the more modern ELF format, now supported by most current UNIX implementations.
  - ✓ ELF has a number of advantages over a. out, including flexibility and extensibility.
  - ✓ New sections can be added to an ELF binary (for example, to add extra debugging information) without causing the loader routines to become confused.
  - ✓ By allowing registration of multiple loader routines, Linux can easily support the ELF and a. out binary formats in a single running system.

  - **Mapping of Programs into Memory**

    - ✓ The pages of the binary file are mapped into regions of virtual memory.

- ✓ Only when the program tries to access a given page a page fault result in the loading of that page into physical memory using demand paging.
- ✓ It is the responsibility of the kernel's binary loader to set up the initial memory mapping.
- ✓ An ELF-format binary file consists of a header followed by several page-aligned sections. The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory.
- ✓ Figure 21.8 shows the typical layout of memory regions set up by the ELF loader. In a reserved region at one end of the address space sits the kernel in its own privileged region of virtual memory inaccessible to normal user-mode programs.
- ✓ The rest of virtual memory is available to applications, which can use the kernel's memory-mapping functions to create regions that map a portion of a file or that are available for application data.
- ✓ The loader's job is to set up the initial memory mapping to allow the execution of the program to start. The regions that need to be initialized include the stack and the program's text and data regions.
- ✓ The stack is created at the top of the user-mode virtual memory; it grows downward toward lower-numbered addresses.
- ✓ It includes copies of the arguments and environment variables given to the program in the exec() system call.
- ✓ The other regions are created near the bottom end of virtual memory.
- ✓ The sections of the binary file that contain program text or read-onlydata are mapped into memory as a write-protected region.
- ✓ Writable initialized data are mapped next; then any uninitialized data are mapped in as a private demand-zero region.



Memory layout for ELF programs

- ✓ Directly beyond these fixed-sized regions is a variable-sized region that programs can expand as needed to hold data allocated at run time.
- ✓ Each process has a pointer, brk, that points to the current extent of this data region, and processes can extend or contract their brk region with a single system call -sbrkO.

- ✓ Once these mappings have been set up, the loader initializes the process's program-counter register with the starting point recorded in the ELF header, and the process can be scheduled.

- **Static and dynamic Linking**

  - ✓ In the simplest case, the necessary library functions are embedded directly in the program's executable binary file.
  - ✓ Such a program is statically linked to its libraries, and statically linked executables can commence running as soon as they are loaded.
  - ✓ The main disadvantage of static linking is that every program generated must contain copies of exactly the same common system library functions.
  - ✓ It is much more efficient, in terms of both physical memory and disk-space usage, to load the system libraries into memory only once.
  - ✓ Dynamic linking allows this single loading to happen.
  - ✓ Linux implements dynamic linking in user mode through a special linker library.
  - ✓ Every dynamically linked program contains a small, statically linked function that is called when the program starts.
  - ✓ This static function just maps the link library into memory and runs the code that the function contains.
  - ✓ The link library determines the dynamic libraries required by the program and the names of the variables and functions needed from those libraries by reading the information contained in sections of the ELF binary.
  - ✓ It then maps the libraries into the middle of virtual memory and resolves the references to the symbols contained in those libraries i.e., It does not matter exactly where in memory these shared libraries are mapped: they are compiled into position-independent code (PIC), which can run at any address in memory.

5.22 File System

- ✓ In UNIX, a file does not have to be an object stored on disk or fetched over a network from a remote file server.
- ✓ Rather, UNIX files can be anything capable of handling the input or output of a stream of data.
- ✓ Device drivers can appear as files, and interprocess communication channels or network connections also look like files to the user.
- ✓ The Linux kernel handles all these types of files by hiding the implementation details of any single file type behind a layer of software, the virtual file system (VFS).

- **The Virtual File System**

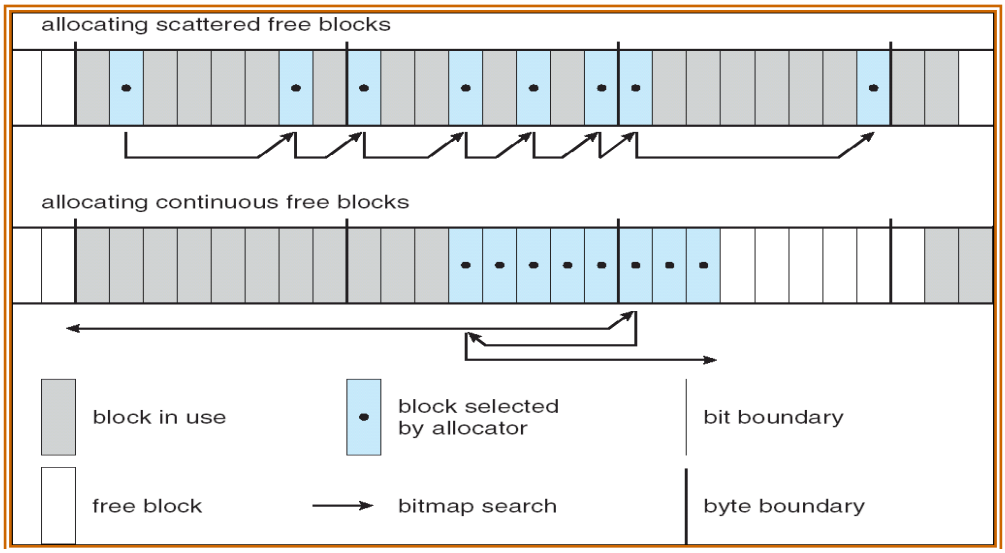
  - ✓ The Linux VFS is designed around object-oriented principles.
  - ✓ It has two components: a set of definitions that specify what file-system objects are allowed to look like and a layer of software to manipulate the objects.
  - ✓ The VFS defines four main object types:
    - An **inode object** represents an individual file.
    - A **file object** represents an open file.
    - A **superblock object** represents an entire file system.
    - A **dentry object** represents an individual directory entry.

- ✓ For each of these four object types, the VFS defines a set of operations.
- ✓ Every object of one of these types contains a pointer to a function table.
- ✓ The function table lists the addresses of the actual functions that implement the defined operations for that object.
- ✓ For example, an abbreviated API for some of the file object's operations includes:
  - o int open (. . .) - Open a file.
  - o ssize_t read(. . .) -Read from a file.
  - o ssize_t write (. . .) - Write to a file.
  - o intmmap (. . .) - Memory-map a file.
- ✓ The complete definition of the file object is specified in the structfile_operations which is located in the file /usr/include/linux/fs.h.
- ✓ An implementation of the file object (for a specific file type) is required to implement each function specified in the definition of the file object.
- ✓ The VFS software layer can perform an operation on one of the file-system objects by calling the appropriate function from the object's function table, without havil<g to know in advance exactly what kind of object it is dealing with.
- ✓ The VFS does not know, or care, whether an inode represents a networked file, a disk file, a network socket, or a directory file.
- ✓ The appropriate function for that file's read() operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.
- ✓ The inode and file objects are the mechanisms used to access files.
- ✓ An inode object is a data structure containing pointers to the disk blocks that contain the actual file contents, and a file object represents a point of access to the data in an open file.
- ✓ File objects typically belong to a single process, but inode objects do not.
- ✓ Even when a file is no longer being used by any processes, its inode object may still be cached by the VFS to improve performance if the file is used again in the near future.
- ✓ All cached file data are linked onto a list in the file's inode object.
- ✓ The inode also maintains standard information about each file, such as the owner, size, and time most recently modified.
- ✓ Directory files are dealt with slightly differently from other files.
- ✓ The system calls for these directory operations do not require that the user open the files concerned, unlike the case for reading or writing data.
- ✓ The VFS therefore defines these directory operations in the inode object, rather than in the file object.
- ✓ The superblock object represents a connected set of files that form a self-contained file system.
- ✓ The operating-system kernel maintains a single superblock object for each disk device mounted as a file system and for each networked file system currently connected.
- ✓ The main responsibility of the superblock object is to provide access to inodes.
- ✓ The VFS identifies every inode by a unique file-system/inode number pair, and it fil<ds the inode corresponding to a particular inode number by asking the superblock object to return the inode with that number.
- ✓ Finally, a dentry object represents a directory entry that may include the name of a directory in the path name of a file (such as /usr) or the actual file (such as stdio. h). For example, the file /usr*I* include/ stdio. h contains the directory entries (1) /, (2) usr, (3) include, and (4) stdio. h. Each one of these values is represented by a separate dentry object.

- **The Linux ext2fs File System**

  - ✓ The standard on-disk file system used by Linux is called ext2fs.
  - ✓ Linuxs ext2fs has much in common with the BSD Fast File System (FFS).
  - ✓ It uses a similar mechanism for locating the data blocks belonging to a specific file, storing data-block pointers in indirect blocks throughout the file system with up to three levels of indirection.
  - ✓ As in FFS, directory files are stored on disk just like normal files, although their contents are interpreted differently.
  - ✓ Each block in a directory file consists of a linked list of entries; each entry contains the length of the entry, the name of a file, and the inode number of the inode to which that entry refers.
  - ✓ The main differences between ext2fs and FFS lie in their disk-allocation policies.
  - ✓ In FFS, the disk is allocated to files in blocks of 8 KB.
  - ✓ These blocks are subdivided into fragments of 1 KB for storage of small files or partially filled blocks at the ends of files.
  - ✓ In contrast, ext2fs does not use fragments at all but performs all its allocations in smaller units.
  - ✓ The default block size on ext2fs is 1 KB, although 2-KB and 4-KB blocks are also supported.
  - ✓ The ext2fs allocation policy comes in two parts.
  - ✓ As in FFS, an ext2fs file system is partitioned into multiple block groups.
  - ✓ FFS uses the similar concept of cylinder groups, where each group corresponds to a single cylinder of a physical disk.
  - ✓ When allocating a file, ext2fs must first select the block group for that file.
  - ✓ For data blocks, it attempts to allocate the file to the block group to which the file's inode has been allocated.
  - ✓ Within a block group, ext2fs tries to keep allocations physically contiguous if possible, reducing fragmentation if it can.
  - ✓ It maintains a bitmap of all free blocks in a block group.
  - ✓ When allocating the first blocks for a new file, it starts searching for a free block from the beginning of the block group; when extending a file, it continues the search from the block most recently allocated to the file.
  - ✓ The search is performed in two stages. First, ext2fs searches for an entire free byte in the bitmap; if it fails to find one, it looks for any free bit. The search for free bytes aims to allocate disk space in chunks of at least eight blocks where possible. Once a free block has been identified, the search is extended backward until an allocated block is encountered.
  - ✓ When a free byte is found in the bitmap, this backward extension prevents ext2fs from leaving a hole between the most recently allocated block in the previous nonzero byte and the zero byte found.
  - ✓ Once the next block to be allocated has been found by either bit or byte search, ext2fs extends the allocation forward for up to eight blocks and preallocates these extra blocks to the file.
  - ✓ This preallocation helps to reduce fragmentation during interleaved writes to separate files and also reduces the CPU cost of disk allocation by allocating multiple blocks simultaneously.
  - ✓ The preallocated blocks are returned to the free-space bitmap when the file is closed.
  - ✓ Figure illustrates the allocation policies.

- ✓ Each row represents a sequence of set and unset bits in an allocation bitmap, indicating used and free blocks on disk.
- ✓ In the first case, if we can find any free blocks sufficiently near the start of the search, then we allocate them no matter how fragmented they may be.
- ✓ The fragmentation is partially compensated for by the fact that the blocks are close together and can probably all be read without any disk seeks, and allocating them all to one file is better in the long run than allocating isolated blocks to separate files once large free areas become scarce on disk.
- ✓ In the second case, we have not immediately found a free blocsk close by, so we search forward for an entire free byte in the bitmap.



ext2fs block-allocation policies

- ✓ If we allocated that byte as a whole, we would end up creating a fragmented area of free space between it and the allocation preceding it so before allocating we back up to make this allocation flush with the allocation preceding it, and then we allocate forward to satisfy the default allocation of eight blocks.

- **Journaling**

- ✓ One popular feature in a file system is journaling, whereby modifications to the file system are sequentially written to a journal.
- ✓ A set of operations that performs a specific task is a transaction.
- ✓ Once a transaction is written to the journal it is considered to be committed, and the system call modifying the file system (write()) can return to the user process, allowing it to continue execution.
- ✓ If the system crashes, some transactions may remain in the journal.
- ✓ Those transactions were never completed to the file system even though they were committed by the operating system, so they must be completed.
- ✓ Journaling file systems are also typically faster than non-journaling systems, as updates proceed much faster when they are applied to the in-memory journal rather than directly to the on-disk data structures.
- ✓ The reason for this improvement is found in the performance advantage of sequential I/0 over random I/0.

- ✓ Costly synchronous random writes to the file system are turned into much less costly synchronous sequential writes to the file system's journal.
- ✓ Those changes in turn are replayed asynchronously via random writes to theappropriate structures.
- ✓ The overall result is a significant gain in performance of file-system metadata-oriented operations, such as file creation and deletion.
- ✓ Journaling is not provided in ext2fs.
- ✓ It is provided, however, in another common file system available for Linux systems, ext3, which is based on ext2fs.

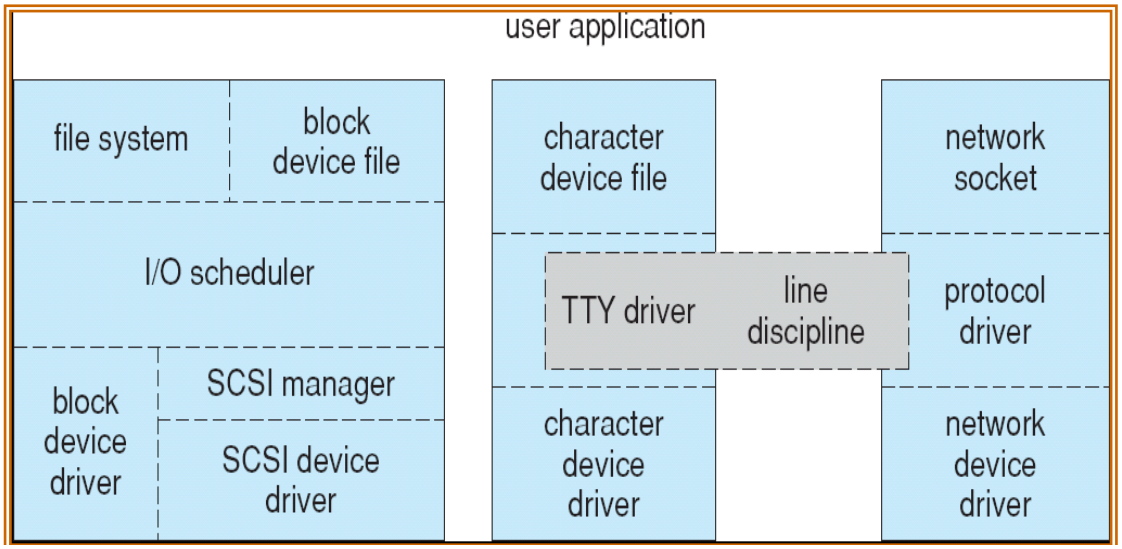
- **The Linux Proc File System**

- ✓ The Linux process file system, known as the /proc file system, is an example of a file system whose contents are not actually stored anywhere but are computed on demand according to user file I/0 requests.
- ✓ A /proc file system is not unique to Linux. SVR4 UNIX introduced a /proc file system as an efficient interface to the kernel's process debugging support.
- ✓ Each subdirectory of the file system corresponded not to a directory on any disk but rather to an active process on the current system.
- ✓ Linux implements such a /proc file system but extends it greatly by adding a number of extra directories and text files under the file system's root directory.
- ✓ These new entries correspond to various statistics about the kernel and the associated loaded drivers.
- ✓ The /proc file system provides a way for programs to access this information as plain text files; the standard UNIX user environment provides powerful tools to process such files.
- ✓ The /proc file system must implement two things: a directory structure and the file contents within.
- ✓ Because a UNIX file system is defined as a set of file and directory inodes identified by their inode numbers, the /proc file system must define a unique and persistent inode number for each directory and the associated files.
- ✓ The mapping from inode number to information type splits the inode number into two fields. In Linux, a PID is 16 bits wide, but an inode number is 32 bits. The top 16 bits of the inode number are interpreted as a PID, and the remaining bits define what type of information is being requested about that process.
- ✓ A PID of zero is not valid, so a zero PID field in the inode number is taken to mean that this inode contains global-rather than process-specific information.
- ✓ Not all the inode numbers in this range are reserved.
- ✓ The kernel can allocate new /procinode mappings dynamically, maintaining a bitmap of allocated in ode numbers.
- ✓ It also maintains a tree data structure of registered global /proc file-system entries.
- ✓ Each entry contains the file's inode number, file name, and access permissions, along with the special functions used to generate the file's contents.
- ✓ Drivers can register and deregister entries in this tree at any time, and a special section of the tree-appearing under the */proc/sys* directory is reserved for kernel variables.

✓ To allow efficient access to these variables from within applications, the */proc/sys* sub tree is made available through a special system call, sysctl (), that reads and writes the same variables in binary, rather than in text, without the overhead of the file system.

5.23 Input and Output

- ✓ To the user, the I/O system in Linux looks much like that in any UNIX system.
- ✓ That is, all device drivers appear as normal files.
- ✓ Users can open an access channel to a device in the same way they opens any other file-devices can appear as objects within the file system.
- ✓ The system administrator can create special files within a file system that contain references to a specific device driver, and a user opening such a file will be able to read from and write to the device referenced.
- ✓ By using the normal file-protection system, which determines who can access which file, the administrator can set access permissions for each device.



Device driver block structure

- ✓ Linux splits all devices into three classes: block devices, character devices, and network devices.
- ✓ Figure illustrates the overall structure of the device-driver system.
- ✓ **Block Devices** include all devices that allow random access to completely independent, fixed-sized blocks of data, including hard disks and floppy disks, CD-ROMs, and flash memory.
- ✓ Block devices are typically used to store file systems, but direct access to a block device is also allowed so that programs can create and repair the file system that the device contains.
- ✓ Applications can also access these block devices directly if they wish; for example, a database application may prefer to perform its own, fine-tuned laying out of data onto the disk, rather than using the general-purpose file system.
- ✓ **Character Devices** include most other devices, such as mice and keyboards.
- ✓ The fundamental difference between block and character devices is random access-block devices may be accessed randomly, while character devices are only accessed serially.
- ✓ For example, seeking to a certain position in a file might be supported for a DVD but makes no sense to a pointing device such as a mouse.
- ✓ **Network Devices** are dealt with differently from block and characterdevices.

- ✓ Users cannot directly transfer data to network devices; instead,they must communicate indirectly by opening a connection to the kernel'snetworking subsystem.

- **Block Devices**

  - ✓ Block devices provide the main interface to all disk devices in a system.
  - ✓ Performance is particularly important for disks, and the block-device system must provide functionality to ensure that disk access is as fast as possible.
  - ✓ This functionality is achieved through the scheduling of I/0 operations.
  - ✓ In the context of block devices, a block represents the unit with which the kernel performs I/0.
  - ✓ When a block is read into memory, it is stored in a buffer.
  - ✓ The request manager is the layer of software that manages the reading and writing of buffer contents to and from a block-device driver.
  - ✓ A separate list of requests is kept for each block-device driver.
  - ✓ Traditionally, these requests have been scheduled according to a unidirectional-elevator(C-SCAN) algorithm that exploits the order in which requests are inserted in and removed from the lists.
  - ✓ The request lists are maintained in sorted order of increasing starting-sector number.
  - ✓ When a request is accepted for processing by a block-device driver, it is not removed from the list.
  - ✓ It is removed only after the I/O is complete, at which point the driver continues with the next request in the list, even if new requests have been inserted into the list before the active request.
  - ✓ As new I/0 requests are made, the request manager attempts to merge requests in the lists.
  - ✓ The scheduling of I/0 operations changed somewhat with Version 2.6 of the kernel.
  - ✓ The **deadline I/O scheduler** used in Version 2.6 works similarly to the elevator algorithm except that it also associates a deadline with each request, thus addressing the starvation issue.
  - ✓ By default, the deadline for read requests is 0.5 second, and that for write requests is 5 seconds.
  - ✓ The deadline scheduler maintains a **sorted queue** of pending I/0 operations ordered by sector number.
  - ✓ However, it also maintains two other queues-a **read queue** for read operations and a **write queue** for write operations.
  - ✓ These two queues are ordered according to deadline.

- **Character Devices**

  - ✓ A character-device driver can be almost any device driver that does not offer random access to fixed blocks of data.
  - ✓ Any character-device drivers registered to the Linux kernel must also register a set of functions that implement the file I/0 operations that the driver can handle.
  - ✓ The kernel maintains a standard interface to these drivers by means of a set of tty_struct structures.

- ✓ Each of these structures provides buffering and flow control on the data stream from the terminal device and feeds those data to a line discipline.
- ✓ A **Line Discipline** is an interpreter for the information from the terminal device.
- ✓ The most common line discipline is the tty discipline, which glues the terminal's data stream onto the standard input and output streams of a user's running processes, allowing those processes to communicate directly with the user's terminal.
- ✓ This job is complicated by the fact that several such processes may be running simultaneously, and the tty line discipline is responsible for attaching and detaching the terminal's input and output from the various processes connected to it as those processes are suspended or awakened by the user.
- ✓ Other line disciplines also are implemented that have nothing to do with I/0 to a user process.
- ✓ The PPP and SUP networking protocols are ways often coding a networking connection over a terminal device such as a serial line.
- ✓ These protocols are implemented under Linux as drivers that at one end appear to the terminal system as line disciplines and at the other end appear to the networking system as network-device drivers.
- ✓ After one of these line disciplines has been enabled on a terminal device, any data appearing on that terminal will be routed directly to the appropriate network-device driver.

5.24 Interprocess Communication

- ✓ Linux provides a rich environment for processes to communicate with each other.
- ✓ Communication may be just a matter of letting another process know that some event has occurred, or it may involve transferring data from one process to another.

- **Synchronization and Signals**

  - ✓ The standard Linux mechanism for informing a process that an event has occurred is the Signals can be sent from any process to any other process, with restrictions on signals sent to processes owned by another user.
  - ✓ Signals are not generated only by processes.
  - ✓ The kernel also generates signals internally.
  - ✓ Internally, the Linux kernel does not use signals to communicate with processes ruml.ing in kernel mode.
  - ✓ If a kernel-mode process is expecting an event to occur it will not normally use signals to receive notification of that event.
  - ✓ Rather, communication about incoming asynchronous events within the kernel takes place through the use of scheduling states and wait_queue structures.
  - ✓ These mechanisms allow kernel-mode processes to inform one another about relevant events, and they also allow events to be generated by device drivers or by the networking system.
  - ✓ Whenever a process wants to wait for some event to complete, it places itself on a wait queue associated with that event and tells the scheduler that it is no longer eligible for execution.
  - ✓ Once the event has completed, it will wake up every process on the **wait Queue**.
  - ✓ This procedure allows multiple processes to wait for a single event.

- ✓ Although signals have always been the main mechanism for communicating asynchronous events among processes, Linux also implements the semaphore mechanism of System V UNIX.
- ✓ A process can wait on a semaphore as easily as it can wait for a signal, but semaphores have two advantages: Large numbers of semaphores can be shared among multiple independent processes, and operations on multiple semaphores can be performed atomically.

- **Passing of Data Among Processes**

- ✓ The standard UNIX mechanism allows a child process to inherit a communication channel from its parent; data written to one end of the pipe can be read at the other.
- ✓ Under Linux, pipes appear as just another type of inode to virtual-file system software, and each pipe has a pair of wait queues to synchronize then reader and writer.
- ✓ UNIX also defines a set of networking facilities that can send streams of data to both local and remote processes.
- ✓ Another process communications method, shared memory, offers an extremely fast way to communicate large or small amounts of data.
- ✓ Any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space.
- ✓ The main disadvantage of shared memory is that, on its own, it offers no synchronization.
- ✓ *A* process can neither ask the operating system whether a piece of shared memory has been written to nor suspend execution until such a write occurs.
- ✓ Shared memory becomes particularly powerful when used in conjunction with another interprocess-communication mechanism that provides the missing synchronization.
- ✓ *A* shared-memory region in Linux is a persistent object that can be created or deleted by processes.
- ✓ Such an object is treated as though it were a small, independent address space. The Linux paging algorithms can elect to page out to disk shared-memory pages, just as they can page out a process's data pages. The shared-memory object acts as a backing store for shared-memory regions, just as a file can act as a backing store for a memory-mapped memory region.

---o0o---