MAHENDRA

ARTS & SCIENCE COLLEGE

(AUTONOMOUS)

(NAAC ACCREDITED & AFFILIATED TO PERIYAR UNIVERSITY, SALEM)

KALIPPATTI (PO) – 637 501, NAMAKKAL (DT)

DEPARTMENT OF COMPUTER SCIENCE & APPLICATIONS

Class	:	**III B.Sc. Computer Science**
Subject	:	**OPERATING SYSTEMS**
Subject Code	:	**M19UCS10**
Semester	:	**V**
Core	:	**THEORY**

Mr. S. SRINIVASAN M.C.A., M.Phil,

SEMESTER V

Core Course – X	**B.Sc. Computer Science**	**2019 - 2020**
M19UCS10	**OPERATING SYSTEMS**	
Credit: 4		

Objectives

To provide the Fundamental Concepts of Operating System.

Course Outcomes

On the successful completion of the course, students will be able to

CO	Statement	Knowledge Level
CO1	Remember the concept of Operating Systems	K1
CO2	Understanding the Process management	K2
CO3	Applying the Process Synchronization	K3
CO4	Analyze the Memory management	K4
CO5	Apply the Storage, File Management	K3

Unit I

Introduction: What Operating System do – Computer system organization – computer system architecture – operating system operations – **Operating system structures:** Operating system services – User and operating system interface – System calls – Types of system calls – System programs.

Unit II

Process Management: Process Concepts – Process scheduling – Operations on processes – Interprocess communications- **Threads:** Overview – Multicore programming – Multithreading models – thread libraries – Implicit threading – thread issues.

Unit III

Process Synchronization: Critical section problem – synchronization hardware – semaphores – **CPU Scheduling:** Scheduling criteria – scheduling algorithms – thread scheduling – multiprocessor scheduling. Deadlock: **Deadlock** Characterization - Methods for Handling Deadlocks - Deadlock Prevention - Deadlock Avoidance - Deadlock Detection - Recovery from Deadlock.

Unit IV

Memory Management: Main memory: Swapping - Contiguous Memory Allocation – Segmentation – Paging - Structure of the Page Table. **Virtual Memory:** Demand Paging - Page Replacement - Allocation of Frames - Thrashing – Memory Mapped Files.

Unit V

Storage Management: Disk Structure - Disk Scheduling - Disk Management - Swap-Space Management - RAID Structure. **File System Interface:** File Concept- Access Methods - Directory and Disk Structure.

Text Book

S.No	Author	Title of Book	Publisher	Year of Publication
1.	Abraham Silberschatz, Peter Baer Galvin, Greg Gagne	Operating System Concepts	John Wiley & Sons, Inc.	9th Edition, 2013

Reference Book

1.	Achyut Godbole and Atul Kahate	Operating Systems	McGraw Hill Publishing	2010

Mapping with Programme Outcomes

Cos	PO1	PO2	PO3	PO4	PO5
CO1	M	S	S	S	M
CO2	S	S	M	S	S
CO3	M	S	M	M	M
CO4	S	M	S	M	S
CO5	S	M	S	S	M

S- Strong; **M**-Medium

UNIT I

INTRODUCTION

An operating system is a program that manages a computer's hardware. It acts as an intermediary between a user of a computer and the computer hardware. Mainframe operating systems-Optimize utilization of hardware. Personal computer (PC) operating systems-support complex games, business applications, and everything in between.

Operating systems for mobile computers provide an environment in which a user can easily interface with the computer to execute programs.

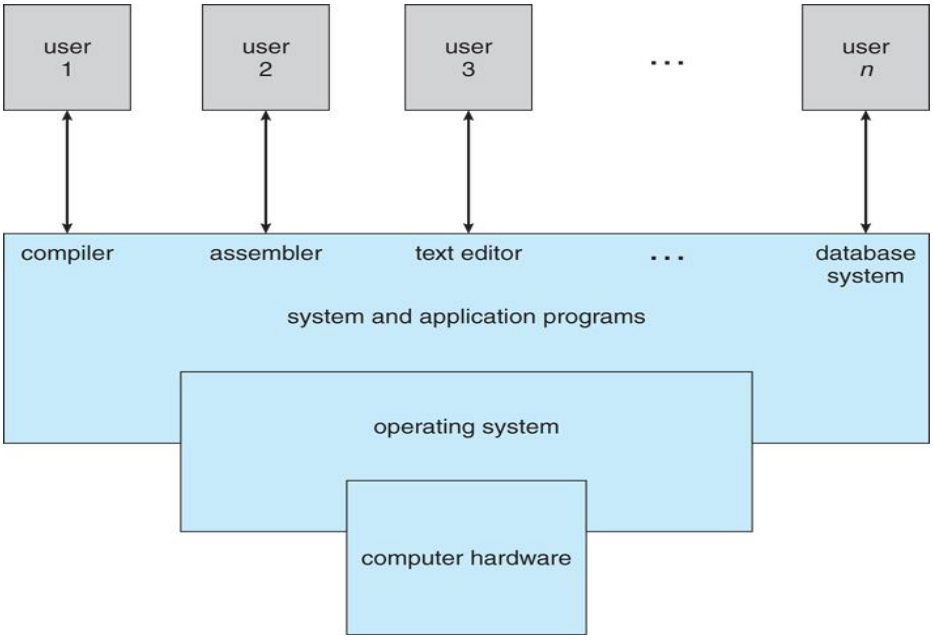
Operating system goals:

1. Execute user programs and make solving user problems easier
2. Make the computer system convenient to use
3. Use the computer hardware in an efficient manner

Computer System Structure:

Computer system can be divided into four components:

1. Hardware – provides basic computing resources CPU, memory, I/O devices
2. Operating system
   Controls and coordinates use of hardware among various applications and users
3. Application programs – define the ways in which the system resources are used to solve the computing problems of the users Word processors, compilers, web browsers, database systems, video games
4. Users People, machines, other computers



WHAT OPERATING SYSTEM DO:

Depends on the point of view. Operating System is divided into

- ➢ User view
- ➢ System view

- **Users view** want convenience, ease of use and good performance
- Don’t care about resource utilization.
- But shared computer such as mainframe or minicomputer must keep all users happy.
- Users of dedicate systems such as workstations have dedicated resources but frequently use shared resources from servers. Ex: Printer
- Handheld computers are resource poor, optimized for usability and battery life.
- Some computers have little or no user interface, such as embedded computers in devices and automobiles.

System view

- OS is a resource allocator

  Manages all resources (CPU time, memory space, I/O devices, etc...) Decides between conflicting requests for efficient and fair resource use.
- OS is a control program

  Controls execution of programs to prevent errors and improper use of the computer.

Operating System Definition

- “The one program running at all times on the computer” is the **kernel**.
- Everything else is either
  1. a system program (ships with the operating system)
  2. An application program-which include all programs not associated with the operation of the system.
- Mobile operating systems: often include not only a core kernel but also **middleware**—a set of software frameworks that provide additional services to application developers.
- For example, each of the two most prominent mobile operating systems, Apple’s ios and Google’s Android—features a core kernel along with middleware that supports databases, multimedia, and graphics.

COMPUTER SYSTEM ORGANIZATION

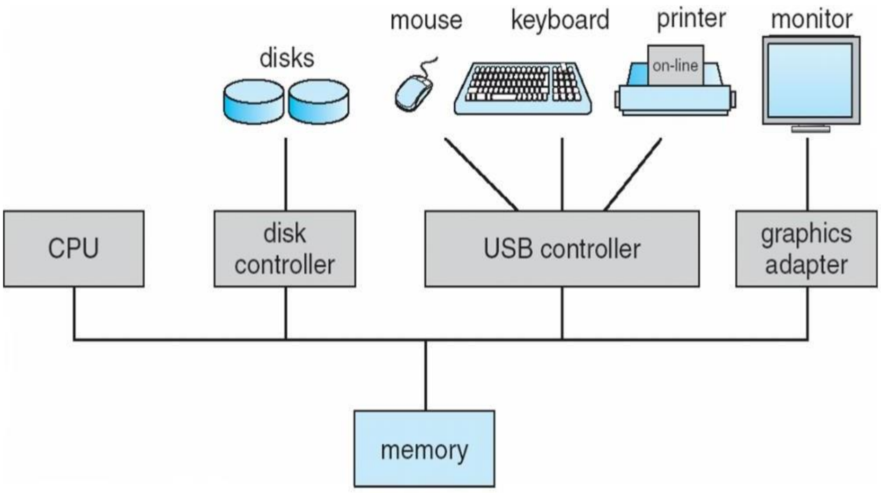
- Computer System Operation
- Storage Structure
- I/O Structure

Computer System Operation

For a computer to start running, bootstrap program is loaded at power-up or reboot. Typically stored in ROM or EPROM, generally known as firmware. Initializes all aspects of system. Loads operating system kernel and starts execution.

One or more CPUs, device controllers connect through common bus providing access to shared memory. Concurrent execution of CPUs and devices competing for memory cycles.

- I/O devices and the CPU can execute concurrently



- Each device controller is in charge of a particular device type.
- Each device controller has a local buffer.
- CPU moves data from/to main memory to/from local buffers.
- I/O is from the device to local buffer of controller.
- Device controller informs CPU that it has finished its operation by causing an interrupt.

Storage Device:

- Main memory – only large storage media that the CPU can access directly
  - **Random access memory (RAM)**
  - Typically, **volatile**
- Secondary storage – extension of main memory that provides large **nonvolatile** Storage capacity.

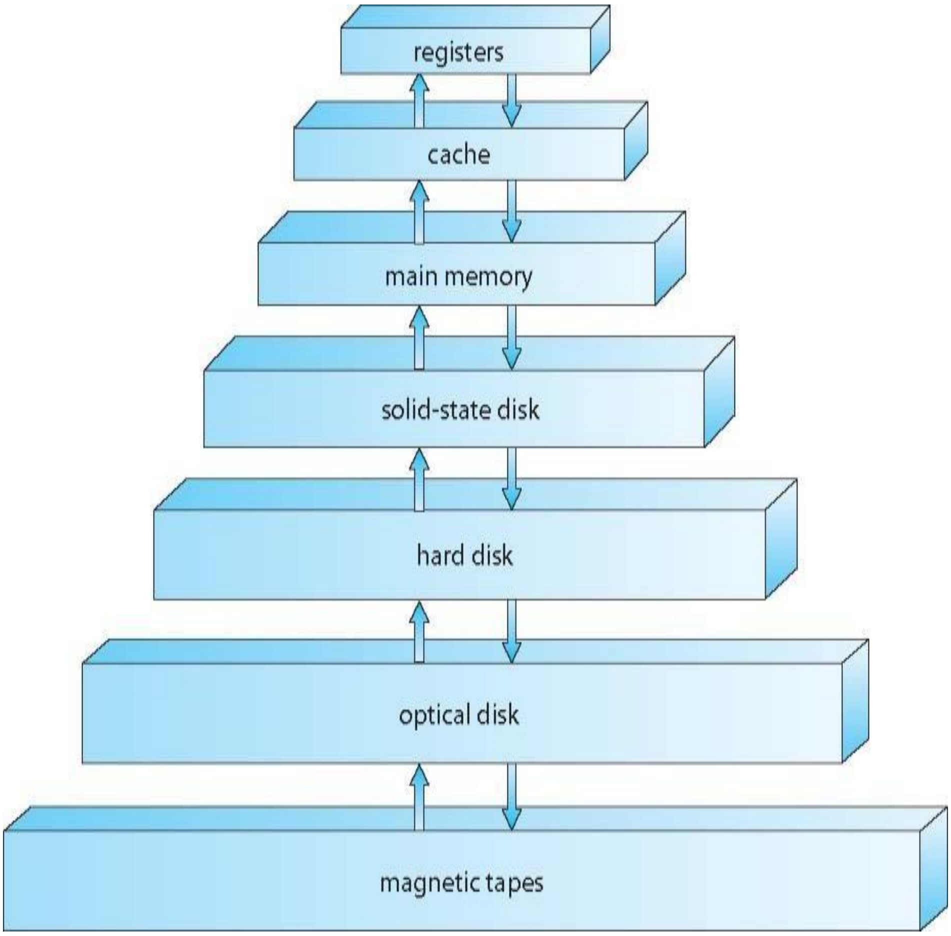
  Hard disks – rigid metal or glass platters covered with magnetic recording material.

- Disk surface is logically divided into **tracks**, which are subdivided into **Sectors.**
- The **disk controller** determines the logical interaction between the device and the computer.

- **Solid-state disks** – faster than hard disks, nonvolatile
  - Various technologies
  - Becoming more popular

Storage Hierarchy

- Storage systems organized in hierarchy Speed
  Cost Volatility
- Caching – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- Device Driver for each device controller to manage I/O
- Provides uniform interface between controller and kernel



I/O Structure:

- General-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus.
- Each device controller is in charge of a specific type of device. Depending on the controller, more than one device may be attached. For instance, seven or more devices can be attached to the small computer-systems interface (SCSI) Controller.
- A device controller maintains some local buffer storage and a set of special- purpose registers.
- The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.
- Operating systems have a device driver for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.

COMPUTER SYSTEM ARCHITECTURE

- A computer system can be organized in a number of different ways, according to the number of general-purpose processors used
  - ✓ Single-Processor systems
  - ✓ Multiple-Processor systems
  - ✓ Clustered systems

Single processor system

- On a single processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes.

Multiprocessor systems

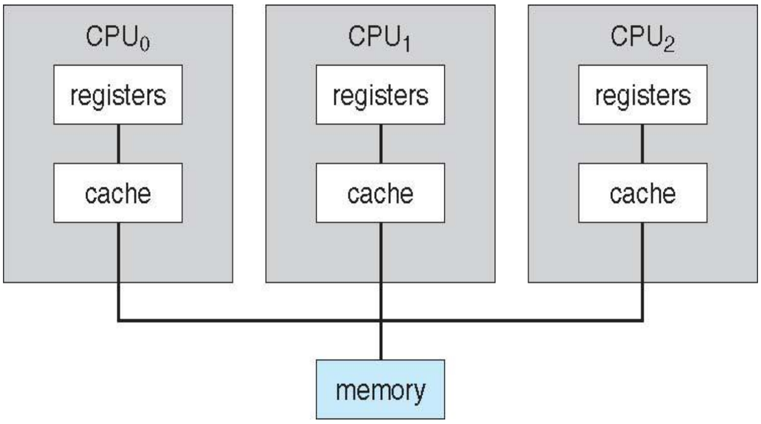
- Also known as parallel systems or Multicore systems.
- Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.
- Multiprocessor systems appeared in server’s desktop and laptop systems. Recently, multiple processors have appeared on mobile devices such as smartphones and tablet computers.
- Multiprocessor systems have three main advantages,
  - ▪ Increased throughput
  - ▪ Economy of scale
  - ▪ Increased reliability
- Multiprocessor systems have three main advantages:

- ✓ Increased throughput: By increasing the number of processors, we expect to get more work done in less time.
- ✓ Economy of scale: Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.
- ✓ Increased reliability: If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

- Increased reliability of a computer system is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called graceful degradation.
- Some systems go beyond graceful degradation and are called fault tolerant, because they can suffer a failure of any single component and still continue operation and produce the result.
- The multiple-processor systems are further divided into two types.
    1. Symmetric multiprocessing
    2. Asymmetric multiprocessing

1. Symmetric multiprocessing (SMP)

- In which each processor performs all tasks within the operating system. SMP means that all processors are peers; no boss–worker relationship exists between processors.

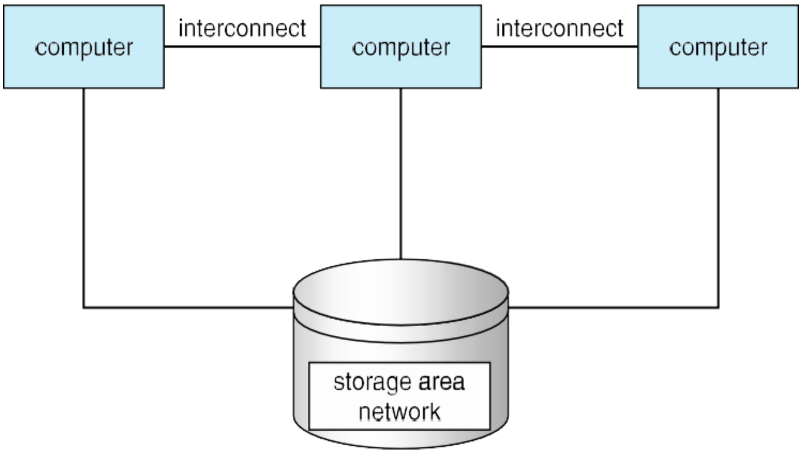


2. Asymmetric multiprocessing

- In which each processor is assigned a specific task.
- A boss processor controls the system; the other processors either look to the boss for instruction or have predefined tasks. This scheme defines a boss– worker relationship. The boss processor schedules and allocates work to the worker processors.

Clustered systems

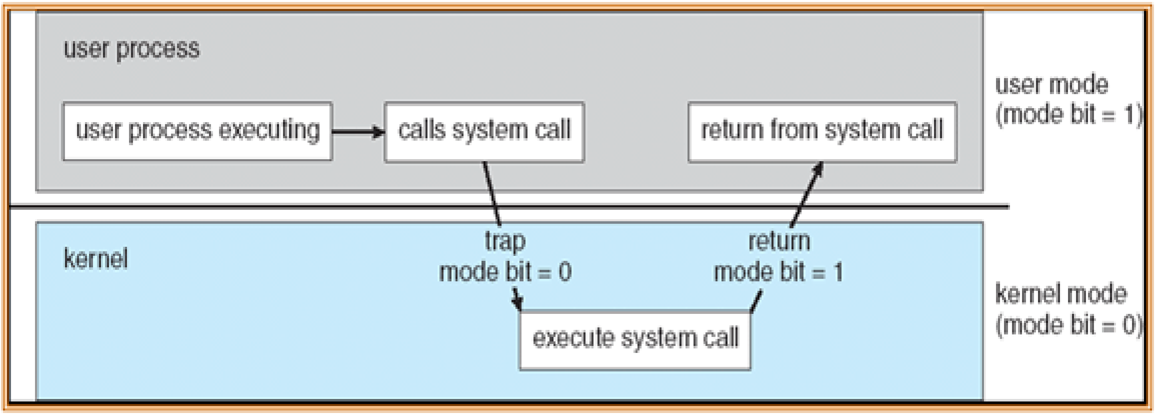
- It differs from the multiprocessor systems in that they are composed of two or more individual systems or nodes joined together. Such systems are considered loosely coupled.
- Each node may be a single processor system or a Multicore system. Clustered computers share storage and are closely linked via a local-area network LAN.
- Ex: systems used in schools and colleges.
- Clustering can be classified in to
  1. Asymmetrically
  2. Symmetrically.
- In **asymmetric clustering,** one machine is in hot-standby mode while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server.
- In **symmetric clustering**, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware.
- Storage-area networks (SANs), which allow many systems to attach to a pool of storage.



OPERATING SYSTEM OPERATIONS:

- Modern operating systems are interrupting driven. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen.
- Events are almost always signaled by the occurrence of an interrupt or a trap.
- A **trap (or an exception**) is a software-generated interrupt caused either by an error or by a specific request from a user program that an operating-system service be performed.

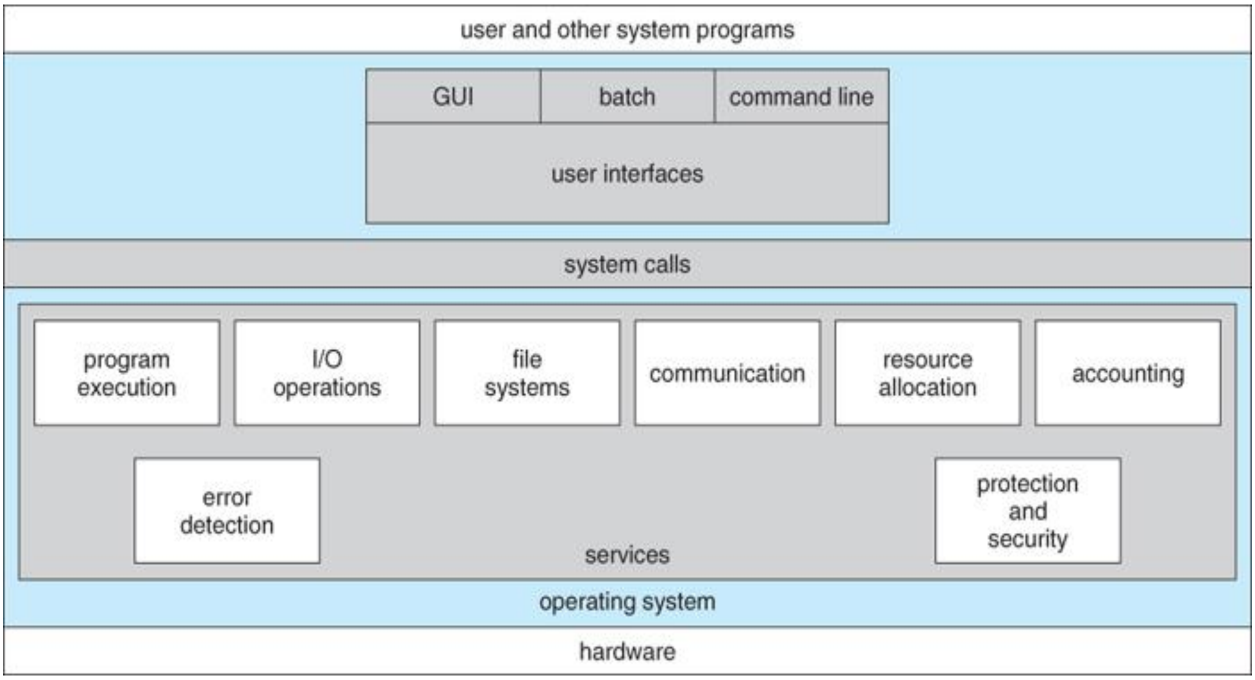
- For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided to deal with the interrupt.
- The OS has two separate modes of operation:
  - ✓ **User mode**
  - ✓ **Kernel mode** (Supervisor mode, System mode, Privileged mode)
- A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: **kernel (0) or user (1).**
- With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user.
- When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request.
- At system boot time, the hardware starts in kernel mode. The operating system 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 (that is, changes the state of the mode bit to 0).



OPERATING SYSTEM STRUCTURES:

- Operating system structures describe the **services an operating system** provides to users, processes, and other systems.
- User and operating system interfaces
- System calls
- Types of system calls
- System Programs

OPERATING SYSTEM SERVICES



The operating system provides certain services to programs and to the users of those programs. The specific services provided, of course, differ from one operating system to another,

1. **<u>User interface</u>:** Almost all operating systems have a user interface (UI). This interface can take several forms. One is a command-line interface (CLI), which uses text commands and a method for entering them. Another method is a graphical user interface (GUI), the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text.
2. **<u>Program execution</u>:** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
3. **<u>I/O operations</u>:** A running program may require I/O, which may involve a file or an I/O device.
4. **<u>File-system manipulation</u>:** Programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information.
5. **<u>Communications:</u>** There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network.

6. **Error detection:** The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware, in I/O devices, and in the user program.

   For each type of errors, the operating system should take the appropriate action like,

   - ✓ Halt the system
   - ✓ Terminate an error-causing process
   - ✓ To detect and possibly correct.

The following operating system functions helps in ensuring the efficient operation of the system.

1. **Resource allocation:** When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources, such as CPU cycles, main memory, and file storage.
2. **Accounting:** We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics.
3. **Protection and security:** Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate him or her to the system, usually by means of a password, to gain access to system resources

USER AND OPERATING SYSTEM INTERFACE

- There are several ways for users to interface with the operating system. Here, we discuss two fundamental approaches.
  - ✓ One provides a command-line interface, or command interpreter, that allows users to directly enter commands to be performed by the Operating system.
  - ✓ Interface with the operating system via a graphical user interface, or GUI.

Command Interpreter

- Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on.
- Some operating systems include the command interpreter in the kernel.
- Operating systems with multiple command interpreters to choose from, the interpreters are known as shells.

✓ UNIX and Linux systems, has different shells,

- Bourne shell
- C shell
- Bourne-Again shell
- Korn shell
- user-written shells

- All the shells have similar functionality, and a user’s choice which shell to use is generally based on personal preference.
- Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.
- The main function of the command interpreter is to get and execute the next user specified command.
- Many of the commands given at this level manipulate files:

Create, delete, list, print, copy, execute, and so on.

Ex: MS-DOS and UNIX shells operate in this way.

- These commands can be implemented in two general ways.

1. Command interpreter itself contains the code to execute the command.

Ex: a command to delete a file.

2. Commands through system programs. In this case, the command interpreter does not understand the command in anyway; it merely uses the command to identify a file to be loaded into memory and executed.

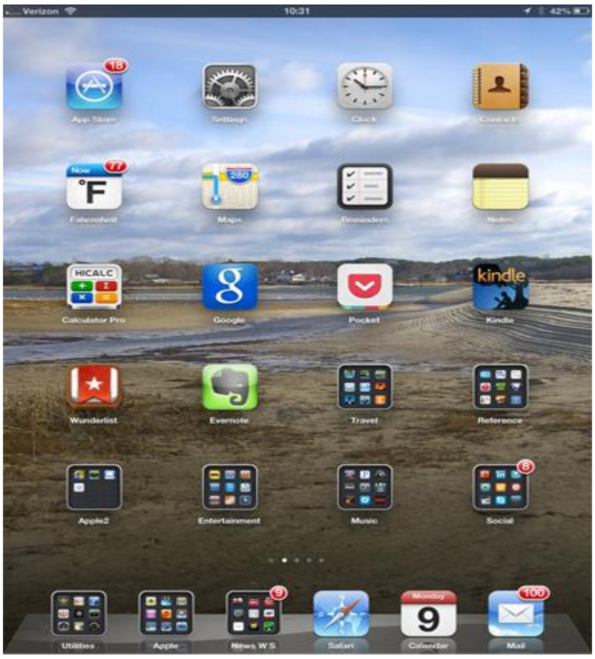
Ex: rm file.txt

- Search for the file and load it in the memory

Graphical User Interface:

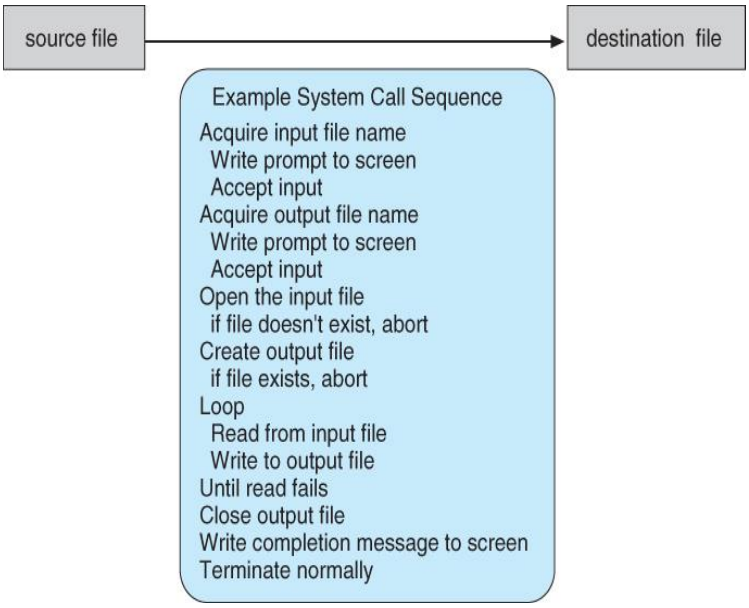
- A second strategy for interfacing with the operating system is through a user- friendly graphical user interface, or GUI.
- Here, users employ a mouse-based window and menu system characterized by a desktop metaphor. The user moves the mouse to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories, and system functions.
- Clicking a button on the mouse can invoke a program, select a file or directory known as a folder or pull down a menu that contains commands.
- Graphical user interfaces first appeared in the early 1970s at Xerox PARC research facility. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s.
- Smart phones and handheld tablet computers typically use a touch screen interface.

Users interact by making gestures on the touch screen—for example, pressing and swiping fingers across the screen.



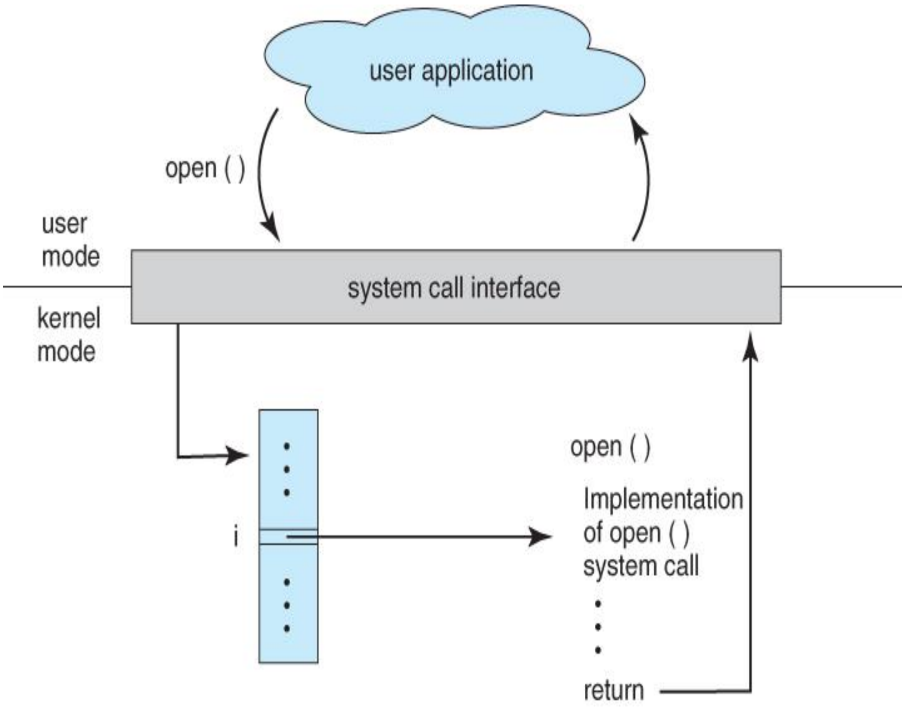
SYSTEM CALLS

- System calls provide an interface to the services made available by an operating System.
- These calls are generally available as routines written in C and C++. Low level task is written using assembly-language instructions.
- Operating systems execute thousands of systems calls per second.
- Ex: System call sequence to copy the contents of one file to another file



```
Example System Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

- Mostly system calls are accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call.
- Three most common APIs are
  1. Win32 API - Windows
  2. POSIX API - POSIX-based systems (UNIX, Linux, MacOS X)
  3. Java API - the Java virtual machine (JVM)
- A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called libc.
- Typically, a number associated with each system call.
- System-call interface maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
- Just needs to obey API and understand what OS will do as a result call
- Most details of OS interface hidden from programmer by API
- Managed by run-time support library (set of functions built into libraries included with compiler).



API-System Call-OS Relationship

TYPES OF SYSTEM CALLS

- System calls can be grouped roughly into six major categories:
    1. Process control
    2. File manipulation
    3. Device manipulation
    4. Information maintenance
    5. Communications
    6. Protection

1. Process Control:

- ✓ A running program needs to be able to halt its execution either normally (end ()) or abnormally (abort ()).
- ✓ To ensure the integrity of the data being shared, operating systems often provide system calls allowing a process to lock shared data.

Process control

- ✓ create process, terminate process
- ✓ end, abort
- ✓ load, execute
- ✓ get process attributes, set process attributes
- ✓ wait for time
- ✓ wait event, signal event
- ✓ allocate and free memory
- ✓ Dump memory if error
- ✓ **Debugger** for determining **bugs, single step** execution
- ✓ **Locks** for managing access to shared data between processes

2. File management

- ✓ create file, delete file
- ✓ open, close file
- ✓ read, write, reposition(rewind/skip)
- ✓ get and set file attributes

3. Device management

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

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

5. **Communications**
    - ✓ create, delete communication connection
    - ✓ send, receive messages if **message passing model** to **host name** or **Process name**
        - ▪ From **client** to **server**
    - ✓ **Shared-memory model** create and gain access to memory regions
    - ✓ transfer status information
    - ✓ attach and detach remote devices

6. **Protection**
    - ✓ Control access to resources
    - ✓ Get and set permissions
    - ✓ Allow and deny user access

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. System program are divided into the following categories,
    1. File Management
    2. Status information
    3. File modification
    4. Programming language support
    5. Program loading and execution
    6. Communication
    7. Background services

File management:

✓ These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

Status information:

✓ Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Some systems also support a registry, which is used to store and retrieve configuration information.

File modification:

- ✓ Several text editors may be 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.

Programming-language support:

- ✓ Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system.

Program loading and execution:

- ✓ Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide
  - ▪ Absolute loaders
  - ▪ Relocatable loaders
  - ▪ Linkage editors
  - ▪ Overlay loaders.

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 screens, to browse Web pages.

Background services:

- ✓ All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted.

Ex:

A system needed a service to listen for network connections.

Process schedulers that start processes according to a specified schedule System error monitoring services, and print servers

----

UNIT II

PROCESS MANAGEMENT:

- A process is a program in execution. A process will need certain resources like CPU time, memory, files, and I/O devices to accomplish its task.
- A process is the unit of work. Systems consist of a collection of processes:
    - ✓ operating-system processes execute system code
    - ✓ User processes execute user code.
- All these processes may execute concurrently.
- The operating system is responsible for several important aspects of process management:
    - ✓ the creation and deletion of both user and system processes
    - ✓ the scheduling of processes
    - ✓ The provision of mechanisms for synchronization, communication, and deadlock handling for processes.

PROCESS CONCEPTS

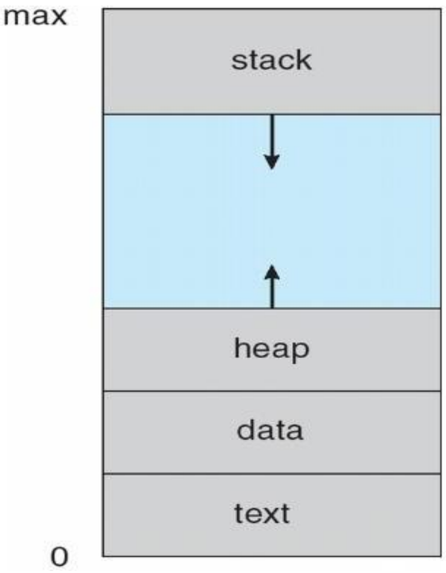
- User may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package.
- On an embedded device that does not support multitasking; the operating system may need to support its own internal programmed activities, such as memory management.
- All these activities are called as processes.
- The terms *job* and *process* are used interchangeably.
- The process concept involves,
    1. The process
    2. Process state
    3. Process control block
    4. Threads

The Process

- A program is a *passive* entity, such as a file containing a list of instructions stored on disk.
- In contrast, a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.
- Two common techniques available for loading executable files,
  - ✓ Double-clicking an icon representing the executable file.
  - ✓ Entering the name of the executable file on the command line.

Structure of Process in memory

- A process is more than the program code, which is sometimes known as the text section.
- It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers.
- A process also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables),
- D**ata 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 below.



Process State

- As a process executes, it changes state.
- The state of a process is defined by the current activity of that process. A process may be in one of the following states:

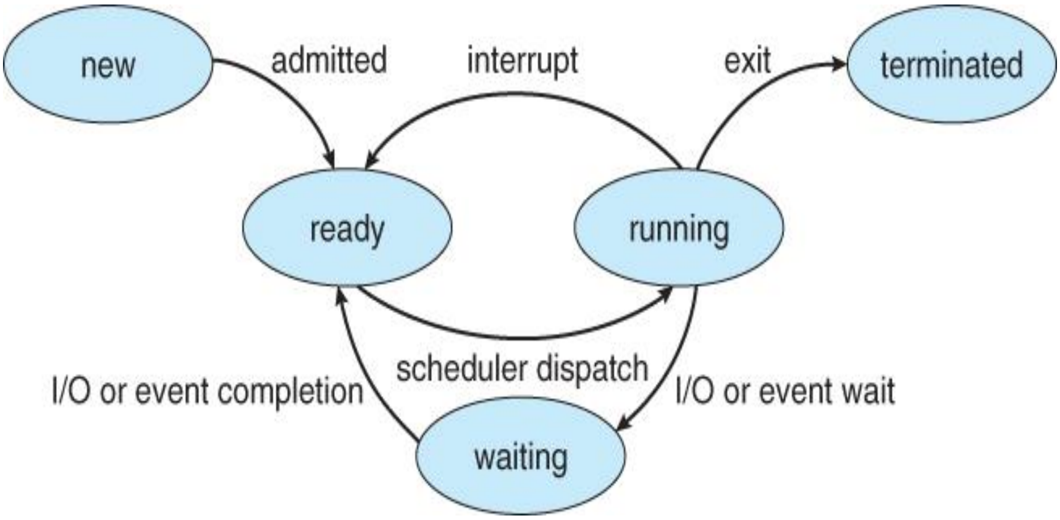
  1. New. The process is being created.

  2. Running. Instructions are being executed.

  3. Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

  4. Ready. The process is waiting to be assigned to a processor.

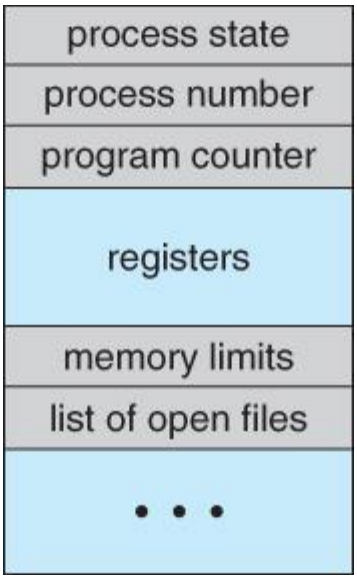
  5. Terminated. The process has finished execution.
- Only one process can be *running* on any processor at a time. Many processes may be in *ready* and *waiting state.*



Process Control Block

- Each process is represented in the operating system by a process control block (PCB) also called a task control block.
- It contains many pieces of information associated with a specific process, including the following:

  1. Process state
  2. Program counter
  3. CPU registers
  4. CPU-scheduling information
  5. Memory-management information
  6. Accounting information
  7. I/O status information.



Process state: The state may be new, ready, running, and waiting, halted, and so on.

Program counter: The counter indicates the address of the next instruction to be executed for this process.

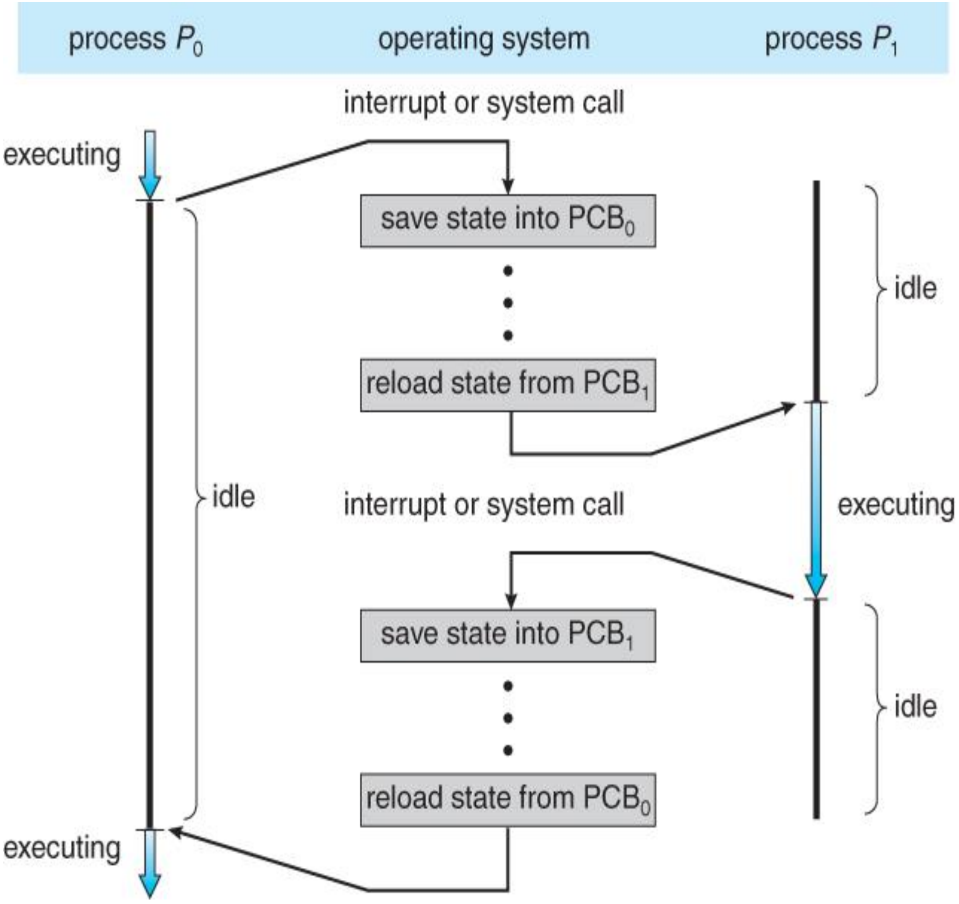
CPU registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and other scheduling parameters.

Memory-management information: This includes the page tables, or the segment tables, depending on the memory system used by the operating system.

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 Switching from Process to Process

Threads

- The process is a program that performs a single thread of execution.
- Ex: when a process is running a word-processor program, a single thread of instructions is being executed.
- This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process.
- Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution.
- Perform more than one task at a time i.e., where multiple threads can run in parallel.

PROCESS SCHEDULING

- Multiprogramming is to have some process running at all times, to maximize CPU utilization.

- Time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

- To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU.

- For a single-processor system, has only one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

- The process scheduling involves;

  - ✓ Scheduling Queues
  - ✓ Schedulers
  - ✓ Context Switch

Scheduling Queues:

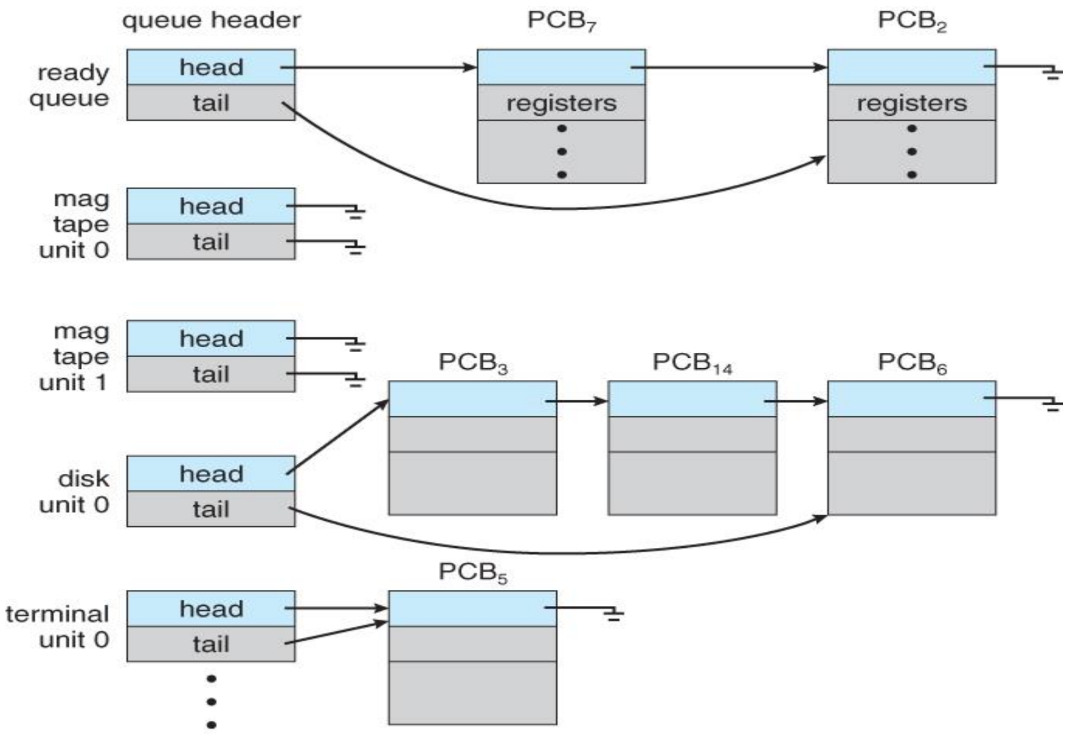
- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.

- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.

- This queue is generally stored as a linked list.

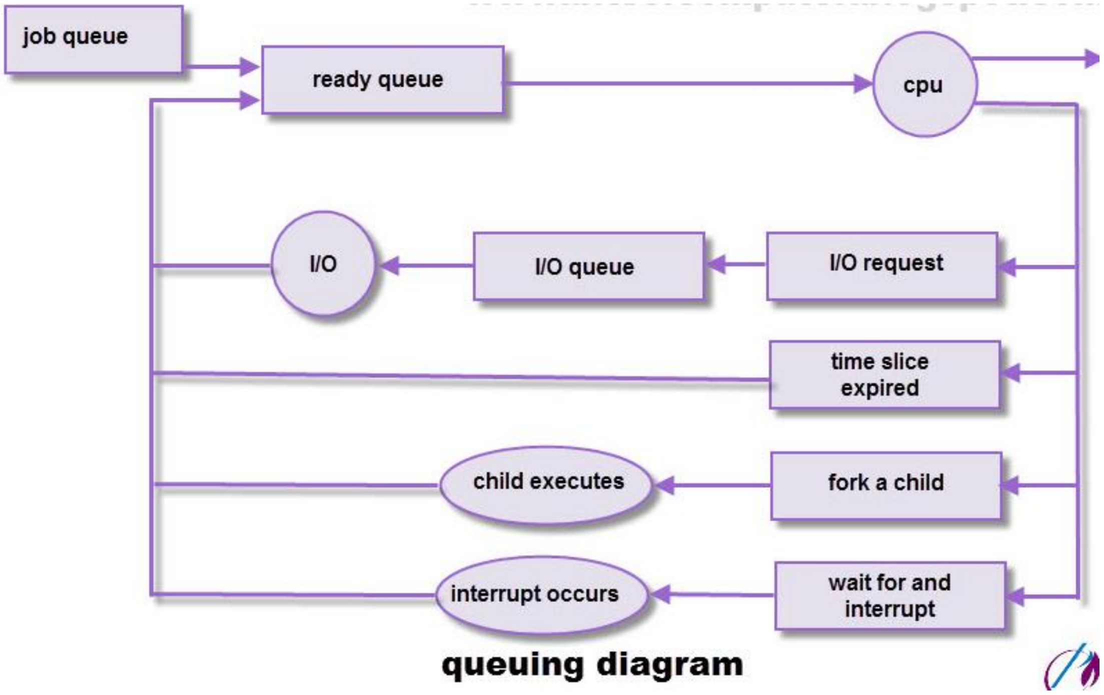
- A ready-queue header contains pointers to the first and final PCBs in the list.

- Each PCB includes a pointer field that points to the next PCB in the ready queue.



Ready Queue and various I/O Device Queue

- When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.
- Suppose the process makes an I/O request to a shared device, such as a disk.
- Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk.
- The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.
- A common representation of process scheduling is a queuing diagram.



Queuing-Diagram Representation of Process Scheduling

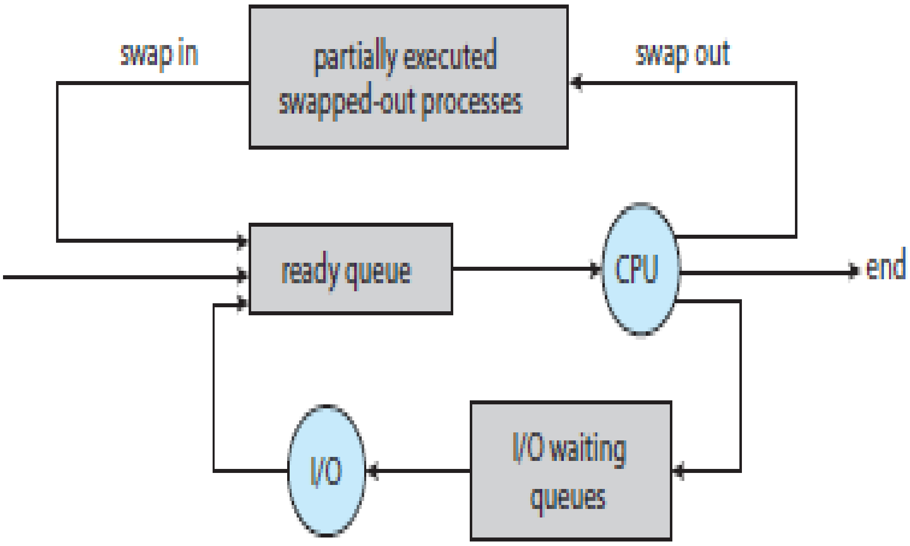
- Each rectangular box represents a queue.
- Two types of queues are present:
  - ✓ Ready queue
  - ✓ Device queues
- The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system
- A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**.
- Once the process is allocated the CPU and is executing, one of several events could occur:

- ✓ The process could issue an I/O request and then be placed in an I/O queue.
  - ✓ The process could create a new child process and wait for the child's termination.
  - ✓ The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.
- In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources reallocated.

Schedulers:

- A process migrates among the various scheduling queues throughout its lifetime.
- The operating system must select the processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler.**
- **Batch system:** more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device where they are kept for later execution.
- The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution.
- The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory).
- If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- Most processes can be described as either I/O bound or CPU bound.
  - ✓ An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations.
  - ✓ A **CPU-bound process** uses most of its time doing computations.
- If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do.
- If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.
- The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.

- Time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.
- Time-sharing systems introduces an additional, intermediate level of scheduling called medium-term scheduler.
- The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory and thus reduce the degree of multiprogramming.
- The process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.
- The process is swapped out, and is later swapped in, by the medium-term scheduler.
- Swapping may be necessary to improve the process mix, because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.



Addition of medium-term scheduling to the queuing diagram

<u>Context Switch:</u>

- When an interrupt occurs, the system needs to save the current context of the process running on the CPU.
- So that it can restore that context when it’s processing is done, essentially suspending the process and then resuming it.
- The context is represented in the PCB of the process. It includes

    - ✓ Value of the CPU registers,
    - ✓ Process state
    - ✓ Memory-management information

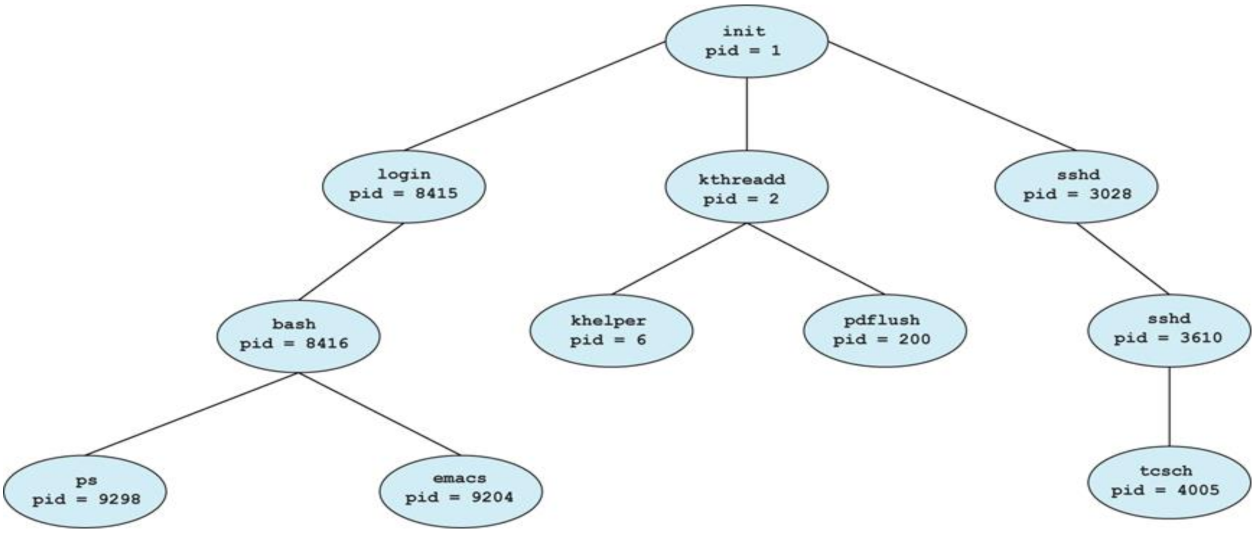
- State save of the current state of the CPU, be it in kernel or user mode, and then a state restores to resume operations
- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Switching speed varies from machine to machine, depending on the
    - ✓ Memory speed,
    - ✓ Number of registers that must be copied
    - ✓ Existence of special instructions

OPERATIONS ON PROCESSES

- The processes in most systems can execute concurrently.
- The process can be created and deleted dynamically.

Process Creation:

- During the course of execution, a process may create several new processes.
- The creating process is called a parent process, and the new processes are called the children of that process.
- Each of these new processes may in turn create other processes, forming a **tree** of processes.
- Most operating systems identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number.
- The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.



<u>Tree of Process on a LINUX System</u>

- The node in a process tree has name of each process and its pid.
- The init process has a pid of 1, serves as the root parent process for all user processes.
- Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server, and the like.
- The three children of init—login, kthreadd and sshd.
- The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel
- The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for ***secure shell***).
- The login process is responsible for managing clients that directly log onto the system.
- In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416.
- Using the bash command-line interface, this user has created the process ps as well as the emacs editor.
- On UNIX and Linux systems, we can obtain a listing of processes by using the ps command. For example, the command **<u>ps -el</u>** will list complete information for all processes currently active in the system.
- When a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.

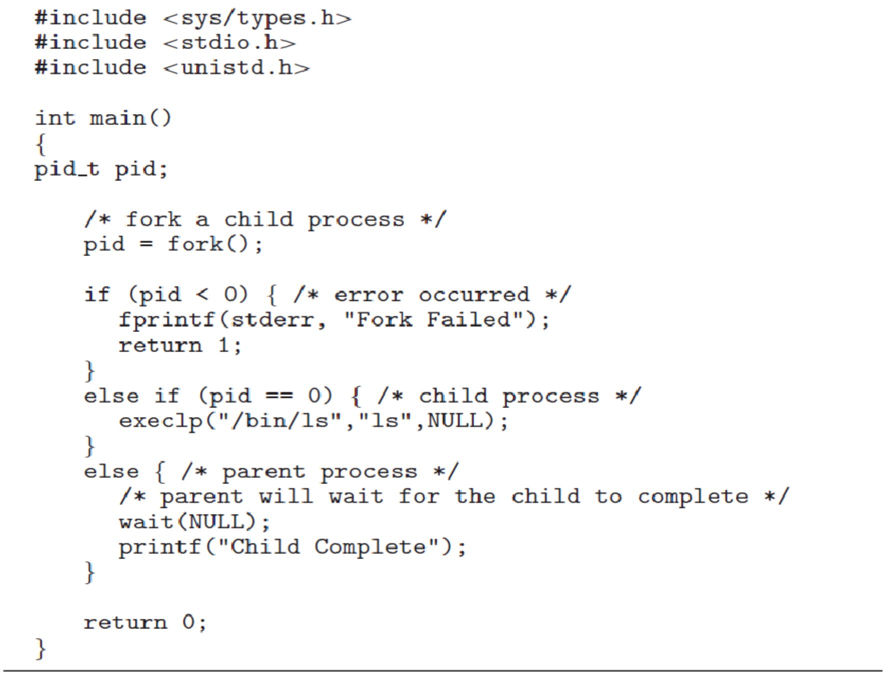
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.
- In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process.
- Ex: consider a process whose function is to display the contents of a file say, image.jpg on the screen of a terminal.
- When the process is created, it will get, as an input from its parent process, the name of the file ***image.jpg***.
- Using that file name, it will open the file and write the contents out. It may also getthe name of the output device.
- Alternatively, some operating systems pass resources to child processes.
- On such a system, the new process may get two open files, image.jpg and the terminal device, and may simply transfer the datum between the two.
- When a process creates a new process, two possibilities for execution exist:
    - ✓ The parent continues to execute concurrently with its children.
    - ✓ The parent waits until some or all of its children have terminated.
- There are also two address-space possibilities for the new process:
    - ✓ The child process is a duplicate of the parent process (it has the same program and data as the parent).
    - ✓ The child process has a new program loaded into it.

UNIX Operating System:

- In UNIX, each process is identified by its process identifier, which is a unique integer.
- A new process is created by the fork () system call.
- The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.
- Both processes (the parent and the child) continue execution at the instruction afterthe fork(), with one difference:
    - ✓ The return code for the fork () is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
- One of the two processes typically uses the exec () system call to replace the process's memory space with a new program.
- The exec () system call loads a binary file into memory and starts its execution.
- In this manner, the two processes are able to communicate and then go their separate ways.
- The parent can then create more children; or, if it has nothing else to do while the child runs,

it can issue a wait () system call to move itself off the ready queue until the termination of the child.

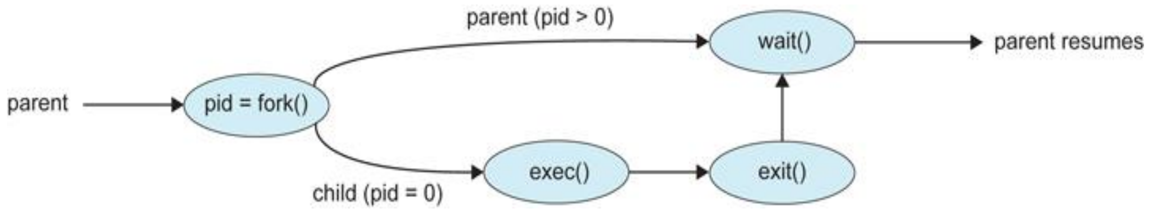
- Because the call to exec () overlays the process's address space with a new program, the call to exec () does not return control unless an error occurs.
- The value of pid for the child process is zero, while that for the parent is an integer value greater than zero.



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

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

    return 0;
}
```

Creating a separate process using the UNIX fork () system call

- The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files.
- The parent waits for the child process to complete with the wait () system call.
- When the child process completes by either implicitly or explicitly invoking exit (), the parent process resumes from the call to wait(), where it completes using the exit() system call.



Process creation using the fork () system call

<u>Process Termination:</u>

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call.
- At that point, the process may return a status value (typically an integer) to its parent

process.

- All the resources of the process including physical and virtual memory, open files, and I/O buffers are reallocated by the operating system.
- A process can cause the termination of another process via an appropriate system call (Ex: Terminate Process () in Windows).
- Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.
- Parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
    - ✓ The child has exceeded its usage of some of the resources that it has been allocated.
    - ✓ The task assigned to the child is no longer required.
    - ✓ The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Cascading termination

- Some systems do not allow a child to exist if its parent has terminated.
- In such systems, if a process terminates then all its children must also be terminated. This phenomenon, referred to as **cascading termination**.
- It is normally initiated by the operating system.

/* exit with status 1 */ exit(1);

Child Process Termination:

- A parent process may wait for the termination of a child process by using the wait() system call.
- The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child.
- This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid t pid; int status;
pid = wait(&status);
```

- A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie** process.

- Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.

Orphan Process:

- If a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans**.
- The Operating System assigning the init process as the new parent to orphan processes.
- The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan’s process identifier and process-table entry.

INTERPROCESS COMMUNICATIONS

- Processes executing concurrently in the operating system may be either
  - ✓ Independent processes
  - ✓ 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.
- There are several reasons for providing an environment that allows process cooperation:
  1. Information sharing
  2. Computation speedup
  3. Modularity
  4. Convenience

Information sharing:

- Since several users may be interested in the same piece of information, we must provide an environment to allow concurrent access to such information.

Computation speedup:

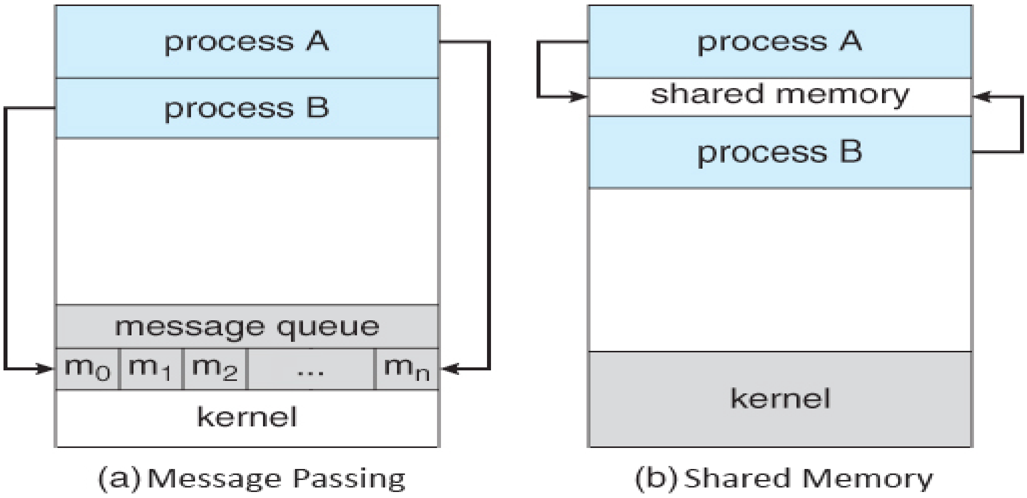
- If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

Modularity:

- We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

Convenience:

- Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.



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

- Cooperating processes require an **inter-process communication (IPC)** mechanism that will allow them to exchange data and information.
- There are two fundamental models of inter-process communication:
  - ✓ **Shared memory**
  - ✓ **Message passing**.
- In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.
- Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
- Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
- In shared-memory systems, system calls are required only to establish shared memory regions.
- Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Shared-Memory Systems

- Inter-process communication using shared memory requires communicating processes to establish a region of shared memory.
- A shared-memory region resides in the address space of the process creating the shared-memory segment.
- Two or more processes can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- For Ex; let's consider the producer–consumer problem, which is a common paradigm for cooperating processes.
- A **producer** process produces information that is consumed by a **consumer** process.
- Ex: compiler may produce assembly code that is consumed by an assembler.
- The assembler, in turn, may produce object modules that are consumed by the loader.
- Server as a producer and a client as a consumer.
- For Ex; a web server produces HTML files and images, which are consumed by the client web browser requesting the resource.
- One solution to the producer–consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes. A 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.

- Two types of buffers can be used.
  - ✓ **Unbounded buffer**
  - ✓ **bounded buffer**

- The **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.
- The **bounded buffer** assumes a fixed buffer size. In this case, the 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 typedef struct {
. . .
}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
    - ✓ 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.

Message-Passing Systems

- Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.
- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- Ex: an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.
- A message-passing facility provides at least two operations:
    - ✓ send(message)
    - ✓ receive(message)
- Messages sent by a process can be either fixed or variable in size.
- If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction makes the task of programming more difficult.
- Variable-sized messages require a more complex system level implementation, but the programming task becomes simpler.
- If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other: a *communication link* must exist between them. This link can be implemented in a variety of ways.
- We are concerned here not with the link's physical implementation but rather with its logical implementation.
- Here are several methods for logically implementing a link and the send()/receive() operations:

Naming

- ✓ Direct or indirect communication: Naming
- ✓ Synchronous or asynchronous communication
- ✓ Automatic or explicit buffering
  - 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**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, 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
  - A communication link in this scheme has the following properties:
    - ✓ A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
    - ✓ A link is associated with exactly two processes.
    - ✓ Between each pair of processes, there exists exactly one link.

Addressing:

- ***Symmetry*** in addressing, both the sender process and the receiver process must name the other to communicate.
- A***symmetry*** addressing, here, only the sender names the recipient; the recipient is not required to name the sender.
- In this scheme, the send() and receive() primitives are defined as follows:
  - ✓ 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 this scheme, a communication link has the following properties:
  - ✓ A link is established between a pair of processes only if both members of the pair have a shared mailbox.
  - ✓ A link may be associated with more than two processes.
  - ✓ Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Mailbox

- A mailbox may be owned either by a process or by the operating system.
- If the mailbox is owned by a process then we distinguish between the owners the user.
- Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox.
- When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.
- A mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process.
- The operating system then must provide a mechanism that allows a process to do the following:
  - ✓ Create a new mailbox.
  - ✓ Send and receive messages through the mailbox.
  - ✓ Delete a mailbox.
- The process that creates a new mailbox is that mailbox's owner by default.
- Initially, the owner is the only process that can receive messages through this mailbox.
- However, the ownership and receiving privilege may be passed to other processes through appropriate system calls.

Synchronization

- Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive.
- Message passing may be either **blocking** or **nonblocking** also known as **synchronous** and **asynchronous**.
  - ✓ **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
  - ✓ **Nonblocking send**. The sending process sends the message and resumes operation.
  - ✓ **Blocking receive**. The receiver blocks until a message is available.
  - ✓ **Nonblocking receive**. The receiver retrieves either a valid message or a null

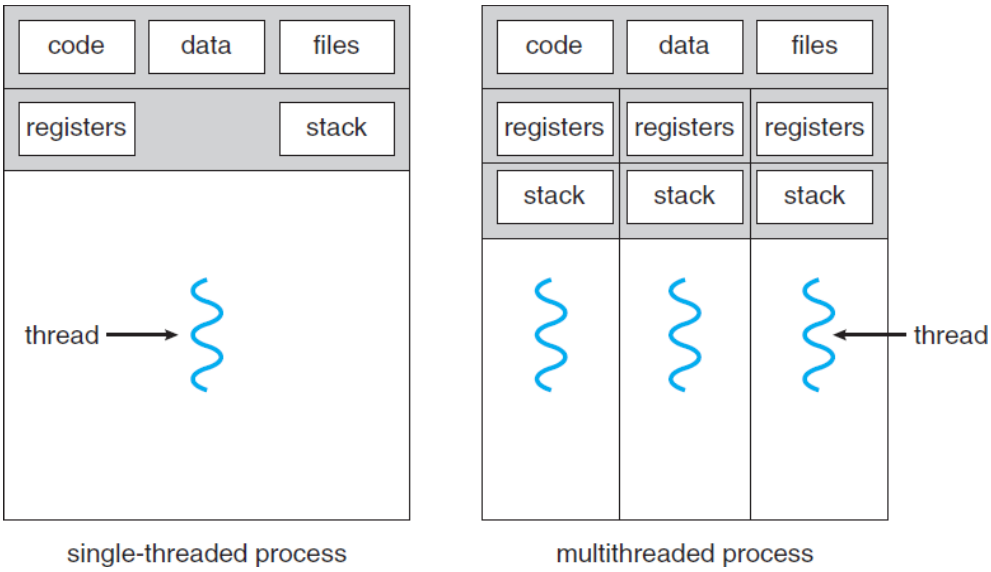
Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.

- Basically, such queues can be implemented in three ways:
- **Zero capacity**. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**. The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity**. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.
- Zero-capacity - Message system with no buffering.
- Bounded/Unbounded capacity - Systems with automatic buffering.

THREADS: OVERVIEW

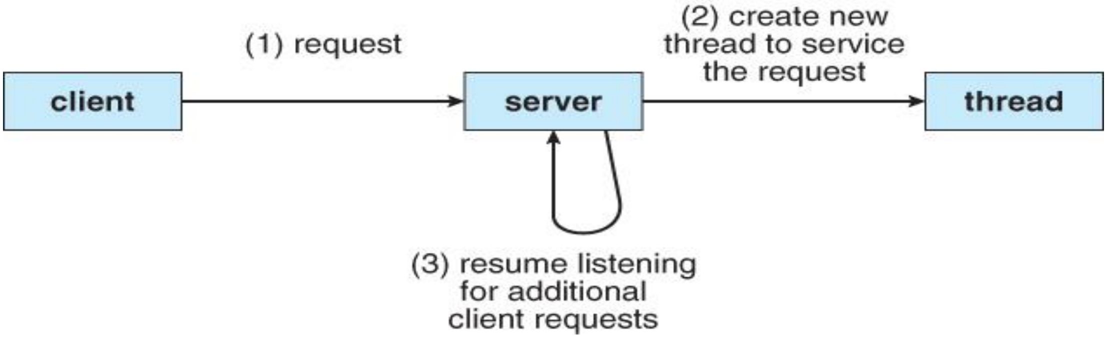
- A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- A traditional (or *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.
    - ✓ Motivation
    - ✓ Benefits



Single-Threaded and Multithreaded Processes

Motivation:

- Most software applications that run on modern computers are multithreaded.
- An application typically is implemented as a separate process with several threads of control.
- A web browser might have one thread display images or text while another thread retrieves data from the network.
- Ex: A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.
- In certain situations, a single application may be required to perform several similar tasks.
- Ex: A web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several clients concurrently accessing it.
- If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.
- One solution is to have the server run as a single process that accepts requests.
- When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular.
- Process creation is time consuming and resource intensive. It is generally more efficient to use one process that contains multiple threads.
- If the web-server process is multithreaded, the server will create a separate thread that listens for client requests.
- When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests.


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Multithreaded Server Architecture

- Threads also play a vital role in remote procedure call (RPC) systems.
- RPCs allow inter-process communication by providing a communication mechanism similar to ordinary function or procedure calls.
- Typically, RPC servers are multithreaded. When a server receives a message, it services the message using a separate thread. This allows the server to service several concurrent requests.
- Finally, most operating-system kernels are now multithreaded.
- Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling.
- Ex: Solaris has a set of threads in the kernel specifically for interrupt handling.
- Linux uses a kernel thread for managing the amount of free memory in the system.

Benefits

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

Responsiveness:

- ✓ Responsiveness
- ✓ Resource sharing
- ✓ Scalability
- ✓ Economy

- Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- This quality is especially useful in designing user interfaces.
- Ex: Consider what happens when a user clicks a button that results in the performance of a time-consuming operation.
- In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.

Resource sharing

- Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer.
- However, threads share the memory and the resources of the process to which they belong by default.
- 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.

Scalability:

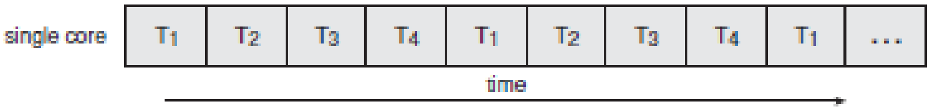
- The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.
- A single-threaded process can run on only one processor, regardless how many are available.

Economy:

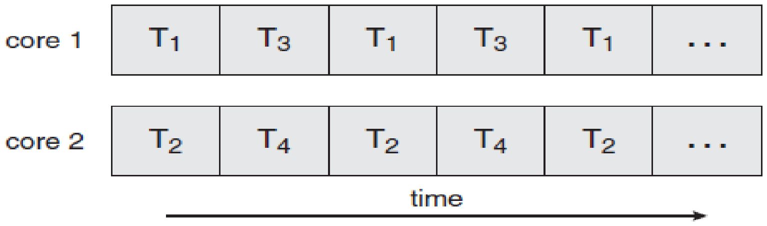
- Allocating memory and resources for process creation is costly.
- Threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
- Empirically gauging the difference in overhead can be difficult, but in general it is significantly more time consuming to create and manage processes than threads.
- In Solaris, Ex: Creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

MULTICORE PROGRAMMING

- Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems.
- A more recent, similar trend in system design is to place multiple computing cores on a single chip.
- Each core appears as a separate processor to the operating system. Whether the cores appear across CPU chips or within CPU chips, we call these systems multicore or multiprocessor systems.
- Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.
- Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time because the processing core is capable of executing only one thread at a time.
- On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core.



Concurrent Execution on a Single-Core System



Parallel Execution on a Multicore System

Parallelism and Concurrency:

- A system is parallel if it can perform more than one task simultaneously.
- In contrast, a concurrent system supports more than one task by allowing all the tasks to make progress.
- Thus, it is possible to have concurrency without parallelism.
- Before the advent of SMP and multicore architectures, most computer systems had only a single processor.
- CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes in the system, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.
- As systems have grown from tens of threads to thousands of threads, CPU designers have improved system performance by adding hardware to improve thread performance.
- Modern Intel CPUs frequently support two threads per core, while the Oracle T4 CPU supports eight threads per core.
- This support means that multiple threads can be loaded into the core for fast switching.

Programming Challenges:

- The trend towards multicore systems continues to place pressure on system designers and application programmers to make better use of the multiple computing cores.
- Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow the parallel execution.
- For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded.
- In general, five areas present challenges in programming for multicore systems:
  - ✓ Identifying tasks

- ✓ Balance
- ✓ Data splitting
- ✓ Data dependency
- ✓ Testing and debugging

Identifying tasks:

- This involves examining applications to find areas that can be divided into separate, concurrent tasks.
- Ideally, tasks are independent of one another and thus can run in parallel on individual cores.

Balance:

- While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.
- In some instances, a certain task may not contribute as much value to the overall process as other tasks.
- Using a separate execution core to run that task may not be worth the cost.

Data splitting:

- Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

Data dependency:

- The data accessed by the tasks must be examined for dependencies between two or more tasks.
- When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.

Testing and debugging:

- When a program is running in parallel on multiple cores, many different execution paths are possible.
- Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

Types of Parallelism

- There are two types of parallelism:
  - ✓ Data parallelism
  - ✓ Task parallelism
- **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.
- Consider, for example, summing the contents of an array of size *N*. On a single-core system, one thread would simply sum the elements [0] . . . [$N - 1$].
- On a dual-core system, however, thread *A*, running on core 0, could sum the elements [0] . . . [$N/2 - 1$] while thread *B*, running on core 1, could sum the elements [$N/2$] . . . [$N - 1$].
- The two threads would be running in parallel on separate computing cores.
- **Task parallelism** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation.
- Different threads may be operating on the same data, or they may be operating on different data.
- In contrast to that situation, an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements.
- The threads again are operating in parallel on separate computing cores, but each is performing a unique operation.
- Data parallelism - Distribution of data across multiple cores
- Task parallelism - Distribution of tasks across multiple cores.
- Few applications follow either data or task parallelism. In most instances, applications use a hybrid of these two strategies.

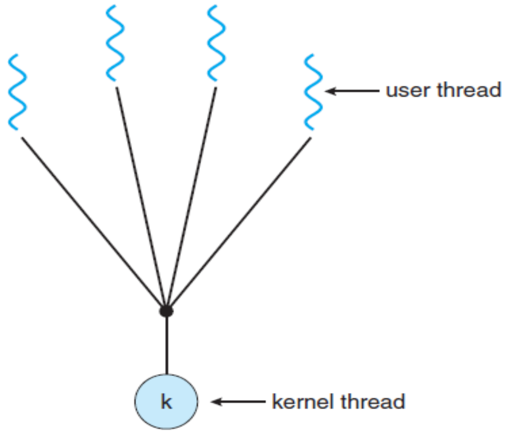
MULTITHREADING MODELS

- Support for the threads is provided at the user level, for user threads, by the kernel, for kernel threads.
- User threads are supported above the kernel, whereas kernel threads are supported and managed directly by the operating system.
- Virtually all contemporary operating systems including Windows, Linux, Mac OS X, and Solaris support kernel threads.

- A relationship must exist between user threads and kernel threads. The three common ways of establishing such a relationship are:
  - ✓ Many-to-one model
  - ✓ One-to-one model
  - ✓ Many-to many model

Many-to-One Model:

- The many-to-one model maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient.
- The entire process will block if a thread makes a blocking system call.
- Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.
- **Green threads** a thread library available for Solaris systems and adopted in early versions of Java used the many-to-one model.
- However, very few systems continue to use the model because of its inability to take advantage of multiple processing cores.

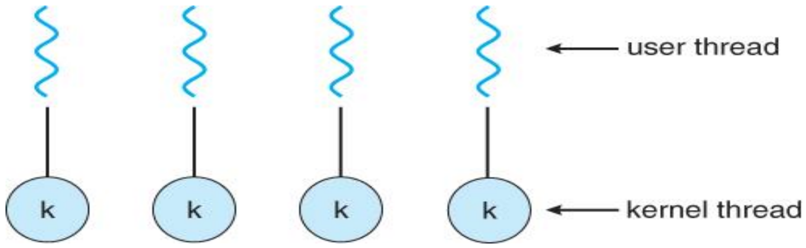


Many-to-one model

One-to-One Model

- The one-to-one model maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.

- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.
- Linux, along with the family of Windows operating systems, implement the one-to-one model.

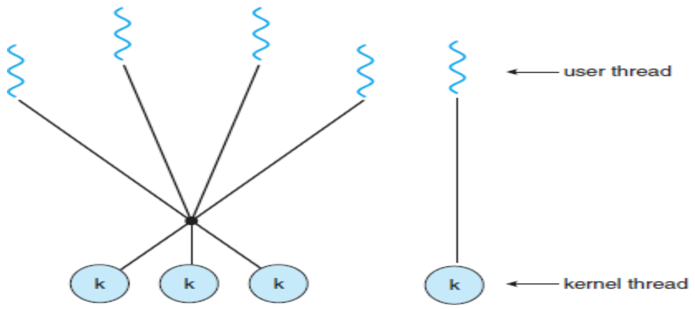


One-to-one model

Many-to-Many Model:

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine.
- The many to- one models allows the developer to create as many user threads as she wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time.
- The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application.
- The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- When a thread performs a blocking system call, the kernel can schedule another thread for execution.
- One variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread.
- This variation is sometimes referred to as the **two-level model**
- The Solaris operating system supported the two-level model in versions older than Solaris 9.

Many-to-many model



Two-level model

THREAD LIBRARIES

- A thread library provides the programmer with an API for creating and managing threads.
- There are two primary ways of implementing a thread library.
- 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.
- This means that invoking a function in the library results in a local function call in user space and not a system call.
- The second approach is to implement a kernel-level library supported directly by the operating system.
- In this case, code and data structures for the library exist in kernel space.
- Invoking a function in the API for the library typically results in a system call to the kernel.
- Three main thread libraries are available:
    - ✓ POSIX Pthreads
    - ✓ Windows
    - ✓ Java
- Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library.
- The Windows thread library is a kernel-level library available on Windows systems.
- The Java thread API allows threads to be created and managed directly in Java programs.
- The JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system.
- In Windows systems, Java threads are typically implemented using the Windows API; UNIX and Linux systems often use Pthreads.

Asynchronous and Synchronous threading

- There are two general strategies available for creating multiple threads:
    - ✓ Asynchronous threading
    - ✓ Synchronous threading.
- With asynchronous threading, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently.
- Each thread runs independently of every other thread, and the parent thread need not know when its child terminates.
- Because the threads are independent, there is typically little data sharing between threads.
- Synchronous threading occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes called ***fork-join*** strategy.
- Here, the threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed.
- Once each thread has finished its work, it terminates and joins with its parent.
- Only after all of the children have joined can the parent resume execution.
- Typically, synchronous threading involves significant data sharing among threads.
- The parent thread may combine the results calculated by its various children.

Pthreads:

- **Pthreads** refers to the POSIX standard defining an API for thread creation and synchronization. This is a ***specification*** for thread behavior, not an ***implementation***.
- Numerous systems implement the Pthreads specification; most are UNIX-type systems, including Linux, Mac OS X, and Solaris.
- Windows doesn’t support Pthreads.
- The C program is used to demonstrate the basic Pthreads API for constructing a multithreaded program.
- Each thread has a set of attributes, including stack size and scheduling information.
- A separate thread is created with the pthread create () function call.
- The parent thread will wait for the child thread to terminate by calling the pthread join () function.
- The child thread will terminate when it calls the function pthread exit ().
- Once the child thread has returned, the parent thread will output the value of the shared data sum.

Windows Threads:

- The technique for creating threads using the Windows thread library is similar to the Pthreads technique in several ways.
- Threads are created in the Windows API using the Create Thread () function and a set of attributes for the thread is passed to this function.
- The 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.
- Once the child thread is created, the parent must wait for it to complete before outputting the value of Sum.
- In the Windows API using the WaitForSingleObject() function, which causes the creating thread to block until the child thread has existed.
- In situations that require waiting for multiple threads to complete, the WaitForMultipleObjects() function is used.
- This function is passed four parameters:
  - ✓ The number of objects to wait for
  - ✓ A pointer to the array of objects
  - ✓ A flag indicating whether all objects have been signaled
  - ✓ A timeout duration (or INFINITE)
- For example, if THandles is an array of thread HANDLE objects of size N, the parent thread can wait for all its child threads to complete with this statement:

Ex: WaitForMultipleObjects(N, THandles, TRUE, INFINITE); **Java Threads:**

- Threads are the fundamental model of program execution in a Java.
- 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 even a simple Java program consisting of only a main() method runs as a single thread in the JVM.
- Java threads are available on all system that provides a JVM including Windows, Linux, and Mac OS X and Android applications as well.
- There are two techniques for creating threads in a Java program.
  - ✓ Create a new class that is derived from the Thread class and to override its run() method.
  - ✓ Define a class that implements the Runnable interface.

```
public interface Runnable

{

public abstract void run();

}
```

- Creating a Thread object does not specifically create the new thread; rather, the start() method creates the new thread.
- Calling the start() method for the new object does two things:
  - ✓ It allocates memory and initializes a new thread in the JVM.
  - ✓ It calls the run() method, making the thread eligible to be run by the JVM.
- If two or more threads are to share data in a Java program, the sharing occurs by passing references to the shared object to the appropriate threads.
- The join() method in Java is called to wait for the child threads to finish before proceeding.

IMPLICIT THREADING

- With the continued growth of multicore processing, applications containing hundreds or even thousands of threads are looming on the horizon.
- Programmers must design application to handle the difficulties, which relate to program correctness.
- One way to address these difficulties and better support the design of multithreaded applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries. This strategy is called as implicit threading.
- In this section, we explore three alternative approaches for designing multithreaded programs that can take advantage of multicore processors through implicit threading.

  - ✓ Thread Pools
  - ✓ OpenMP
  - ✓ Grand Central Dispatch

<u>Thread Pools:</u>

- Whenever the server receives a request, it creates a separate thread to service the request.
- Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems.

<u>Issues in creating Thread:</u>

- The first issue concerns the amount of time required to create the thread, together with the fact that the thread will be discarded once it has completed its work.

- The second issue is more troublesome. If we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system.
- Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a **thread pool**.
- The general 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 if one is available and passes it the request for service.
- Once the thread completes its service, it returns to the pool and awaits more work.
- If the pool contains no available thread, the server waits until one becomes free.
- Thread pools offer these benefits:
- Servicing a request with an existing thread is 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.
- Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task.
- 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 client requests.
- More sophisticated thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns.
- Such architectures provide the further benefit of having a smaller pool thereby consuming less memory when the load on the system is low.
- The Windows API provides several functions related to thread pools.
- Using the thread pool API is similar to creating a thread with the Thread Create() function.
- Here, a function that is to run as a separate thread is defined. Such a function may appear as follows:

```
PoolFunction(AVOID Param) {
/*
* this function runs as a separate thread.
*/
}
```

- A pointer to PoolFunction() is passed to one of the functions in the thread pool API, and a thread from the pool executes this function.

OpenMP:

- OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments.
- OpenMP identifies **parallel regions** as blocks of code that may run in parallel.
- Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel.
- When OpenMP encounters the directive, it creates as many threads are there are processing cores in the system.

```
#pragma omp parallel
```

- For a dual-core system, two threads are created, for a quad-core system, four are created; and so forth.
- All the threads then simultaneously execute in the parallel region. As each thread exits the parallel region, it is terminated.
- OpenMP is available on several open-source and commercial compilers for Linux, Windows, and Mac OS X systems.

Grand Central Dispatch:

- Grand Central Dispatch (GCD) a technology for Apple's Mac OS X and iOS operating systems.
- It is a combination of extensions to the C language, an API, and a run-time library that allows application developers to identify sections of code to run in parallel.
- Like OpenMP, GCD manages most of the details of threading.
- GCD identifies extensions to the C and C++ languages known as **blocks**.
- A block is simply a self-contained unit of work. It is specified by a caret ˆ inserted in front of a pair of braces *{ }*. A simple example of a block is shown below:

```
ˆ { printf(“ I am a block "); }
```

- GCD schedules blocks for run-time execution by placing them on a **dispatch queue**.
- When it removes a block from a queue, it assigns the block to an available thread from the thread pool it manages.
- GCD identifies two types of dispatch queues: ***serial*** and ***concurrent***.
- Blocks placed on a serial queue are removed in FIFO order.
- Once a block has been removed from the queue, it must complete execution before another block is removed.
- Each process has its own serial queue known as its **main queue**.

- Developers can create additional serial queues that are local to particular processes.
- Serial queues are useful for ensuring the sequential execution of several tasks.
- Blocks placed on a concurrent queue are also removed in FIFO order, but several blocks may be removed at a time, thus allowing multiple blocks to execute in parallel.
- There are three system-wide concurrent dispatch queues, and they are distinguished according to priority: low, default, and high.
- Priorities represent an approximation of the relative importance of blocks.
- GCD actively manages the pool, allowing the number of threads to grow and shrink according to application demand and system capacity.

THREAD ISSUES

- In this section, we discuss some of the issues to consider in designing multithreaded programs.
    - ✓ The fork() and exec() System Calls
    - ✓ Signal Handling
    - ✓ Thread Cancellation
    - ✓ Thread-Local Storage
    - ✓ Scheduler Activations

The fork() and exec() System Calls:

- The fork() system call is used to create a separate, duplicate process.
- The semantics of the fork() and exec() system calls change in multithreaded program.
- Some UNIX systems have chosen to have two versions of fork(),
    - ✓ One that duplicates all threads
    - ✓ The other 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.
- 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.
- If, however, the separate process does not call exec() after forking, the separate process should duplicate all threads.

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:
    - ✓ A signal is generated by the occurrence of a particular event.
    - ✓ The signal is delivered to a process.
    - ✓ Once delivered, the signal must be handled.
- Ex, of synchronous signal include illegal memory access and division by 0.
- Ex, of asynchronous signals includes terminating a process with specific keystrokes (such as <control><C>) and having a timer expires. An asynchronous signal is sent to another process.
- A signal may be ***handled*** by one of two possible handlers:
    - ✓ Default signal handler
    - ✓ User-defined signal handler
- Every signal has a **default signal handler** that the kernel runs when handling that signal.
- This default action can be overridden by a **user-defined signal handler** that is called to handle the signal. Signals are handled in different ways;
    - ✓ Some signals are simply ignored
    - ✓ Some signals terminate the program
- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads.
- In general, the following options exist:
    - ✓ Deliver the signal to the particular thread.
    - ✓ Deliver the signal to every thread in the process.
    - ✓ Deliver the signal to certain threads in the process.
    - ✓ Assign a specific thread to receive all signals for the process.

<u>Thread Cancellation:</u>

- **Thread cancellation** involves terminating a thread before it has completed.
- Ex, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
- When a user presses a button on a web browser that stops a web page from loading any further. Often, a web page loads using several threads each image is loaded in a separate thread.
- When a user presses the stop button on the browser, all threads loading the page are canceled.

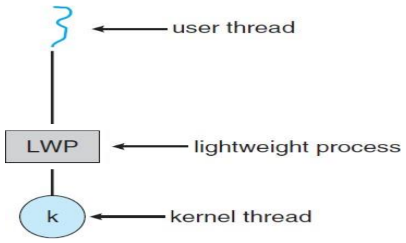
- A thread that is to be canceled is often referred to as the **target thread**.
- Cancellation of a target thread may occur in two different scenarios:
  - ✓ **Asynchronous cancellation**. One thread immediately terminates the target thread.
  - ✓ **Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

Thread-Local Storage:

- Threads belonging to a process share the data of the process. Indeed, this data sharing provides one of the benefits of multithreaded programming.
- However, in some circumstances, each thread might need its own copy of certain data; such data are called **thread-local storage** (or **TLS**.)
- For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier.
- To associate each thread with its unique identifier, we could use thread-local storage.
- TLS data are visible across function invocations. TLS data are unique to each thread.

Scheduler Activations:

- Many systems implementing either the many-to-many or the 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.**
- To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run.
- Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors.
- If a kernel thread blocks, the LWP blocks as well. The user-level thread attached to the LWP also blocks.



UNIT III

PROCESS SYNCHRONIZATION:

- A cooperating process is one that can affect or be affected by other processes executing in the system.
- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.
- Sharing data through logical address space is achieved through the use of threads.
- Concurrent access to shared data may result in data inconsistency.
- We have various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.
- The data consistency is maintained using the below mechanism;
  - ✓ Critical-Section Problem
  - ✓ Peterson's Solution
  - ✓ Synchronization Hardware
  - ✓ Mutex Locks
  - ✓ Semaphores
  - ✓ Monitors

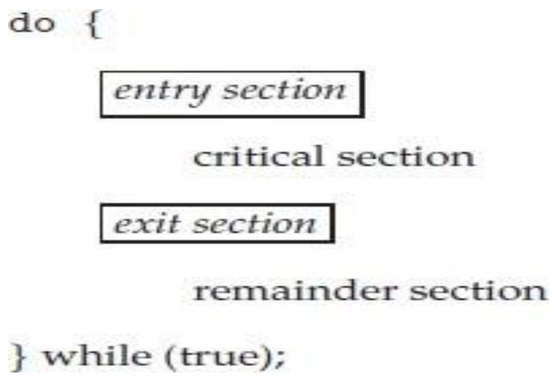
Race condition:

- 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 the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter.
- To make such a guarantee, we require that the processes be synchronized.

CRITICAL SECTION PROBLEM

- The first mechanism in process synchronization is critical-section problem.
- Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*−1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on.
- When one process is executing in its critical section, no other process is allowed to execute in its critical section. i.e. 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.

General Structure of a Typical Process:



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

- 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 *Pi* is shown in the fig. The entry section and exit section are enclosed in boxes to highlight these important segments of code.
- A solution to the critical-section problem must satisfy the following three requirements:

Mutual exclusion:

- If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.

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 deciding which will enter its critical section next.

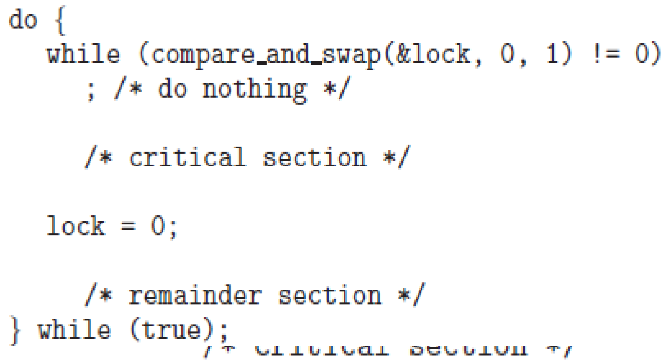
Bounded waiting:

- 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.

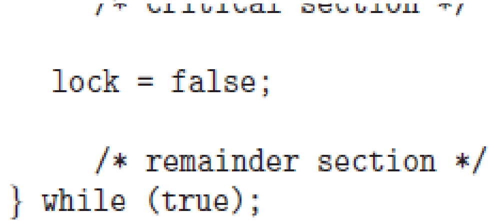
SYNCHRONIZATION HARDWARE

- Critical-section problem using techniques ranging from hardware to software-based APIs available to both kernel developers and application programmers.
- All these solutions are based on the premise of locking , that is, protecting critical regions through the use of locks.
- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified.

- In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption.
- No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonprimitive kernels.
- Unfortunately, this solution is not as feasible in a multiprocessor environment.
- Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.
- Also consider the effect on a system's clock if the clock is kept updated by interrupts.
- 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.



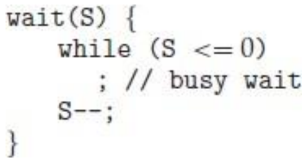
```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```



```
      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

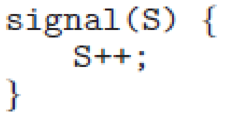
SEMAPHORES

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().
- The wait() operation was originally termed P (from the Dutch *proberen*, “to test”); signal() was originally called V (from *verhogen*, “to increment”).
- The definition of wait() is as follows:



```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- The definition of signal() is as follows:



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

- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.
- The semaphores can be used in the following ways;
  - ✓ Semaphore Usage
  - ✓ Semaphore Implementation
  - ✓ Deadlocks and Starvation
  - ✓ Priority Inversion

Semaphore 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.
- 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 the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

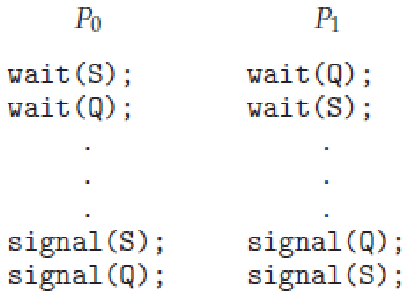
Semaphore Implementation

- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, 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.
- Then control is transferred to the CPU scheduler, which selects another process to execute.
- 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. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

<u>Deadlocks and Starvation:</u>

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be deadlocked.
- To illustrate this, consider a system consisting of two processes, *P*0 and *P*1, each accessing two semaphores, S and Q, set to the value 1:



```
     P0              P1

wait(S);        wait(Q);
wait(Q);        wait(S);
    .               .
    .               .
    .               .
signal(S);      signal(Q);
signal(Q);      signal(S);
```

- Suppose that *P*0 executes 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*0 executes 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.

<u>Priority Inversion</u>

- A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process or a chain of lower-priority processes.
- Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource.
- The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

- Process with a lower priority is changed to the process with higher priority. This problem is known as **priority inversion**. (Priority is changed).
- It occurs only in systems with more than two priorities, so one solution is to have only two priorities. These systems solve the problem by implementing a **priority- inheritance protocol**.
- According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources.
- When they are finished, their priorities revert to their original values.

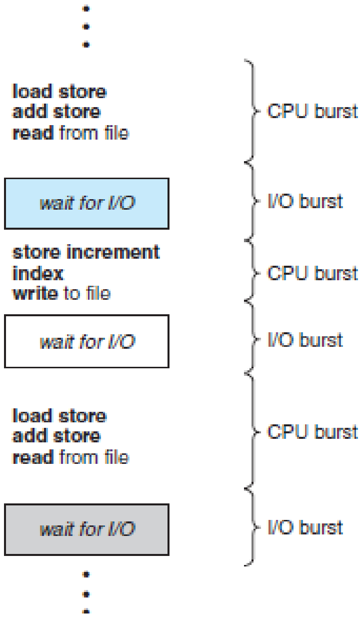
CPU SCHEDULING:

Basic Concept:

- CPU scheduling is the basis of multi-programmed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

CPU–I/O Burst Cycle:

- Process execution consists of a cycle of CPU execution and I/O wait.
- Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution.



CPU Scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.

- The selection process 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.
- 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 generally process control blocks (PCBs) of the processes.

Preemptive Scheduling

- CPU-scheduling decisions may take place under the following four circumstances:
    - ✓ When a process switches from the running state to the waiting state.
    - ✓ When a process switches from the running state to the ready state.
    - ✓ When a process switches from the waiting state to the ready state.
    - ✓ When a process terminates.
- Under non preemptive 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. This scheduling method was used by Microsoft Windows 3.x.
- Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling.
- 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**.

SCHEDULING CRITERIA

- Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.
- In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
- The criteria include the following:

CPU utilization:

- We want to keep the CPU 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 loaded 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 ten processes per second.

Turnaround time:

- From the point of view of a particular process, the important criterion is how long it takes to execute that process. 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:

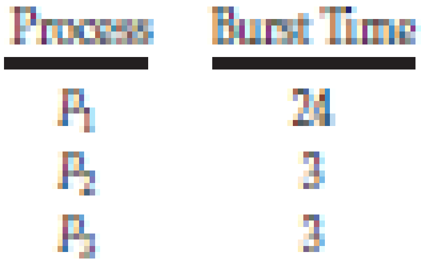
- A process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

SCHEDULING ALGORITHMS

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.
- There are many different CPU-scheduling algorithms available, they are;
  - ✓ First-Come, First-Served Scheduling
  - ✓ Shortest-Job-First Scheduling
  - ✓ Priority Scheduling
  - ✓ Round-Robin Scheduling
  - ✓ Multilevel Queue Scheduling
  - ✓ Multilevel Feedback Queue Scheduling

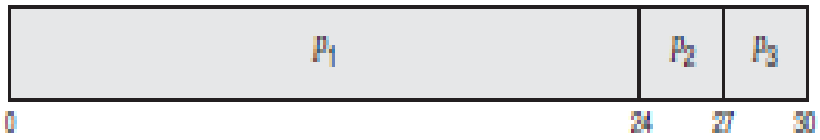
First-Come, First-Served Scheduling:

- 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.
- On the negative side, 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:

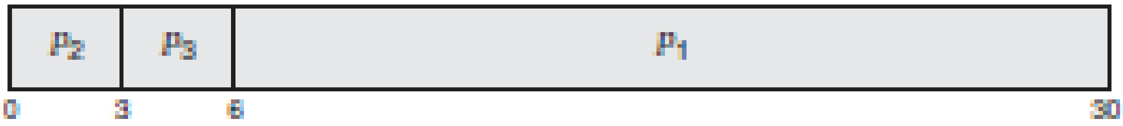


Process	Burst Time
P_1	24
P_2	3
P_3	3

- 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**, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



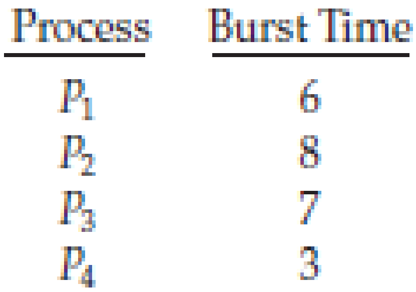
- The waiting time is 0 milliseconds for process *P*1, 24 milliseconds for process *P*2, and 27 milliseconds for process *P*3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.
- If the processes arrive in the order *P*2, *P*3, *P*1, however, the results will be as shown in the following Gantt chart:



- The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial.
- Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

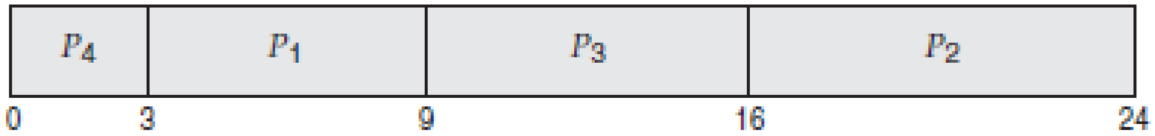
Shortest-Job-First Scheduling:

- This algorithm depends on 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.
- This scheduling method is also called as *shortest-next- CPU-burst* algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
- 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:

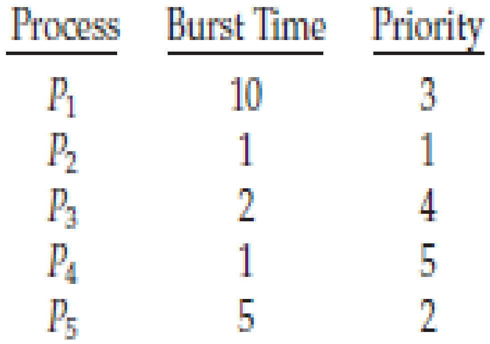


- The waiting time is 3 milliseconds for process *P*1, 16 milliseconds for process *P*2, 9 milliseconds for process *P*3, and 0 milliseconds for process *P*4.
- Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

Priority Scheduling

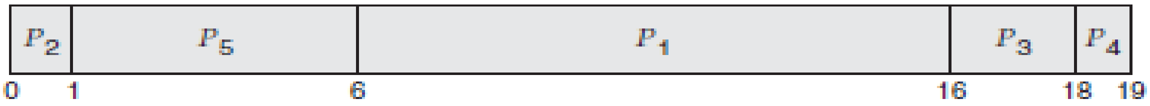
- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095.
- We assume that low numbers represent high priority.
- 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:

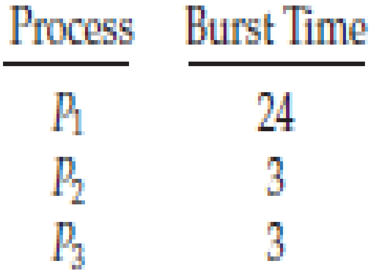


- The average waiting time is 8.2 milliseconds.
- A major problem with priority scheduling algorithms is indefinite blocking, or starvation.
- A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority 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 involves 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 enable the system 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 in length.
- 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 1time quantum.
- To implement RR scheduling, we again treat 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 1time quantum, and dispatches the process.

- 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.
- 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, process *P*2.
- Process *P*2 does not need 4 milliseconds, so it quits before its time quantum expires.
- The CPU is then given to the next process, 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 as follows:

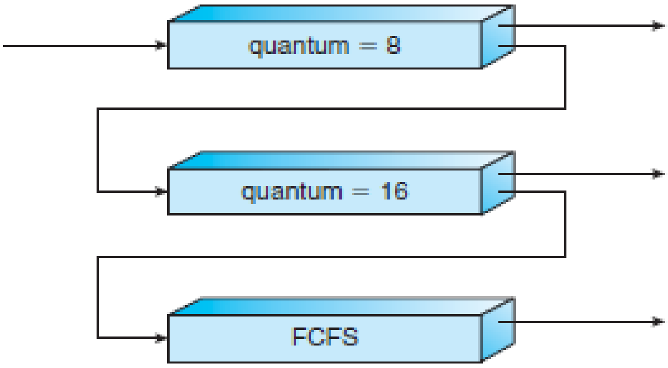
Multilevel Queue Scheduling

- Another class of scheduling algorithm 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 so may have different scheduling needs.
- In addition, foreground processes may have priority over background processes.
- A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues.
- The processes are permanently assigned to one queue, generally 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.
- 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.
- Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.
- For instance, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

<u>Multilevel Feedback Queue Scheduling</u>

- When the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system.
- Processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.
- The **multilevel feedback queue** scheduling algorithm, allows a process to move between queues.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- In addition, 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.



- The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1.

- Processes in queue 2 will be executed only 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 run on an FCFS basis but are run only when queues 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:
  - ✓ Number of queues
  - ✓ Scheduling algorithm for each queue
  - ✓ Method used to determine when to upgrade a process to a higher priority queue
  - ✓ Method used to determine when to demote a process to a lower priority queue
  - ✓ Method used to determine which queue a process will enter when that process needs service

THREAD SCHEDULING

- There are two types of threads *user-level* and *kernel-level* threads. The operating system supports the, kernel-level threads.
- 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 ultimately be mapped to an associated kernel- level thread; this mapping may be indirect and may use a lightweight process (LWP).
- In this section, we explore scheduling issues involving user-level and kernel-level threads and offer specific examples of scheduling for Pthreads.

- ✓ Contention Scope
- ✓ Pthread Scheduling

Contention Scope

- 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. This scheme is known as process contention scope (PCS).
- To decide which kernel-level 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.
- Systems using the one-to-one model such as Windows, Linux, and Solaris, schedule threads using only SCS.
- 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 and are not adjusted by the thread library.
- PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing among threads of equal priority.

Pthread Scheduling

- POSIX Pthread API that allows specifying PCS or SCS during thread creation.
- Pthreads identifies the following contention scope values:
  - ✓ 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, perhaps 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.

MULTIPROCESSOR SCHEDULING

- If multiple CPUs are available, load sharing becomes possible but scheduling problems become correspondingly more complex.
- If the processors are identical (homogeneous) in terms of their functionality. We can then use any available processor to run any process in the queue.

- ✓ Approaches to Multiple-Processor Scheduling
- ✓ Processor Affinity
- ✓ Load Balancing
- ✓ Multicore Processors

Approaches to Multiple-Processor Scheduling

- One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities 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.
- The schedulers for each processor examine the ready queue and select a process to execute.
- If we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully.
- We must ensure that two separate processors do not choose to schedule the same process and that processes are not lost from the queue.
- All modern operating systems support SMP, including Windows, Linux, and Mac OS X.

Processor Affinity

- When a process has been running on a specific processor. The data most recently accessed by the process populate the cache for the processor.
- If the process migrates to another processor. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated.
- Because of the high cost of invalidating and repopulating caches, most systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as processor affinity.
- Processor affinity takes two forms.
  - ✓ Soft affinity
  - ✓ Hard affinity
- The operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors. This is called soft affinity.

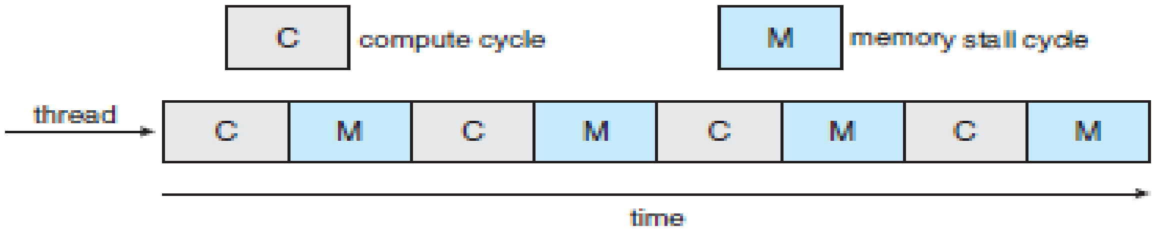
- Some systems provide system calls that support hard affinity, thereby allowing a process to specify a subset of processors on which it may run.

Load Balancing

- Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.
- It is important to note that load balancing is typically necessary only on systems where each processor has its own private queue of eligible processes to execute.
- On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.
- There are two general approaches to load balancing:
  - ✓ Push migration
  - ✓ Pull migration.
- In 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.
- The benefit of keeping a process running on the same processor is that the process can take advantage of its data being in that processor's cache memory.
- Either pulling or pushing a process from one processor to another removes this benefit.

Multicore Processors

- A recent practice in computer hardware has been to place multiple processor cores on the same physical chip, resulting in a multicore processor.
- Each core maintains its architectural state and thus appears to the operating system to be a separate physical processor.
- SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip.



- When a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a memory stall.a

DEADLOCK:

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; if the resources arenot available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

EX:

- “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

DEADLOCK CHARACTERIZATION

- In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.
- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
    - ✓ Mutual exclusion.
    - ✓ Hold and wait
    - ✓ No preemption
    - ✓ Circular wait

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 that 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 other processes.

No preemption:

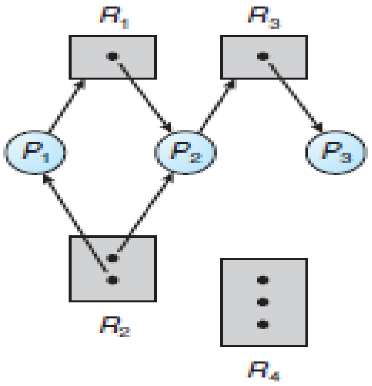
- Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait:

- A set $\{P0, P1, ..., Pn\}$ of waiting processes must exist such that $P0$ is waiting for a Resource held by $P1$, $P1$ is waiting for a resource held by $P2$, ..., $Pn-1$ is waiting for a resource held by Pn, and Pn is waiting for a resource held by $P0$.

Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices *V* and a set of edges *E*.
- The set of vertices *V* is partitioned into two types of nodes:
  - ✓ *P* = {*P*1, *P*2, ..., *Pn*}, the set consisting of all the active processes in the System
  - ✓ *R* = {*R*1, *R*2, ..., *Rm*}, the set consisting of all resource types in the system.
- A directed edge from process *Pi* to resource type *Rj* is denoted by *Pi* → *Rj*, it signifies that process *Pi* has requested an instance of resource type *Rj* and is currently waiting for that resource.
- A directed edge from resource type *Rj* to process *Pi* is denoted by *Rj* → *Pi*, it signifies that an instance of resource type *Rj* has been allocated to process *Pi* .
- A directed edge *Pi* → *Rj* is called a request edge; a directed edge *Rj* → *Pi* is called an assignment edge.
- We represent each process *Pi* as a circle and each resource type *Rj* as a rectangle. Since resource type *Rj* may have more than one instance, we represent each such instance as a dot within the rectangle.
- A request edge points to only the rectangle *Rj* , whereas an assignment edge must also designate one of the dots in the rectangle.



- When process *Pi* requests an instance of resource type *Rj* , a request edge is inserted in the resource-allocation graph.

  When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge.

- When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.
- The resource-allocation graph shown in above has the following situation.

1. The sets *P*, *R*, and *E*:

- ✓ *P* = {*P*1, *P*2, *P*3}
- ✓ *R* = {*R*1, *R*2, *R*3, *R*4}

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

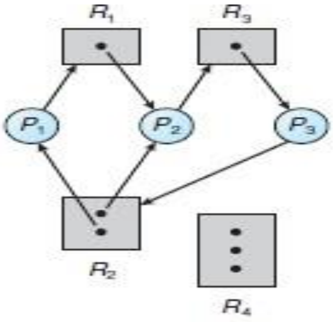
2. Resource instances:

- ✓ 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

3. 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.

- The definition of a resource-allocation graph, is that,
  - ✓ If the graph contains no cycles, then no process in the system is deadlocked.
  - ✓ If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
- If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.
- Each process involved in the cycle is deadlocked.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.
- In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.



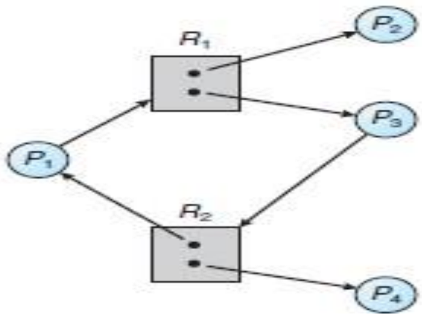
Resource-allocation graph with a deadlock.

- Suppose that process *P*3 requests an instance of resource type *R*2. Since no resource instance is currently available, we add a request edge *P*3→ *R*2 to the graph.
- At this point, 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*1 or process *P*2 to release resource *R*2.
- In addition, process *P*1 is waiting for process *P*2 to release resource *R*1.
- Now consider the resource-allocation graph below. In this example, we also have a cycle but no deadlock:

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$



- However, there is no deadlock. Observe that process *P*4 may release its instance of resource type *R*2. That resource can then be allocated to *P*3, breaking the cycle.

METHODS FOR HANDLING DEADLOCKS

- The deadlock can be resolved in three ways:

1. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state.
2. We can allow the system to enter a deadlocked state, detect it, and recover.
3. We can ignore the problem altogether and pretend that deadlocks never occur in the system.

- The third solution is the one used by most operating systems, including Linux and Windows. It is then up to the application developer to write programs that handle deadlocks.
- The system can use either a deadlock prevention or a deadlock-avoidance scheme.
- **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.
- **Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.

- With this additional knowledge, the operating system can decide for each request whether or not the process should wait.
- To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available,
- The resources currently allocated to each process, and the future requests and releases of each process.

DEADLOCK PREVENTION

- For a deadlock to occur, each of the four necessary conditions must hold.
- By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock.

  - ✓ **Mutual Exclusion**
  - ✓ **Hold and Wait**
  - ✓ **No Preemption**
  - ✓ **Circular Wait**

Mutual Exclusion

- The mutual exclusion condition must hold. That is, at least one resource must be non-sharable.
- Ex: Read-only files are a good example of a sharable resource.
- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.

Hold and Wait

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when it has none.
- A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.
- Both these protocols have two main disadvantages:
- First, resource utilization may be 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

- There will be no preemption of resources that have already been allocated.
- If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.
- In otherwords, these resources are implicitly released.
- The preempted resources are added to the list of resources for which the process is waiting.
- The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- If a process requests some resources, we first check whether they are available. If they are, we allocate them.
- If they are not, we check whether they are allocated to some other process that is waiting for additional resources.
- If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.
- If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them.
- A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.
- This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space.

Circular Wait

- The fourth and final condition for deadlocks is the circular-wait condition.
- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- Let $R = \{R1, R2, ..., Rm\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.
- Formally, we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers.
- For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:
  - ✓ F(tape drive) = 1
  - ✓ F(disk drive) = 5

- ✓ *F*(printer) = 12

- Each process can request resources only in an increasing order of enumeration.
- That is, a process can initially request any number of instances of a resource type say, *Ri* .
- After that, the process can request instances of resource type *Rj* if and only if $F(Rj) > F(Ri)$.
- For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.
- Alternatively, we can require that a process requesting an instance of resource type *Rj* must have released any resources *Ri* such that $F(Ri) \geq F(Rj)$.

DEADLOCK AVOIDANCE

- Deadlock-prevention algorithms prevent deadlocks by limiting how requests can be made.
- The limits ensure that at least one of the necessary conditions for deadlock cannot occur.
- Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.
- The resource allocation *state* is defined by the number of available and allocated resources and the maximum demands of the processes.
- In the following sections, we explore two deadlock-avoidance algorithms.
  - ✓ Resource-Allocation-Graph Algorithm
  - ✓ Banker’s Algorithm

<u>Safe State</u>

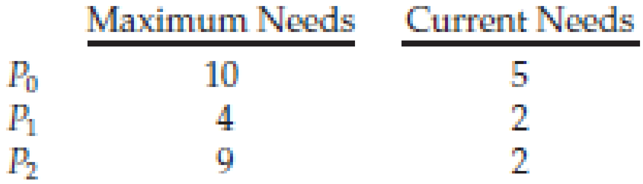
- A state is *safe* if the system can allocate resources to each process in some order and still avoid a deadlock.
- A system is in a safe state only if there exists a **safe sequence**.
- A sequence of processes <*P*1, *P*2, ..., *Pn*> is a safe sequence for the current allocation state if, for each *Pi* , the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *Pj*, with $j < i$.
- if the resources that *Pi* needs are not immediately available, then *Pi* can wait until all *Pj* have finished.
- When they have finished, *Pi* can obtain all of its needed resources, complete its designated

task, return its allocated resources, and terminate.

- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state.

Example:

- A system with twelve magnetic tape drives and three processes: *P*0, *P*1, and *P*2.
- Process *P*0 requires ten tape drives, process *P*1 may need as many as four tape drives, and process *P*2 may need up to nine tape drives.
- Suppose that, at time *t*0, process *P*0 is holding five tape drives, process *P*1 is holding two tape drives, and process *P*2 is holding two tape drives.



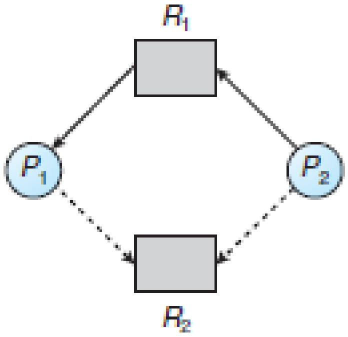
	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- Process *P*1 can immediately be allocated all its tape drives and then return them; then process *P*0 can get all its tape drives and return them; and finally process *P*2 can get all its tape drives and return them.

Resource-Allocation-Graph Algorithm

- In addition to the request and assignment edges already described, we introduce a new type of edge, called a **claim edge**.
- A claim edge *Pi* → *Rj* indicates that process *Pi* may request resource *Rj* at some time in the future.
- This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- When process *Pi* requests resource *Rj* , the claim edge *Pi* → *Rj* is converted to a request edge.
- Before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.
- We can relax this condition by allowing a claim edge *Pi* → *Rj* to be added to the graph only if all the edges associated with process *Pi* are claim edges.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state.
- we consider the resource-allocation graph of below. Suppose that *P*2 requests *R*2. Although *R*2 is currently free, we cannot allocate it to *P*2, since this action will create a cycle in the graph.
- 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

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.
- The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the **banker’s algorithm.**
- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
- This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
- The following data structures are used in Bankers algorithm, where *n* is the number of processes in the system and *m* is the number of resource types:

Available: A vector of length *m* indicates the number of available resources of each type.

Max: An *n* × *m* matrix defines the maximum demand of each process.

Allocation: An *n* × *m* matrix defines the number of resources of each type currently allocated to each process.

Need: An *n* × *m* matrix indicates the remaining resource need of each process.

$$\boldsymbol{Need[i][j] = Max[i][j] - Allocation[i][j]}.$$

1. Safety Algorithm

- The algorithm for finding out whether or not a system is in a safe state.

2. Resource-Request Algorithm

- The algorithm for determining whether requests can be safely granted.

DEADLOCK DETECTION

- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.

1. An algorithm that examines the state of the system to determine whether a deadlock has occurred
2. An algorithm to recover from the deadlock

- The detection-and-recovery scheme requires overhead that includes not only the run- time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.
    - ✓ Single Instance of Each Resource Type
    - ✓ Several Instances of a Resource Type

Single Instance of Each Resource Type

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph.
- We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

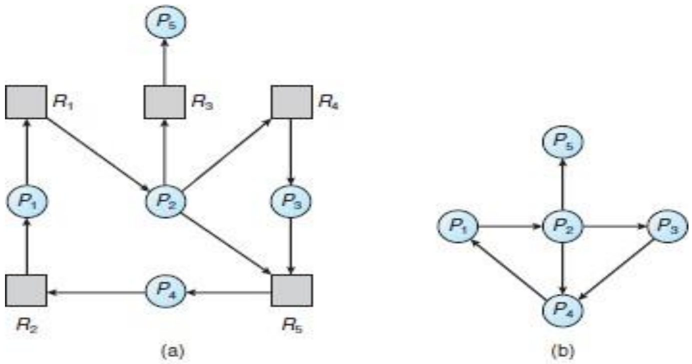


Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

- More precisely, an edge from *Pi* to *Pj* in a wait-for graph implies that process *Pi* is waiting for process *Pj* to release a resource that *Pi* needs.
- An edge *Pi* → *Pj* exists in a wait-for graph if and only if the corresponding resource

allocation graph contains two edges *Pi* → *Rq* and *Rq* → *Pj* for some resource *Rq* .

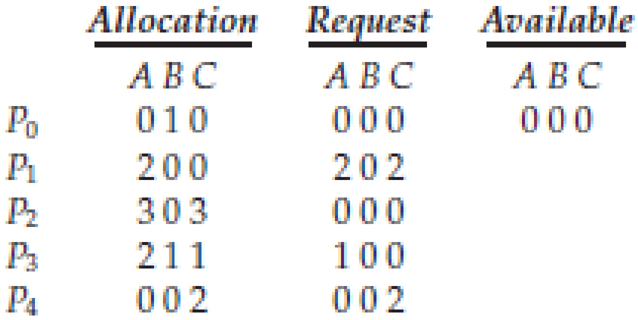
- In Figure we present a resource-allocation graph and the corresponding wait-forgraph.
- A deadlock exists in the system if and only if the wait-for graph contains a cycle.
- To detect deadlocks, the system needs to ***maintain*** the wait for graph and periodically ***invoke an algorithm*** that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of *n*2 operations, where *n* is the number of vertices in the graph.

Several Instances of a Resource Type

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
- We turn now to a deadlock detection algorithm that is applicable to such a system.
- The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm:

Available: A vector of length *m* indicates the number of available resources of each type. **Allocation:** An *n* × *m* matrix defines the number of resources of each type currently allocated to each process. **Request:** A n *n* × *m* matrix indicates the current request of each process.

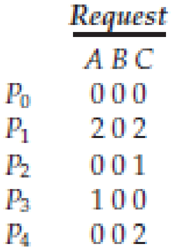
- For example, we consider a system with five processes *P*0 through *P*4 and three resource types *A*, *B*, and *C*.
- Resource type *A* has seven instances, resource type *B* has two instances, and resource type *C* has six instances.
- Suppose that, at time *T*0, we have the following resource-allocation state:



	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <*P*0, *P*2, *P*3, *P*1, *P*4> results in ***Finish***[*i*]== *true* for all *i*.
- Suppose now that process *P*2 makes one additional request for an instance of type *C*.

The ***Request*** matrix is modified as follows:



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

- We claim that the system is now deadlocked. Although we can reclaim the resources held by process *P*0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes *P*1, *P*2, *P*3, and *P*4.

Detection-Algorithm Usage

- When should we invoke the detection algorithm? The answer depends on two factors:
  1. How ***often*** is a deadlock likely to occur?
  2. How ***many*** processes will be affected by deadlock when it happens?
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
- Resources allocated to deadlocked processes will be idle until the deadlock can be broken.
- In addition, the number of processes involved in the deadlock cycle may grow.
- Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes.

RECOVERY FROM DEADLOCK

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock.
  - ✓ Abort one or more processes to break the circular wait.
  - ✓ Preempt some resources from one or more of the deadlocked processes.

Process Termination

- To eliminate deadlocks by aborting a process, we use one of two methods

Abort all deadlocked processes: This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later. **Abort one process at a time until the deadlock cycle is eliminated**: This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still dead locked.

- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.
- Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.
- we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term ***minimum cost*** is not a precise one.
- Many factors may affect which process is chosen, including:

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

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.
- If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim**:

- As in process termination, we must determine the order of preemption to minimize cost and the amount of time the process has thus far consumed.

2, Rollback:

- If we preempt a resource from a process, it cannot continue with its normal execution; it is missing some needed resource.
- We must roll back the process to some safe state and restart it from that state.

- It is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it.
- This method requires the system to keep more information about the state of all running processes.

3. <u>Starvation:</u>

- In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim.
- As a result, this process never completes its designated task, a starvation situation any practical system must address.
- We must ensure that a process can be picked as a victim only a (small) finite number of times.
- The most common solution is to include the number of rollbacks in the cost factor.

UNIT IV

MEMORY MANAGEMENT: MAIN MEMORY

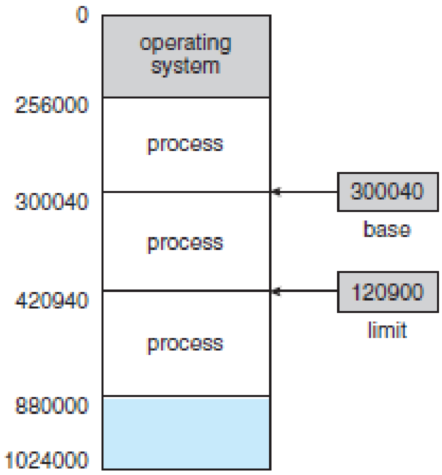
Background

- Memory is central to the operation of a modern computer system.
- Memory consists of a large array of bytes, each with its own address.
- The CPU fetches instructions from memory according to the value of the program counter.
- These instructions may cause additional loading from and storing to specific memory addresses.
- A typical instruction-execution cycle, for example, first fetches an instruction from memory.
- The instruction is then decoded and may cause operands to be fetched from memory.
- After the instruction has been executed on the operands, results may be stored back in memory.
- There are several issues that are related to managing memory:
    - ✓ Basic hardware
    - ✓ The binding of symbolic memory addresses to actual physical addresses,
    - ✓ Distinction between logical and physical addresses.
    - ✓ Dynamic loading
    - ✓ Dynamic linking and shared libraries.

Basic Hardware

- Main memory and the registers built into the processor itself are the only general- purpose storage that the CPU can access directly.
- Any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- If the data are not in memory, they must be moved there before the CPU can operate on them.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.
- Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
- The main memory is accessed via a transaction on the memory bus.
- Completing a memory access may take many cycles of the CPU clock.
- In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing.
- This situation is intolerable because of the frequency of memory accesses.
- The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access.

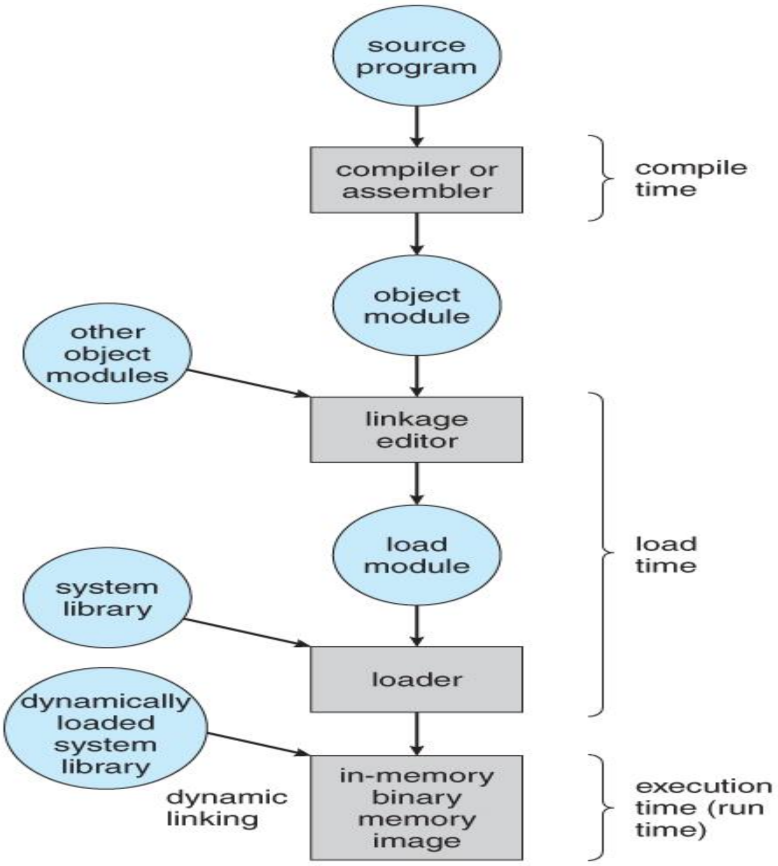
- Each process has a separate memory space. Separate per-process memory space protects the processes from each other.
- It is fundamental to having multiple processes loaded in memory for concurrent execution.
- To separate memory spaces, we need the ability to determine the range of legal addresses 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,
  - ✓ base register
  - ✓ limit registers.
- The **base register** holds the smallest legal physical memory address;
- the **limit register** specifies the size of the range.



Address Binding

- A program resides on a disk as a binary executable file.
- To be executed, the program must be brought into memory and placed within a process.
- Depending on the memory management in use, the process may be moved between disk and memory during its execution.
- The processes on the disk that are waiting to be brought into memory for execution form the **input queue**.
- The normal single-tasking procedure is to select one of the processes in the input queue and to load that process into memory.
- As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.
- A compiler typically **binds** these symbolic addresses to relocatable addresses.
- The linkage editor or loader in turn binds the relocatable addresses to absolute addresses.

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



Multistep processing of a user program

1. <u>Compile time:</u> If you know at compile time where the process will reside in memory, then **absolute code** can be generated.

2. <u>Load time:</u> 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. If the starting address changes, we need only reload the user code to incorporate this changed value.

3. <u>Execution time:</u> If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

<u>Logical Versus Physical Address Space</u>

- An address generated by the CPU is commonly referred to as a **logical address.**
- An address seen by the memory unit, the one loaded into the **memory-address register** of the memory is commonly referred to as a **physical address**.

- The compile-time and load-time address-binding methods generate identical logical and physical addresses.
- However, the execution-time address binding scheme results in differing logical and physical addresses.
- In this case, we usually refer to the logical address as a **virtual address**. We use ***logical address*** and ***virtual address*** interchangeably in this text.
- The set of all logical addresses generated by a program is a **logical address space**.
- The set of all physical addresses corresponding to these logical addresses is a **physical address space**.
- Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**.
- 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.

Dynamic Loading

- It is necessary for the entire program and all data of a process to be in physical memory for the process to execute.
- The size of a process has thus been limited to the size of physical memory.
- To obtain better memory-space utilization, we can use **dynamicloading**. With dynamic loading, a routine is not loaded until it is called.
- All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed.
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program’s address tables to reflect this change.
- Then control is passed to the newly loaded routine.

Dynamic Linking and Shared Libraries

- **Dynamically linked libraries** are system libraries that are linked to user programs when the programs are run.

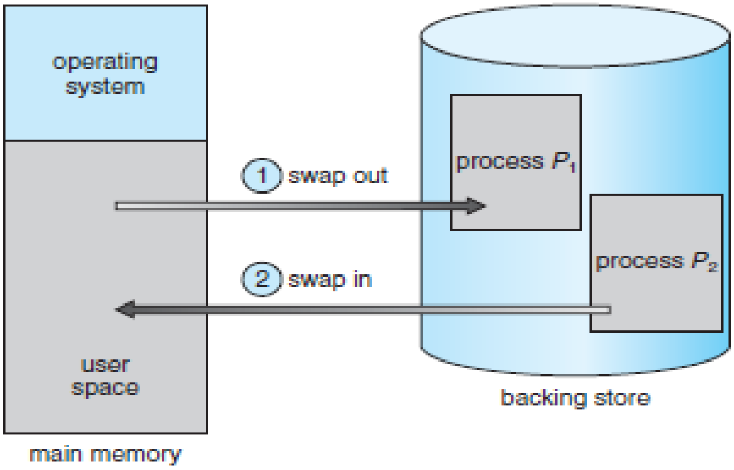
- Some operating systems support only **static linking**, in which system libraries are treated like any other object module and are combined by the loader into the binary program image.
- This feature is usually used with system libraries, such as language subroutine libraries.
- Without this facility, each program on a system must include a copy of its language library in the executable image.
- With dynamic linking, a **stub** is included in the image for each library routine reference.
- The stub is a small piece of code that indicates how to locate library routine or how to load the library if the routine is not already present.

SWAPPING

- A process must be in memory to be executed.
- A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
  - ✓ Standard Swapping
  - ✓ Swapping on Mobile Systems

Standard Swapping

- Standard swapping involves moving processes between main memory and a backing store.
- The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.
- The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.



Swapping on Mobile Systems

- Mobile systems typically do not support swapping in any form.
- Mobile devices generally use flash memory rather than more spacious hard disks as their persistent storage.

Reason:

1. Limited number of writes that flash memory can tolerate

2. Poor throughput between main memory and flash memory in these devices.

CONTIGUOUS MEMORY ALLOCATION

- The main memory must accommodate both the operating system and the various user processes.
- The memory is usually divided into two partitions: 1.One for the resident operating system
  2. One for the user processes.
- We usually want several user processes to reside in memory at the same time.
- We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.
- In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.
  - ✓ Memory Allocation
  - ✓ Fragmentation

Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**.
- Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions.
- In this **multiple partition method**, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.
- In the **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**.
- The memory blocks available comprise a ***set*** of holes of various sizes scattered throughout memory.

- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
- If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
- If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
- This procedure is a particular instance of the general **dynamic storage allocation problem**, which concerns how to satisfy a request of size *n* from a list of free holes.
- There are many solutions to this problem. The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

1. First fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

2. Best fit: Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

3. Worst fit: Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole.

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**.
- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- Storage is fragmented into a large number of small holes. This fragmentation problem can be severe.
- In the worst case, we could have a block of free (or wasted) memory between every two processes.
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

- The memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.

Solution:

- One solution to the problem of external fragmentation is **compaction**.
- The goal is to shuffle the memory contents so as to place all free memory together in one large block.

SEGMENTATION

- Segmentation is a memory management technique in which, the memory is divided into variable size parts.
- Each part is known as segment which can be allocated to a process.
- The details about each segment are stored in a table called as segment table.

Basic Method:

- A logical address space is a collection of segments. Each segment has a name and a length.
- The addresses specify both the segment name and the offset within the segment.
- The programmer therefore specifies each address by two quantities: a segment name and an offset.
- Offset helps to locate address inside a memory segment.
- For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.
- Thus, a logical address consists of a *two tuple:*

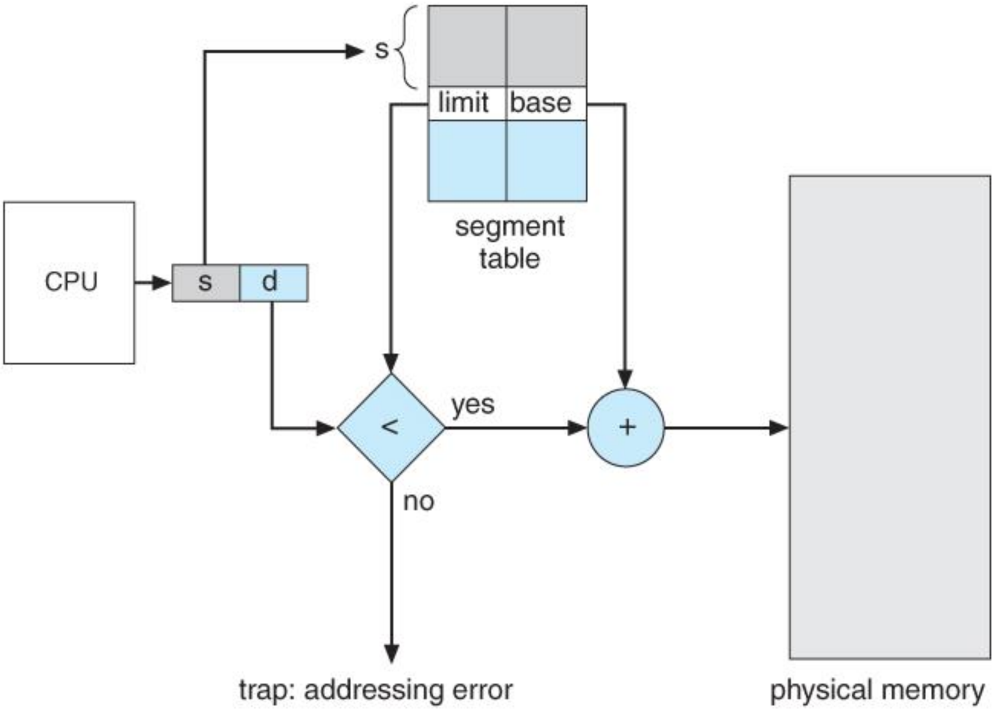
  <segment-number, offset>.
- Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.
- A compiler might create separate segments for the following:
  - ✓ The code
  - ✓ Global variables
  - ✓ The heap, from which memory is allocated
  - ✓ The stacks used by each thread
  - ✓ The standard C library
- Libraries that are linked in during compile time might be assigned separate segments.

- The loader would take all these segments and assign them segment numbers.

Segmentation Hardware

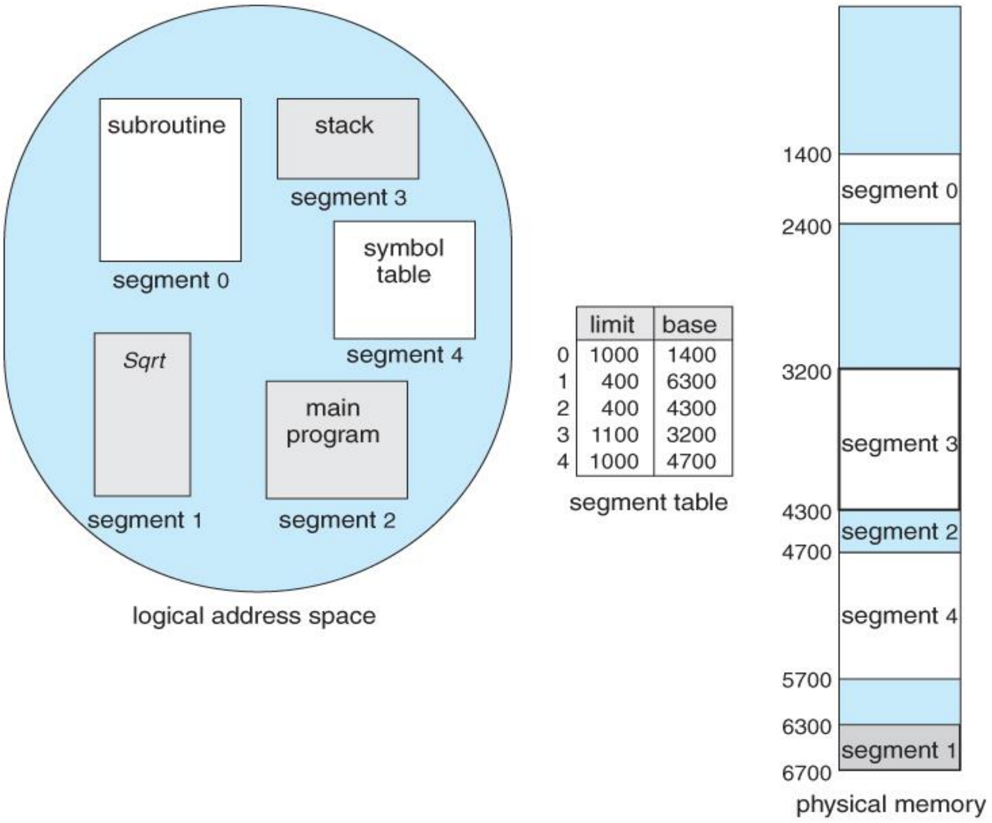
- The Segmentation Hardware define an implementation to map two-dimensional user- defined addresses into one-dimensional physical addresses. This mapping is affected by a **segment table**.
- Each entry in the segment table has a **segment base** and a **segment limit**.
- The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.
- A logical address consists of two parts: a segment number, *s,*

  an offset into that segment, *d.*
- The segment number is used as an index to the segment table.
- The offset *d* of the logical address must be between 0 and the segment limit.



Segmentation Hardware

- We have five segments numbered from 0 through 4. The segments are stored in physical memory.
- The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).
- Ex: segment 2 is 400 bytes long and begins at location 4300.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

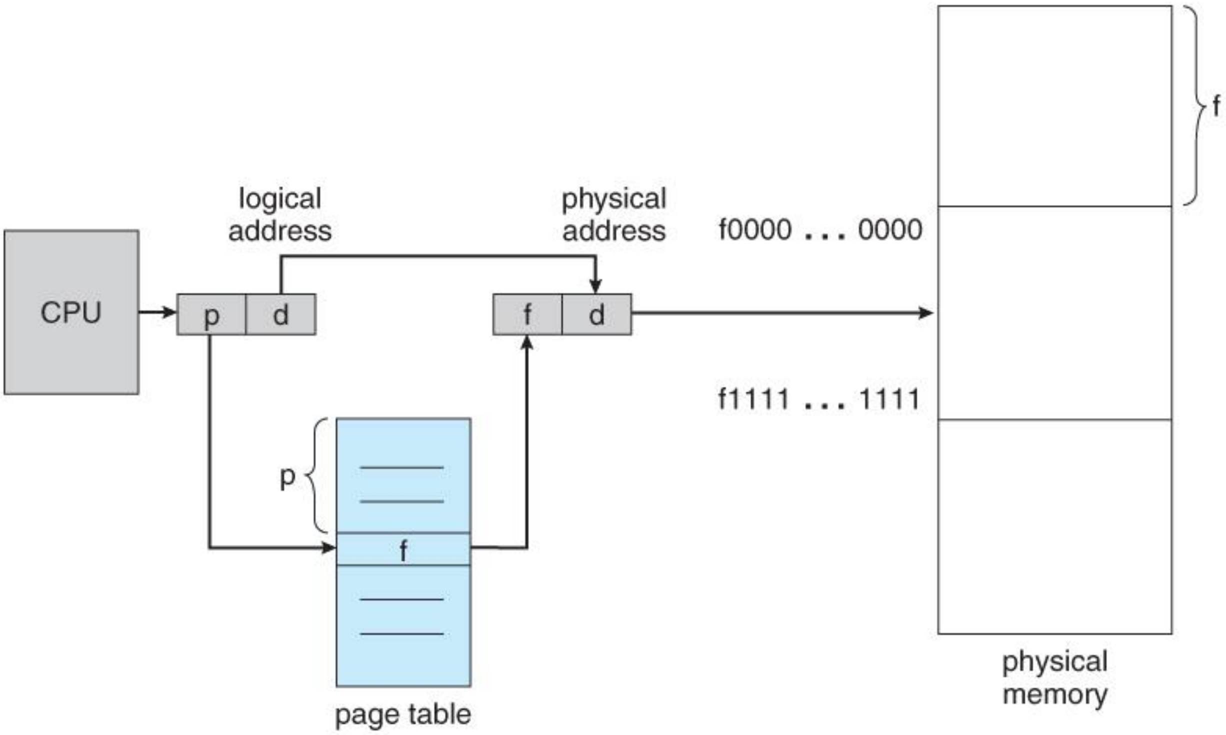
PAGING

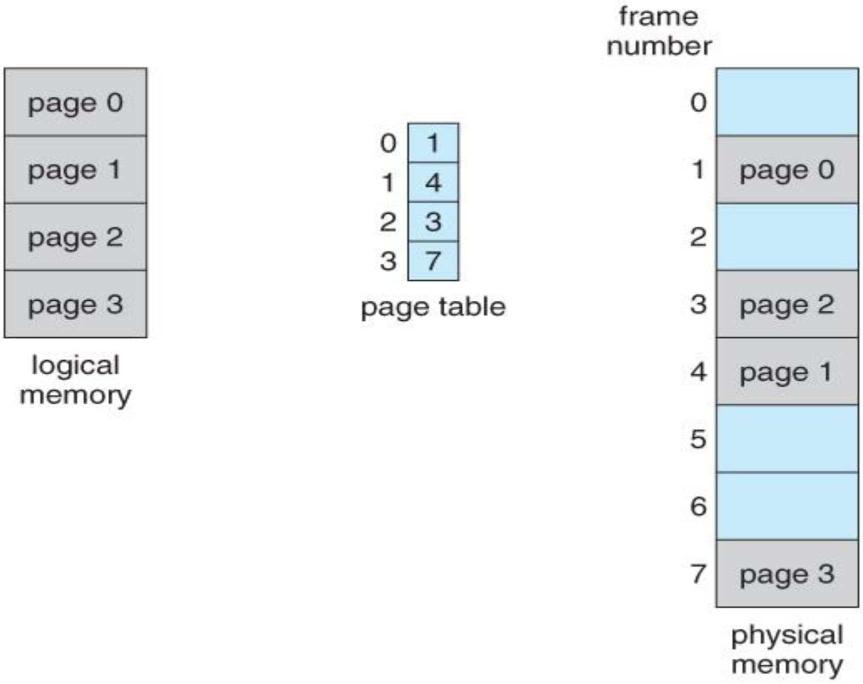
- Segmentation permits the physical address space of a process to be noncontiguous.
- Paging is another memory-management scheme by which a computer stores and retrieves data from secondary storage in same-size blocks called pages.
- Paging is implemented through cooperation between the operating system and the computer hardware.

Basic Method

- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.
- Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d).
- The page number is used as an index into a page table. The page table contains the base address of each page in physical memory.

- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.
- The paging model of memory is shown below





- The operating system is managing physical memory, it must be aware of the allocation details of physical memory like

- ✓ which frames are allocated,
- ✓ which frames are available,
- ✓ how many total frames there

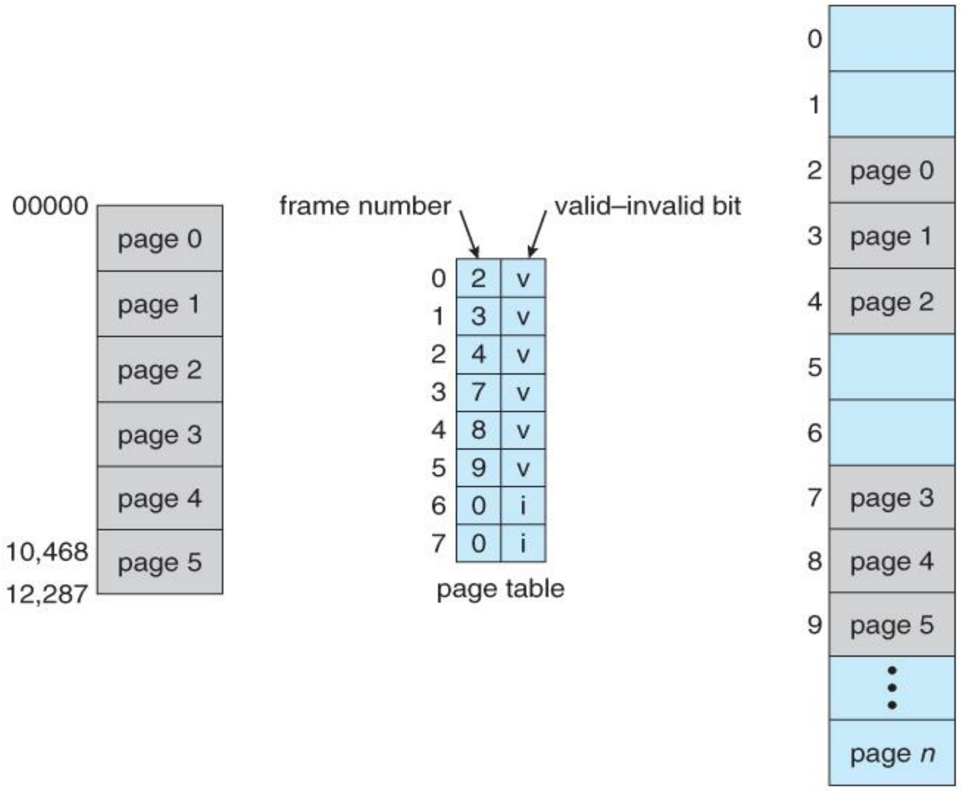
- This information is generally kept in a data structure called a **frame table**.

Hardware Support

- The hardware implementation of the page table can be done in several ways.
- In the simplest case, the page table is implemented as a set of dedicated **registers**, if the page table is reasonably small (246 entries).
- If the page table is very large (1 million entries), the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table.

Protection

- Memory protection in a paged environment is accomplished by protection bits associated with each frame. These bits are kept in the page table.
- One bit can define a page to be read–write or read-only.
- The protection bits can be checked to verify that no writes are being made to a read- only page.
- An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).
- One additional bit is generally attached to each entry in the page table: a **valid–invalid** bit.
- When this bit is set to *valid*, the associated page is in the process's logical address space and is thus a legal (or valid) page.
- When the bit is set to *invalid*, the page is not in the process's logical address space.
- **Page-table length register (PTLR)**, is used to indicate the size of the page table.

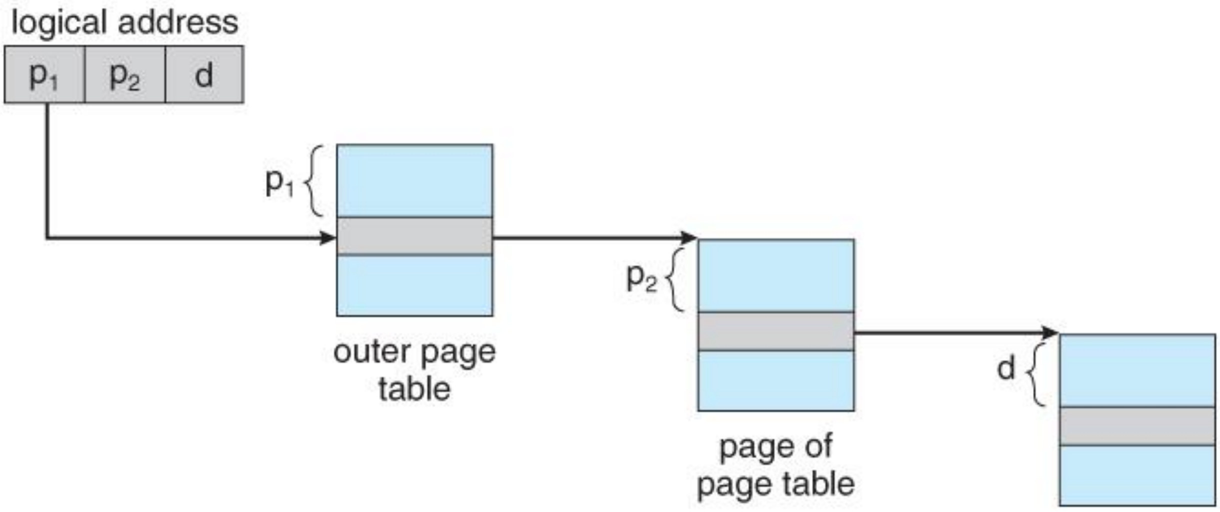


STRUCTURE OF THE PAGE TABLE

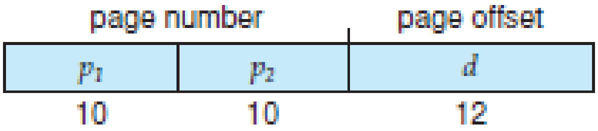
- In this section, we explore some of the most common techniques for structuring the page table, including
  - ✓ Hierarchical paging,
  - ✓ Hashed page tables,
  - ✓ Inverted page tables.

Hierarchical paging

- A page table may consist of up to 1 million entries. Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table
- We would not want to allocate the page table contiguously in main memory.
- One simple solution to this problem is to divide the page table into smaller pieces.
- We can accomplish this division using two-level paging algorithm, in which the page table itself is also paged.



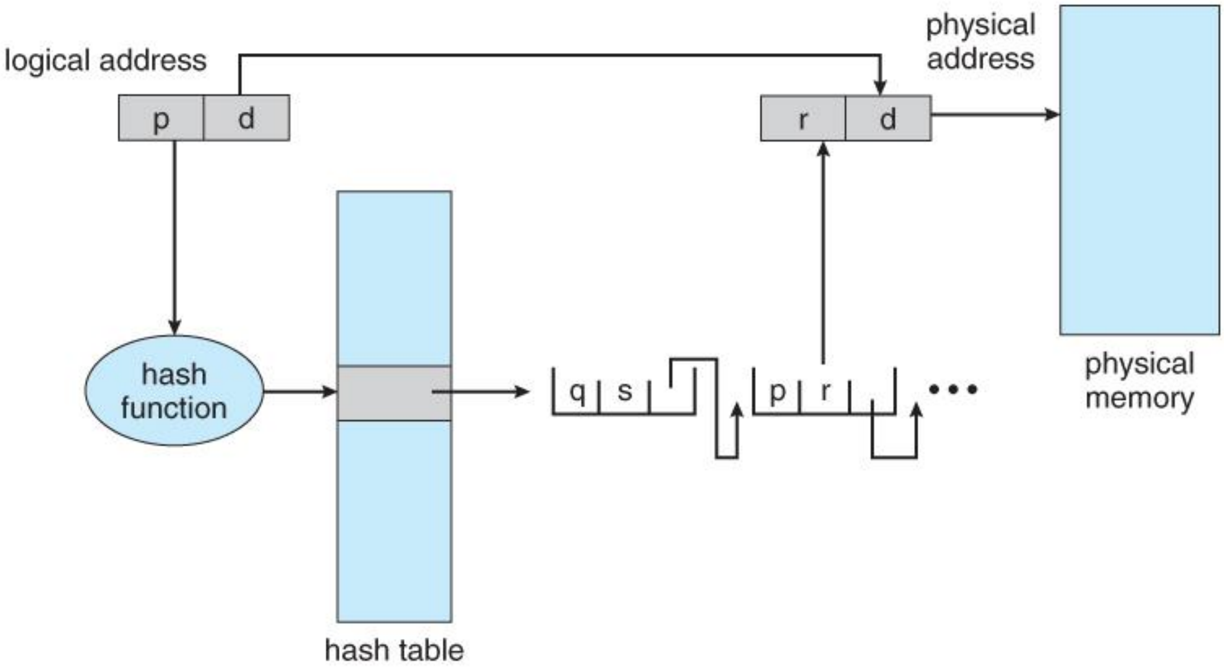
- *Where p*1 is an index into the outer page table and *p*2 is the displacement within the page of the inner page table.
- The address-translation method for this architecture is shown in above Figure.
- Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**.



Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).
- Each element consists of three fields:
  - ✓ the virtual page number,
  - ✓ the value of the mapped page frame,
  - ✓ a pointer to the next element in the linked list.
- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.

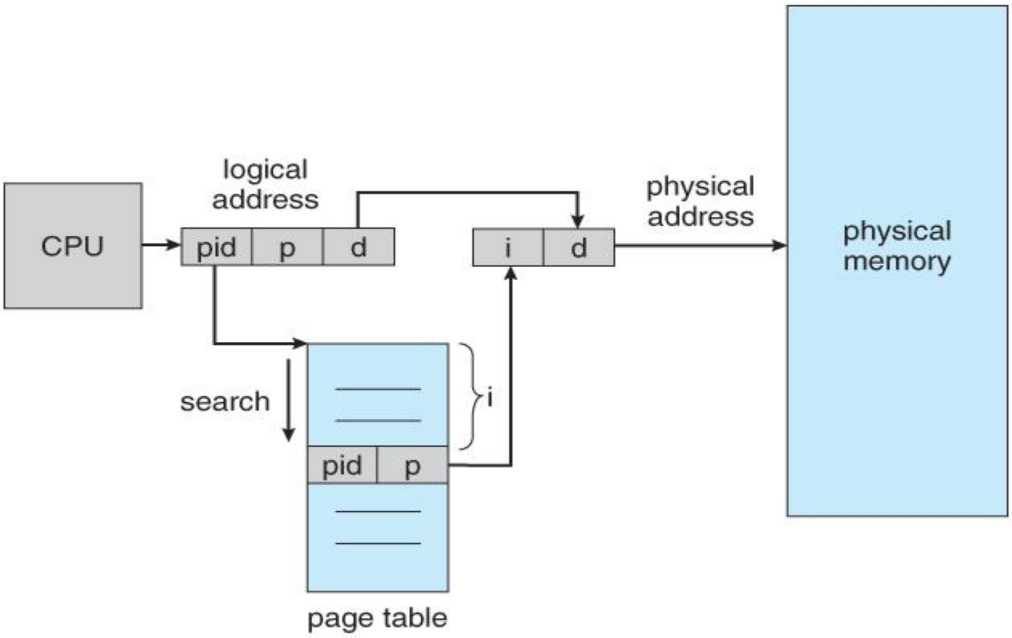
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.
- A variation of this scheme that is useful for 64-bit address spaces has been proposed. This variation uses **clustered page tables.**
- It is similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page.
- Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.



Inverted Page Tables

- An **inverted page table** 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.
- Inverted page tables often require that an address-space identifier be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory.
- Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.

- Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.



- The virtual address in the system consists of a triple:

  <process-id, page-number, offset>

- Each inverted page-table entry is a pair <process-id, page-number> where the process- id assumes the role of the address-space identifier.

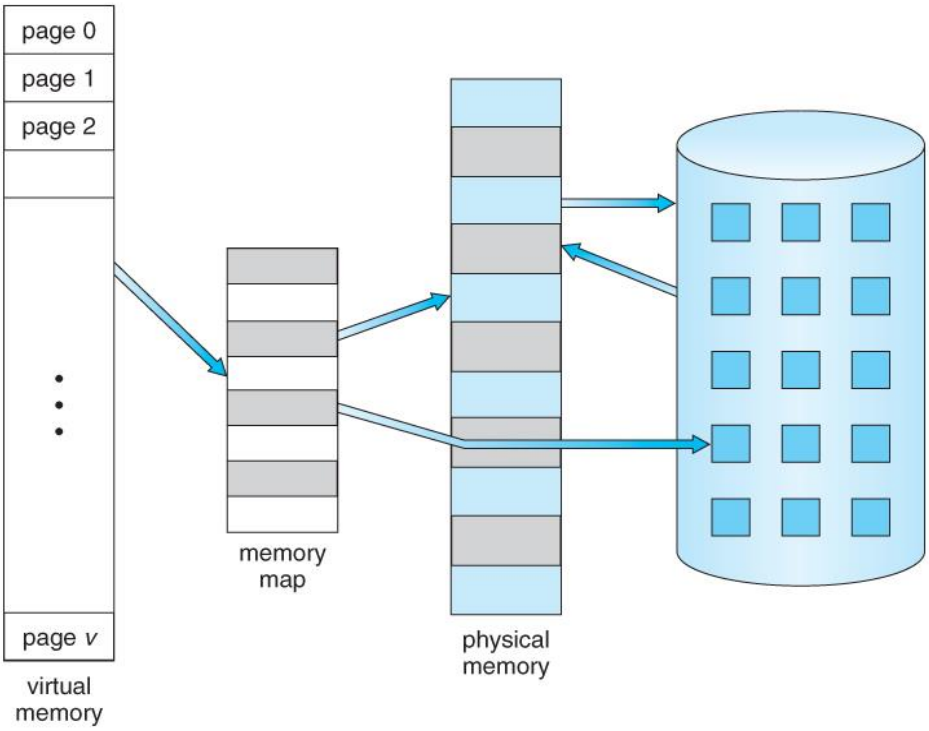
VIRTUAL MEMORY

- A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM.
- In many cases, the entire program is not needed. For instance, consider the following:
  1. Programs often have code to handle unusual error conditions. Since these
  2. errors seldom, if ever, occur in practice, this code is almost never executed.
  3. Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
  4. Certain options and features of a program may be used rarely.
- Even in those cases where the entire program is needed, it may not all be needed at the same time.

<u>Benefits:</u>

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large ***virtual*** address space.

- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput.
- Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

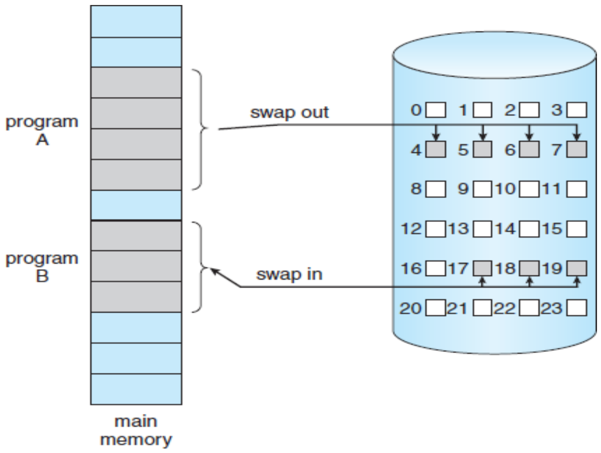


-

DEMAND PAGING

- Consider how an executable program might be loaded from disk into memory.
- One option is to load the entire program in physical memory at program execution time.
- However, a problem with this approach is that we may not initially ***need*** the entire program in memory.
- An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems.
- With demand-paged virtual memory, pages are loaded only when they are demanded during program execution.
- Pages that are never accessed are thus never loaded into physical memory.
- A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk).
- When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, we use a **lazy swapper**.
- A lazy swapper never swaps a page into memory unless that page will be needed.

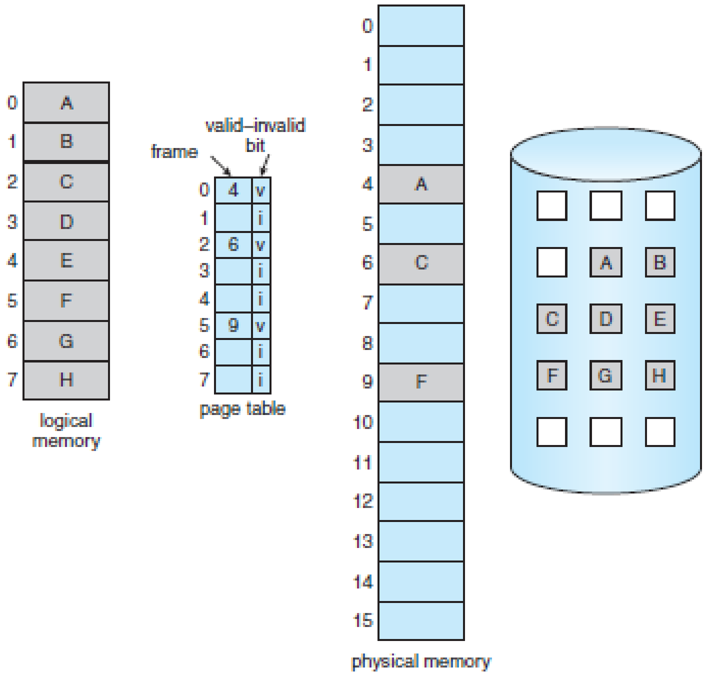
- A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process. We thus use "pager," rather than "swapper," in connection with demand paging.



Basic Concepts

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those pages into memory.
- Thus, it avoids reading into memory, the pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.
- With this scheme, we need some form of support to distinguish between the pages that are in memory and the pages that are on the disk.
- The valid–invalid bit can be used for this purpose. When this bit is set to "valid," the associated page is both legal and in memory.
- If the bit is set to "invalid," the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.

- The page-table entry for a page that is brought into memory is set as usual, 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.



- Access to a page marked invalid causes a **page fault**.
- The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system.
- This trap is the result of the operating system’s failure to bring the desired page into memory.
- The procedure for handling this page fault is as follows;

1. We check an internal table for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

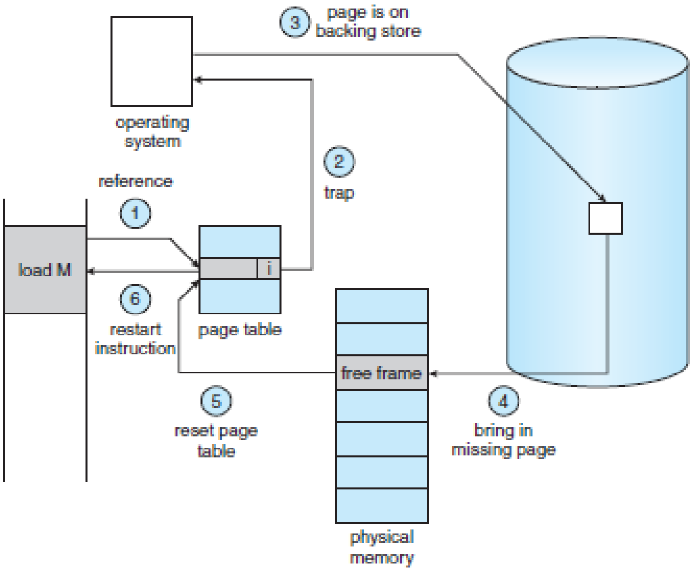
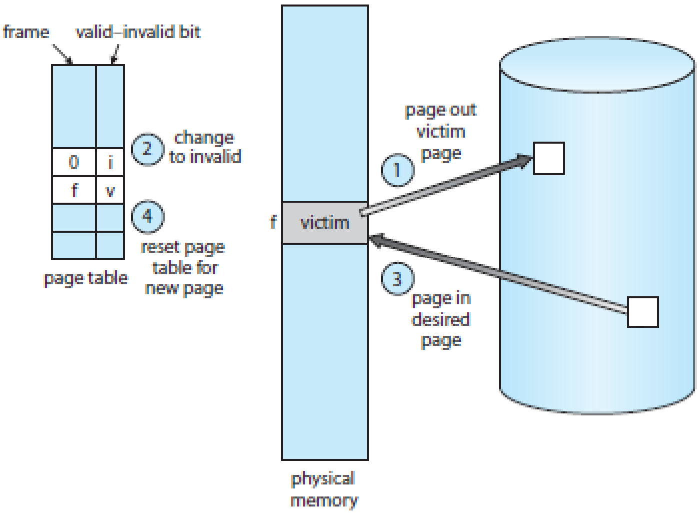


Figure 9.6 Steps in handling a page fault.

PAGE REPLACEMENT

- Page replacement takes the following approach. If no frame is 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 (and all other tables) to indicate that the page is no longer in memory
- We can now use the freed frame to hold the page for which the process faulted.
- We modify the page-fault service routine to include page replacement:



1. Find the location of the desired page on the disk.

2. Find a free frame:

a. If there is a free frame, use it.

b. If there is no free frame, use a page-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 newly freed frame; change the page and frame tables.

4. Continue the user process from where the page fault occurred.

- There are many different page-replacement algorithms.

  1. FIFO Page Replacement

  2. Optimal Page Replacement

  3. LRU Page Replacement

  4. LRU-Approximation Page Replacement

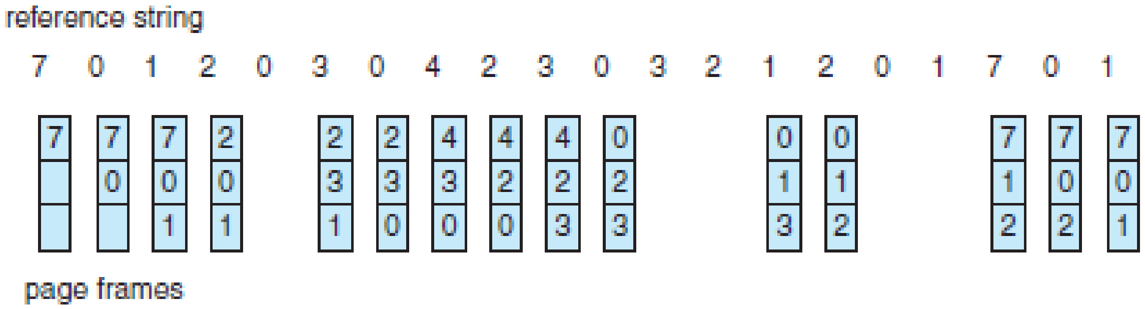
  5. Counting-Based Page Replacement

  6. Page-Buffering Algorithms

FIFO Page Replacement

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- A FIFO replacement algorithm associates with each page the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory.
- We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.
- Ex: Our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames.
- The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
- The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault.
- Page 1 is then replaced by page 0. This process continues as shown in the figure below.

- Every time a fault occurs, we show which pages are in our three frames. There are fifteen faults altogether.

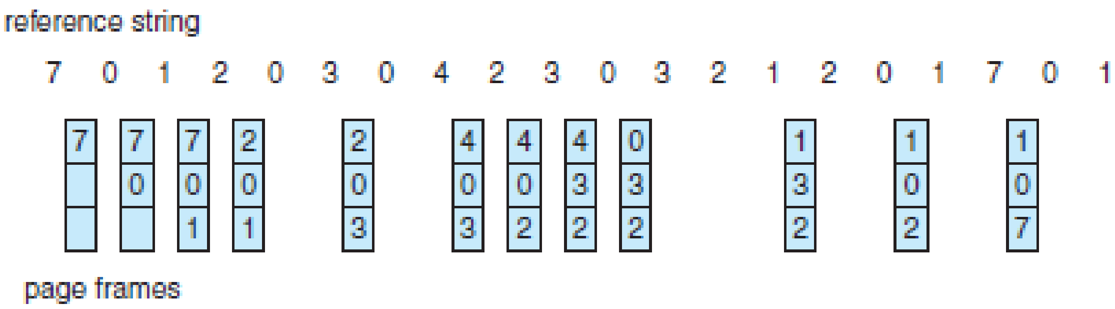


Optimal Page Replacement

- **Optimal page-replacement algorithm** has the lowest page-fault rate of all algorithms. It is simply called as OPT or MIN.
- Replace the page that will not be used for the longest period of time.
- Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.
- The first three references cause faults that fill the three empty frames.
- The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14.
- The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.
- With only nine page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults.

LRU Page Replacement

- Replace the page that ***has not been used*** for the longest period of time. This approach is the **least recently used (LRU) algorithm**.
- 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 LRU algorithm produces twelve faults. Notice that the first five faults are the same as those for optimal replacement.



- When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.
- When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.
- Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen.

LRU-Approximation Page Replacement

- Few computer systems provide sufficient hardware support for true LRU page replacement, in the form of a **reference bit**.
- The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page).
- 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 referenced is set (to 1) by the hardware.
- After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the ***order*** of use.
- This information is the basis for many page-replacement algorithms that approximate LRU replacement.

Counting-Based Page Replacement

- There are many other algorithms that can be used for page replacement.
- We can keep a counter of the number of references that have been made to each page and develop the following two schemes.

1. The **least frequently used (LFU)** page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

- A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again.
- Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.
- One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

2. The **most frequently used (MFU)** page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Page-Buffering Algorithms

- Systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before.
- However, 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, its frame 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.
- It modifies 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 which page 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.
- 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.
- This technique is used in the VAX/VMS system along with a FIFO replacement algorithm.
- When the FIFO replacement algorithm mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame pool.

ALLOCATION OF FRAMES

- Issue in the allocation of frames.
  1. How do we allocate the fixed amount of free memory among the various processes?
  2. If we have 93 free frames and two processes, how many frames does each process get?
- 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.
- Under pure demand paging, all 93 frames would initially be put on the free-frame list.
- When a user process started execution, it would generate a sequence of page faults.
- The first 93-page faults would all get free frames from the free-frame list.
- When the free-frame list was exhausted, 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 terminated, the 93 frames would once again be placed on the free- frame list.

Minimum Number of Frames

- We must 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.
- When a page fault occurs before an executing instruction is complete, the instruction must be restarted.
- Consequently, we must have enough frames to hold all the different pages that any single instruction can reference.
- The minimum number of frames per process is defined by the computer 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, *m*/*n* frames.
- For instance, 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, strictly speaking, wasted.
- To solve this problem, we can use **proportional allocation**, in which we allocate available memory to each process according to its size.
- With either equal or proportional allocation, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes.

Global versus Local Allocation

- With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement**.
- Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process.
- Local replacement requires that each process select from only its own set of allocated frames.

THRASHING

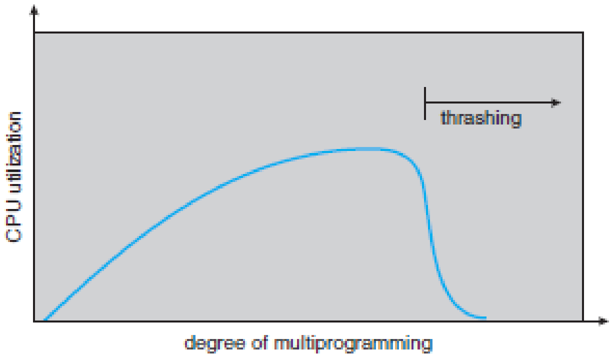
- If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution.
- We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.
- In fact, look at any process that does not have "enough" frames.
- If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault.
- At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away.
- Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately. This high paging activity is called thrashing.
- A process is thrashing if it is spending more time paging than executing.

Cause of Thrashing

- Thrashing results in severe performance problems.
- The operating system monitors CPU utilization. If CPU utilization is too low, we increase the

degree of multiprogramming by introducing a new process to the system.

- A page-replacement algorithm is used; it replaces pages without regard to the process to which they belong.
- Now suppose that a process enters a new phase in its execution and needs more frames.
- It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes.
- As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.
- As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached.
- If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply.
- At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.



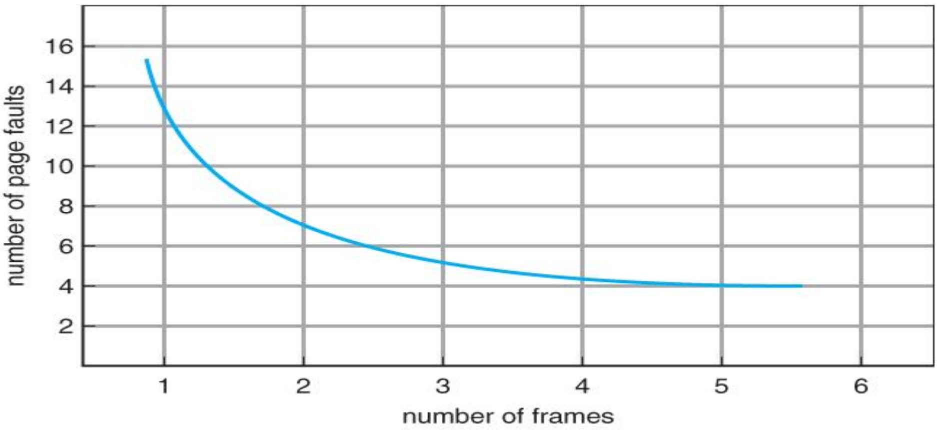
- We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm).
- With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.

Working-set Model:

- To prevent thrashing, we must provide a process with 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 model** 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 together
- A program is generally composed of several different localities, which may overlap.

Page-Fault Frequency

- To control thrashing a new strategy called page**-fault frequency (PFF)** is used.
- Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate.
- When it is too high, we know that the process needs more frames; if the page-faultrate is too low, then the process may have too many frames.
- We can establish upper and lower bounds on the desired page-fault rate. 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.



MEMORY MAPPED FILES

- Consider a sequential read of a file on disk using the standard system calls open(), read(), and write().

- Each file access requires a system call and disk access.

- Memory mapping a file, allows a part of the virtual address space to be logically associated with the file.

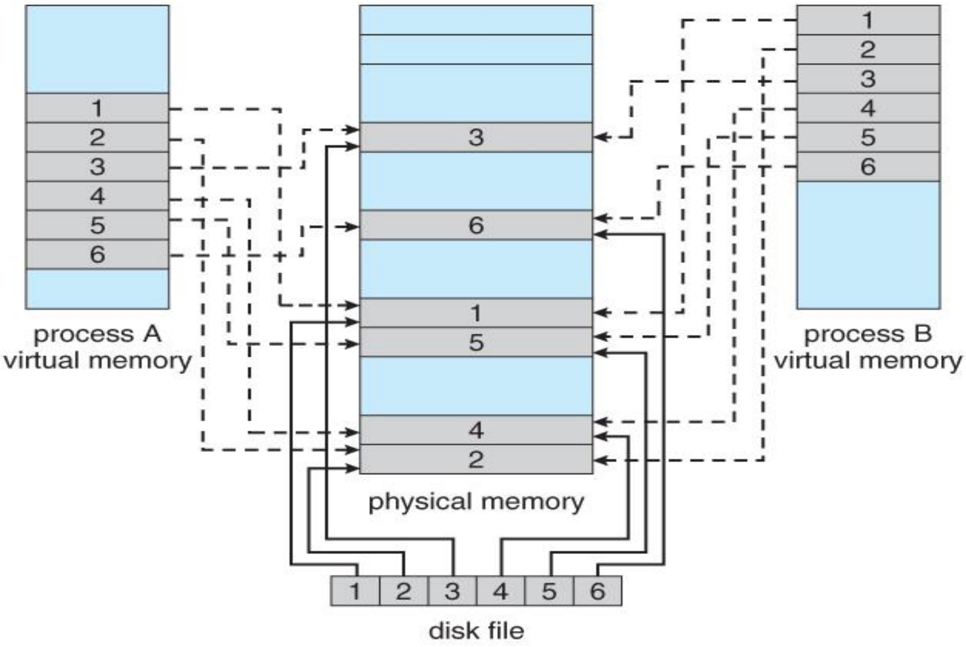
Basic Mechanism

- Memory mapping a file is accomplished by mapping a disk block to a page (or pages) in memory. Initial access to the file proceeds through ordinary demand paging, resulting in a page fault.

- However, a page-sized portion of the file is read from the file system into a physical page.

- Subsequent reads and writes to the file are handled as routine memory accesses.

- Manipulating files through memory rather than incurring the overhead of using the read() and write() system calls simplifies and speeds up file access and usage.
- Some systems may choose to update the physical file when the operating system periodically checks whether the page in memory has been modified.
- When the file is closed, all the memory-mapped data are written back to disk and removed from the virtual memory of the process.
- Some operating systems provide memory mapping only through a specific system call and use the standard system calls to perform all other file I/O.
- However, some systems choose to memory-map a file regardless of whether the file was specified as memory-mapped.
- In Solaris a file is specified as memory-mapped using the mmap() system call.
- Solaris maps the file into the address space of the process. If a file is opened and accessed using ordinary system calls, such as open(), read(), and write(),



UNIT: V

STORAGE MANAGEMENT

- Main memory is usually too small to accommodate all the data and programs permanently.
- The computer system must provide secondary storage to back up main memory.
- Modern computer systems use disks as the primary on-line storage medium for information (both programs and data).
- The file system provides the mechanism for on-line storage of and access to both data and programs residing on the disks.
- A file is a collection of related information defined by its creator.
- The files are mapped by the operating system onto physical devices.
- Files are normally organized into directories for ease of use.

DISK STRUCTURE

- Modern magnetic disk drives are addressed as large one-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer.
- The size of a logical block is usually 512 bytes, although some disks can be low-level formatted to have a different logical block size, such as 1,024 bytes.
- The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially.
- Sector 0 is the first sector of the first track on the outermost cylinder.
- The mapping proceeds in order through that track, then through 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.
- In practice, it is difficult to perform this translation, for two reasons.
- First, most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk.
- Second, the number of sectors per track is not a constant on some drives.
- On media that use **constant 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.
- Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone.
- 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; in this case, 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 **constant angular velocity (CAV)**.
- The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track.
- Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.

DISK SCHEDULING

- One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth.
- For magnetic disks, the access time has two major components,
- The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector.
- The rotational latency is the additional time for the disk to rotate the desired sector to the disk head.
- The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

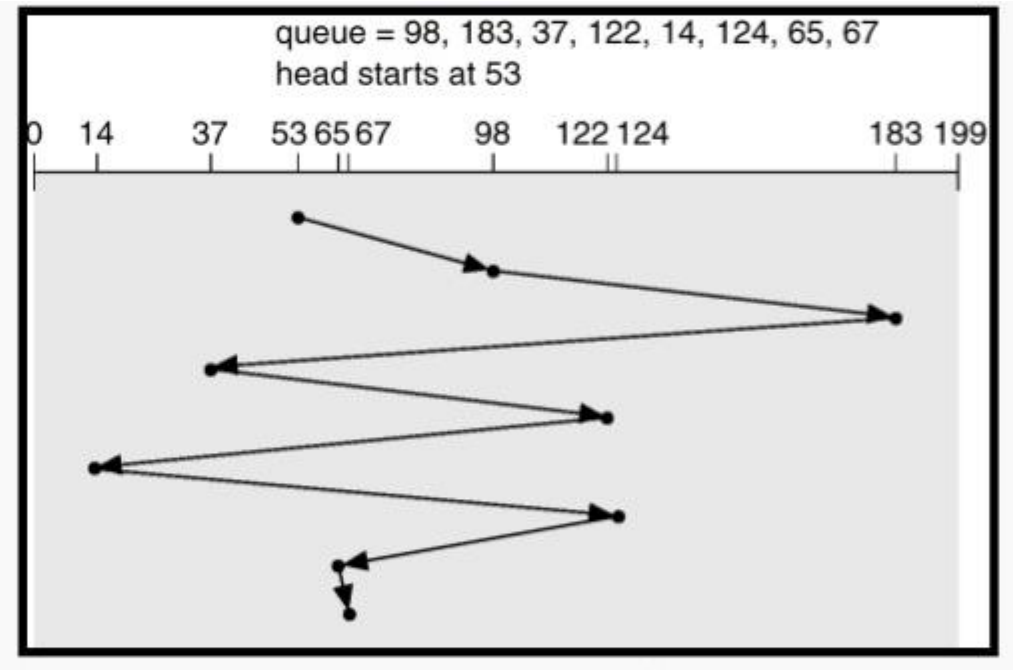
- We can improve both the access time and the bandwidth by managing the order in which disk I/O requests are serviced.
- Whenever a process needs I/O to or from the disk, it issues a system call to the operating system.
    - ✓ The request specifies several pieces of information:
    - ✓ Whether this 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 drive and controller are available, the request can be serviced immediately.
- If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive.
- For a multiprogramming system with many processes, the disk queue may often have several pending requests.
- Thus, when one request is completed, the operating system chooses which pending request to service next.
- The disk-scheduling algorithms can be used, and we discuss them next.
    - ✓ FCFS Scheduling
    - ✓ SSTF Scheduling
    - ✓ SCAN Scheduling
    - ✓ C-SCAN Scheduling
    - ✓ LOOK Scheduling

FCFS Scheduling

- The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm.
- This algorithm is intrinsically fair, but it generally does not provide the fastest service.
- Consider, for example, a disk queue with requests for I/O to blocks on cylinders

98, 183, 37, 122, 14, 124, 65, 67,

- If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders.
- The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule.
- If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

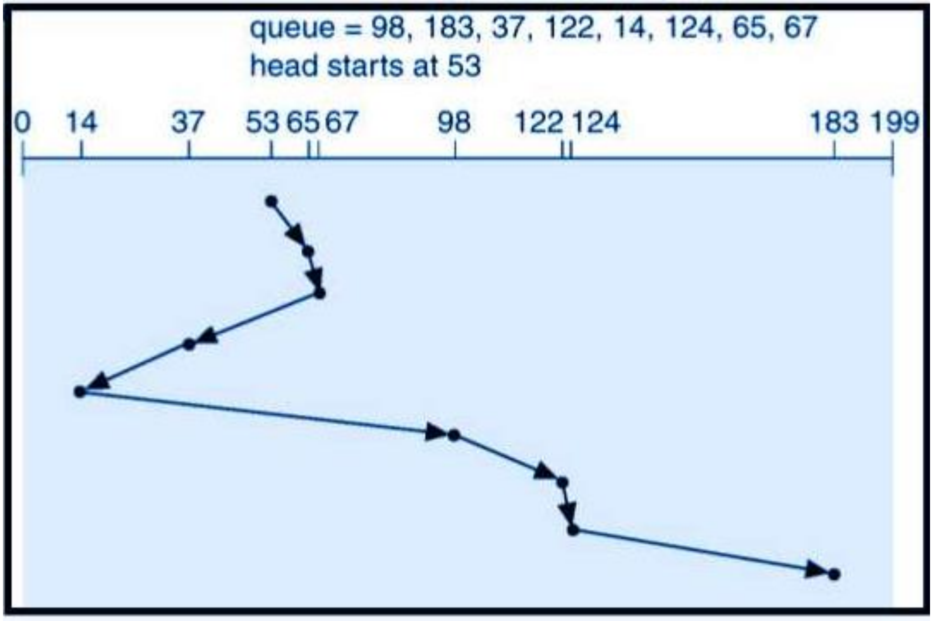


SSTF Scheduling

- It seems reasonable to service all the requests close to the current head position before moving the head far away to service other requests. This assumption is the basis for the **shortest-seek-time-first (SSTF) algorithm**.
- The SSTF algorithm selects the request with the least seek time from the current head position.
- In other words, SSTF chooses the pending request closest to the current head position.
- For our example request queue, the closest request to the initial head position (53) is at cylinder 65.
- Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next.
- Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183. This scheduling method results in a total head movement of only 236 cylinders little more than one-third of the distance needed for FCFS scheduling of this request queue. Clearly, this algorithm gives a substantial improvement in performance.
- SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling; and like SJF scheduling, it may cause starvation of some requests.
- Remember that requests may arrive at any time. Suppose that we have two requests in the queue, for cylinders 14 and 186, and while the request from 14 is being serviced, a

  new request near 14 arrives.
- This new request will be serviced next, making the request at 186 waits. While this request is being serviced, another request close to 14 could arrive.

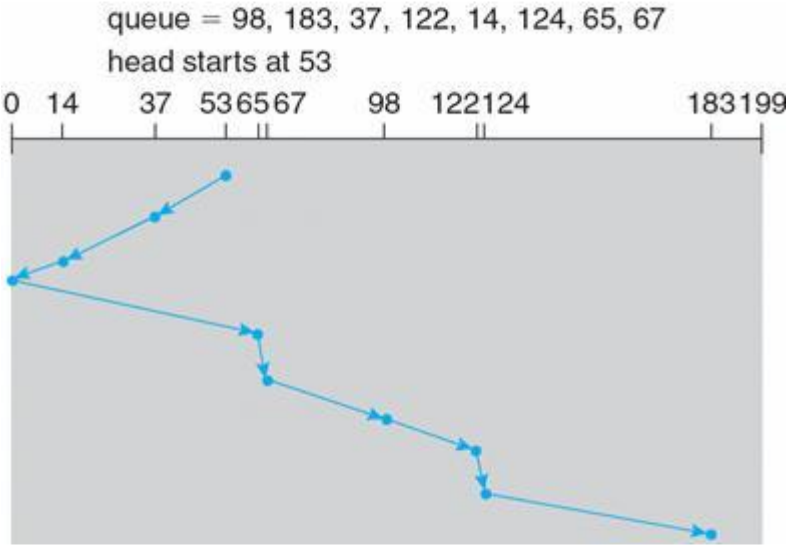
- In theory, a continual stream of requests near one another could cause the request for cylinder 186 to wait indefinitely.
- This scenario becomes increasingly likely as the pending-request queue grows longer.
- Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal.
- In the example, we can do better by moving the head from 53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183.
- This strategy reduces the total head movement to 208 cylinders.



SCAN Scheduling

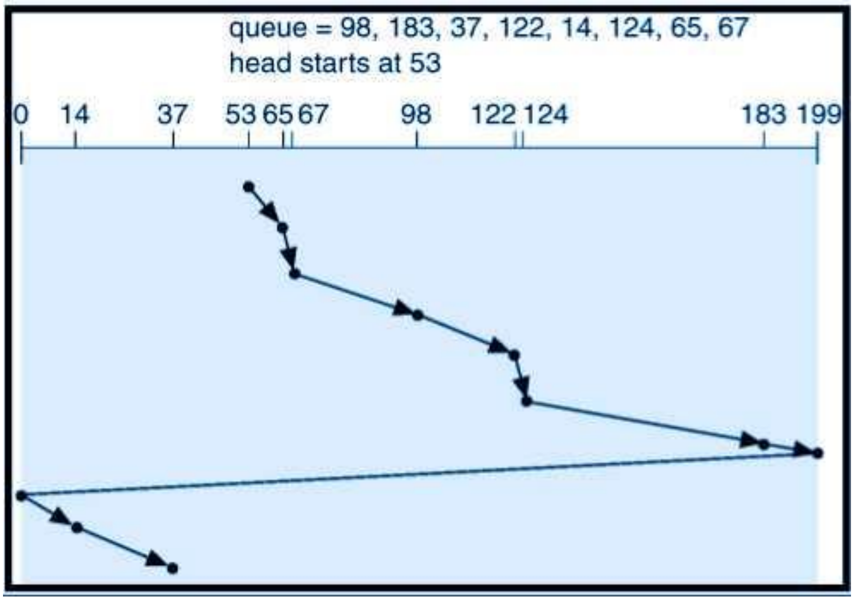
- In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.
- At the other end, the direction of head movement is reversed, and servicing continues.
- The head continuously scans back and forth across the disk.
- The SCAN algorithm is sometimes called the **elevator algorithm**, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.
- Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position.
- Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14.
- At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183.

- If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.
- Assuming a uniform distribution of requests for cylinders, consider the density of requests when the head reaches one end and reverses direction.
- At this point, relatively few requests are immediately in front of the head, since these cylinders have recently been serviced.
- The heaviest density of requests is at the other end of the disk. These requests have also waited the longest.



C-SCAN Scheduling

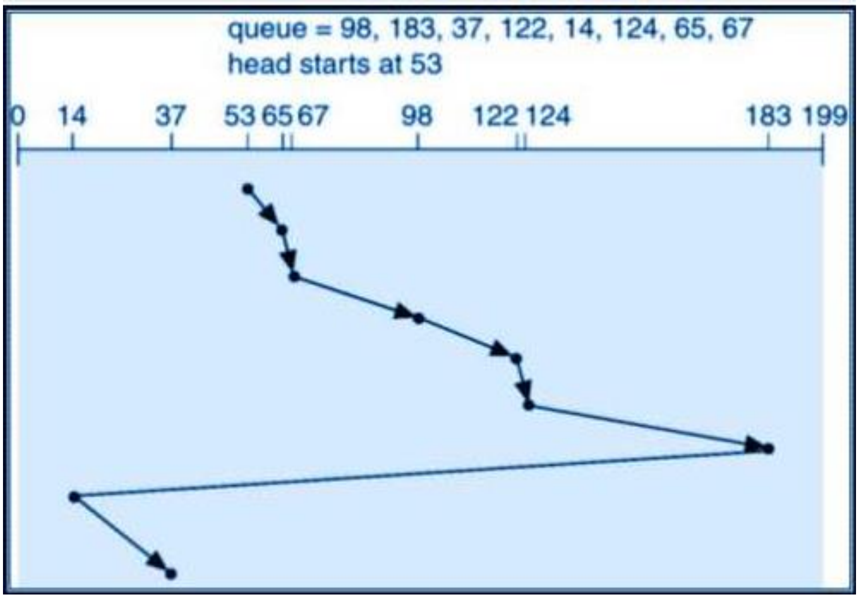
- **Circular SCAN (C-SCAN) scheduling** is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way.



- When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.
- The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

LOOK Scheduling

- Both SCAN and C-SCAN move the disk arm across the full width of the disk.
- More commonly, 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.
- Versions of SCAN and C-SCAN that follow this pattern are called **LOOK** and **C- LOOK scheduling**, because they ***look*** for a request before continuing to move in a given direction.



DISK MANAGEMENT

- The operating system is responsible for several other aspects of disk management like,

Disk Formatting

- ✓ Disk Formatting
- ✓ Booting from disk
- ✓ Bad-block recovery.

- A new magnetic disk is a blank slate: it is just a platter of a magnetic recording material.
- Before a disk can store data, it must be divided into sectors that the disk controller 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. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), 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).
- When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area.
- When the sector is read, the ECC is recalculated and compared with the stored value.
- If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad
- Before using a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps.
- The first step is to **partition** the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk.
- The second step is **logical formatting**, or creation of a file system. In this step, the operating system stores the initial file-system data structures onto the disk.
- These data structures may include maps of free and allocated space and an initial empty directory.

Boot Block

- For a computer to start running, when it is powered up or rebooted it must have an initial program to run called a **bootstrap** program.
- It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system.
- To do its job, the bootstrap program finds the operating-system kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution.

- The bootstrap is stored in **read-only memory (ROM)**. This location is convenient, because ROM needs no initialization and is at a fixed location that the processor can start executing when powered up or reset.
- ROM is read only; it cannot be infected by a computer virus.
- The problem is that changing this bootstrap code requires changing the ROM hardware chips.
- For this reason, most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk.
- The full bootstrap program can be changed easily: a new version is simply written onto the disk.
- The full bootstrap program is stored in the "boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a **boot disk** or **system disk**.
- The code in the boot ROM instructs the disk controller to read the boot blocks into memory and then starts executing that code.
- The full bootstrap program is more sophisticated than the bootstrap loader in the boot ROM.
- It is able to load the entire operating system from a non-fixed location on disk and to start the operating system running.

Bad Blocks

- Because disks have moving parts and small tolerances, they are prone to failure.
- Sometimes the failure is complete; in this case, the disk needs to be replaced and its contents restored from backup media to the new disk.
- More frequently, one or more sectors become defective. Most disks even come from the factory with **bad blocks**.
- Depending on the disk and controller in use, these blocks are handled in a variety of ways.
- One strategy is to scan the disk to find bad blocks while the disk is being formatted. Any bad blocks that are discovered are flagged as unusable so that the file system does not allocate them.
- If blocks go bad during normal operation, a special program must be run manually to search for the bad blocks and to lock them away.
- Data that resided on the bad blocks usually are lost.
- 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.
- Low-level formatting also sets aside spare sectors not visible to the operating system.
- 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**.
- An alternative to sector sparing, some controllers can be instructed to replace a bad block by

sector slipping.

- Soft errors may trigger a process in which a copy of the block data is made and the block is spared or slipped.
- An unrecoverable **hard error**, results in lost data.

SWAP-SPACE MANAGEMENT

- Swapping is moving entire processes between disk and main memory.
- Swapping in that setting occurs when the amount of physical memory reaches a critically low point and processes are moved from memory to swap space to free available memory.
- 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.
  - ✓ How swap space is used?
  - ✓ Where swap space is located on disk?
  - ✓ How swap space is managed?

<u>Swap-Space Use</u>

- Swap space is used in various ways by different operating systems, depending on the memory-management algorithms in use.
- Systems that implement swapping may use swap space to hold an entire process image, including the code and data segments.
- The amount of swap space needed on a system can therefore vary from a few megabytes of disk space to gigabytes, depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used.
- If a system runs out of swap space it may be forced to abort processes or may crash entirely.
- Overestimation wastes disk space that could otherwise be used for files, but it does no other harm.

<u>Swap-Space Location</u>

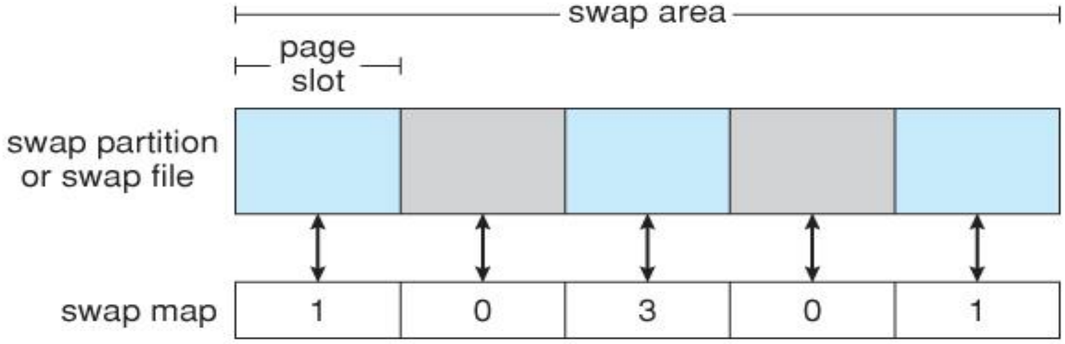
- A swap space can reside in one of two places:
  - ✓ Normal file system
  - ✓ Separate disk partition – Raw Partition
- If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space.
- Navigating the directory structure and the disk allocation data structures takes time and

extra disk accesses.

- Swap space can be created in a separate raw partition. No file system or directory structure is placed in this space.
- A separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition.
- This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems.
- The raw-partition approach creates a fixed amount of swap space during disk partitioning.

Swap-Space Management

- When a process executes, text-segment pages containing code are brought in from the file system, accessed in main memory, and thrown away if selected for pageout.
- 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.
- 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 on modern computers, which have more physical memory than older systems and tend to page less.



- A swap area may be in either a swap file on a regular file system or a dedicated swap partition.
- Each swap area consists of a series of 4-KB page slots, which are used to hold swapped pages.
- Associated with each swap area is a swap map—an array of integer counters, each corresponding to a page slot in the 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.

RAID STRUCTURE

- Disk drives have continued to get smaller and cheaper, so it is now economically feasible to attach many disks to a computer system.
- Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel.
- Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data.
- A variety of disk-organization techniques, collectively called redundant arrays of independent disks (RAID), are commonly used to address the performance and reliability issues.
- In the past, RAIDs composed of small, cheap disks were viewed as a cost-effective alternative to large, expensive disks.
- Today, RAIDs are used for their higher reliability and higher data-transfer rate, rather than for economic reasons.

Improvement of Reliability via Redundancy

- The chance that some disk out of a set of *N* disks will fail is much higher than the chance that a specific single disk will fail.
- The solution to the problem of reliability is to introduce **redundancy**; we store extra information that is not normally needed but that can be used in the event of failure of a disk to rebuild the lost information.
- Thus, even if a disk fails, data are not lost.
- The simplest approach to introducing redundancy is to duplicate every disk. This technique is called **mirroring**.
- With mirroring, a logical disk consists of two physical disks, and every write iscarried out on both disks. The result is called a **mirrored volume**.
- If one of the disks in the volume fails, the data can be read from the other.
- Data will be lost only if the second disk fails before the first failed disk is replaced.

Improvement in Performance via Parallelism

- Parallel access to multiple disks improves performance.
- With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk.
- The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.
- With multiple disks, we can improve the transfer rate as well by striping data across the disks.

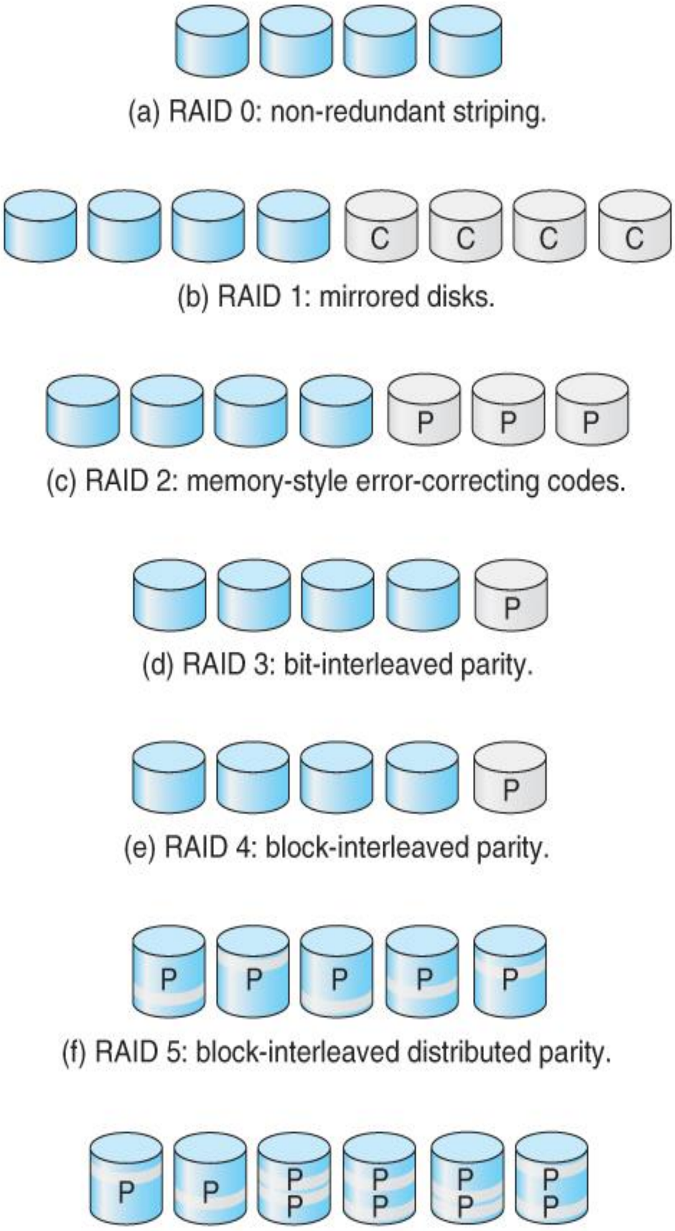
- In its simplest form, **data striping** consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**.
- Parallelism in a disk system, as achieved through striping, has two main goals:

RAID Levels

1. Increase the throughput of multiple small accesses by load balancing.

2. Reduce the response time of large accesses.

- Mirroring provides high reliability, but it is expensive.
- Striping provides high data-transfer rates, but it does not improve reliability.
- Numerous schemes to provide redundancy at lower cost by using disk striping combined with "parity" bits have been proposed.
- These schemes have different cost–performance trade-offs and are classified according to levels called **RAID levels**.
- *P* indicates error-correcting bits and *C* indicates a second copy of the data.
- Four disks' worth of data are stored, and the extra disks are used to store redundant information for failure recovery.
- **RAID level 0 -** refers to disk arrays with striping at the level of blocks but without any redundancy
    - ✓ **RAID level 1 -** refers to disk mirroring.
    - ✓ **RAID level 2 -** is also known as memory-style error- correcting code (ECC) organization.
    - ✓ **RAID level 3 –**also called bit-interleaved parity organization, improves on level 2 by taking into account the fact that, unlike memory systems, disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction as well as for detection.
    - ✓ **RAID level 4 –** also called block-interleaved parity organization, uses block-level striping, as in RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from *N* other disks.
    - ✓ **RAID level 5** – also called block-interleaved distributed parity, differs from level 4 in that it spreads data and parity among all *N*+1 disks, rather than storing data in *N* disks and parity in one disk.
    - ✓ **RAID level 6** - also called the **P + Q redundancy scheme**, is much like RAID level 5 but stores extra redundant information to guard against multiple disk failures.



(a) RAID 0: non-redundant striping.

(b) RAID 1: mirrored disks.

(c) RAID 2: memory-style error-correcting codes.

(d) RAID 3: bit-interleaved parity.

(e) RAID 4: block-interleaved parity.

(f) RAID 5: block-interleaved distributed parity.

(g) RAID 6: P + Q redundancy.

FILE SYSTEM INTERFACE: FILE CONCEPT

- A file is a collection of related information that is recorded on secondary storage.
- Files represent programs and data. Data files may be numeric, alphabetic, alphanumeric, or binary.
- In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file’s creator and user. The information in a file is defined by its creator.
- Many different types of information may be stored in a file source or executable programs, numeric or text data, photos, music, video, and so on.

- A file has a certain defined structure, which depends on its type. A text file is a sequence of characters organized into lines.
- A source file is a sequence of functions, each of which is further organized as declarations followed by executable statements.
- An executable file is a series of code sections that the loader can bring into memory and execute.

File Attributes

- A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters.
- Some systems differentiate between uppercase and lowercase characters in names.
- The file's owner might write the file to a USB disk, send it as an e-mail attachment, or copy it across a network, and it could still be called using the same name on the destination system.
- A file's attributes vary from one operating system to another but typically consist of these:
  - ✓ **Name.** The symbolic file name is the only information kept in human readable form.
  - ✓ **Identifier.** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
  - ✓ **Type.** This information is needed for systems that support different types of files.
  - ✓ **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 (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
  - ✓ **Protection**. Access-control information determines who can do reading, writing, executing, and so on.
  - ✓ **Time, date, and user identification**. This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

File Operations

- The operating system can provide system calls to create, write, read, reposition, delete, and truncate files and renaming a file.

  1. Creating a file: Two steps are necessary to create a file.

  - ▪ First, space in the file system must be found for the file.
  - ▪ Second, an entry for the new file must be made in the directory.

  2. Writing a file: To write a file, we make a system call specifying both the name of the file and the information to be written to the file.

3. Reading a file: To read from a file, we use a system call that specifies the name of the file and where the next block of the file should be put.

4. Repositioning within a file: The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value.

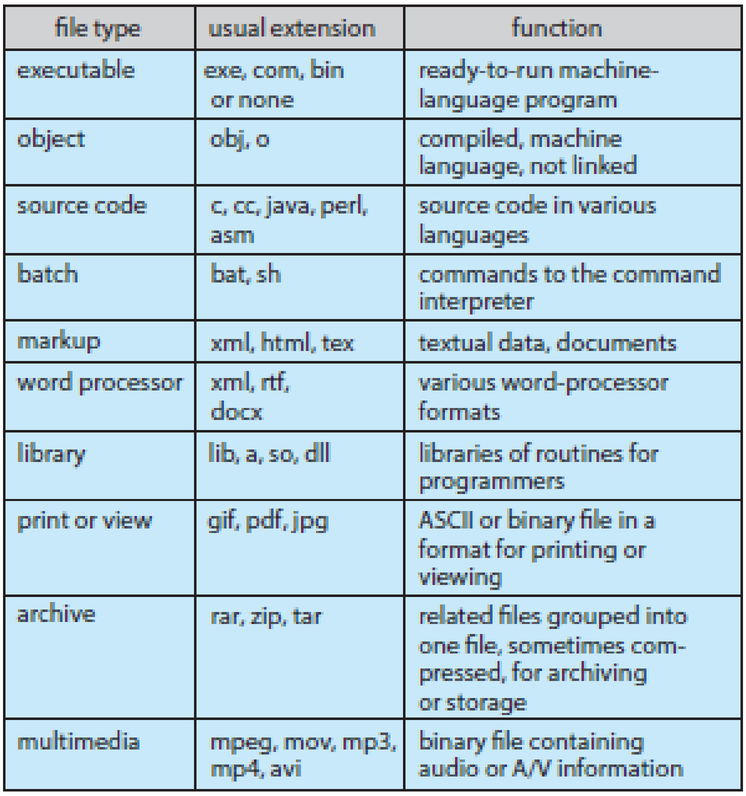
5. Deleting a file: To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.

6. Truncating a file: The user may want to erase the contents of a file but keep its attributes.

7. Renaming a file: The name of the file can be changed whenever needed.

File Types

- The name is split into two parts a name and an extension, usually separated by a period.
- The user and the operating system can tell from the name alone what the type of a file is.



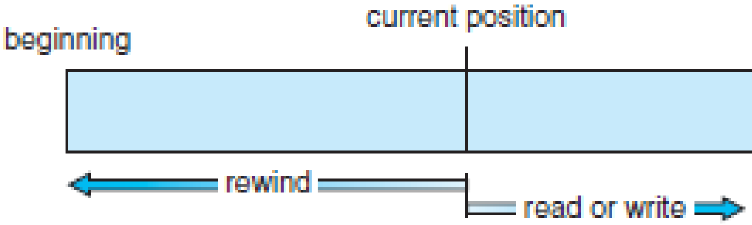
file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

ACCESS METHODS

- Files store information. When it is used, this information must be accessed and read into computer memory.
- The information in the file can be accessed in several ways like:

- ✓ **Sequential Access**
- ✓ **Direct Access**
- ✓ **Indexed Access** **Sequential Access**

- The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the most common.
- Editors and compilers usually access files in this fashion.



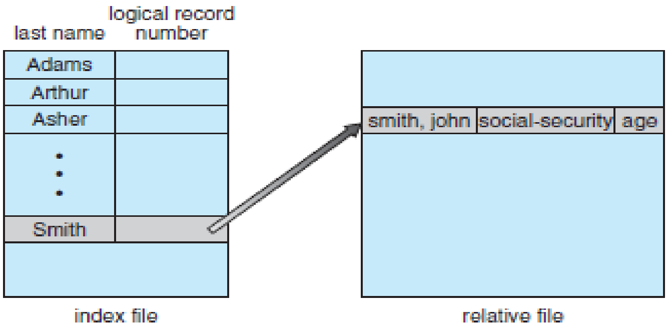
- A read operation read_next() reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
- The write operation write_next() appends to the end of the file and advances to the end of the newly written material (the new end of file).
- Such a file can be reset to the beginning, and on some systems, a program may be able to skip forward or backward *n* records.

Direct Access

- Another method is direct access (or relative access). Here, a file is made up of fixed- length logical records that allow programs to read and write records rapidly in no particular order.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- 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.
- There are no restrictions on the order of reading or writing for a direct-access file.

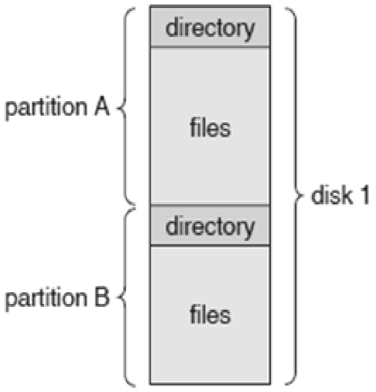
Index Access

- These methods generally involve the construction of an index for the file.
- The index, like an index in the back of a book, contains pointers to the various blocks.
- To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.



DIRECTORY AND DISK STRUCTURE

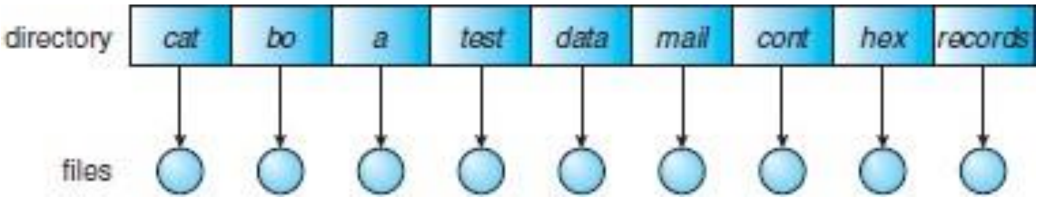
- Next, we consider how to store files. Certainly, no general-purpose computer stores just one file. There are typically thousands, millions, even billions of files within a computer.
- Files are stored on random-access storage devices, including hard disks, optical disks, and solid-state (memory-based) disks.
- A storage device can be used in its entirety for a file system. For example, a disk can be **partitioned** into quarters, and each quarter can hold a separate file system.
- Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device, or leaving part of the device available for other uses.
- A file system can be created on each of these parts of the disk.
- The information about the files in the system is kept in entries in a **device directory** or **volume table of contents**.
- The device directory (more commonly known simply as the **directory**) records information such as name, location, size, and type for all files on that volume.



Single-Level Directory

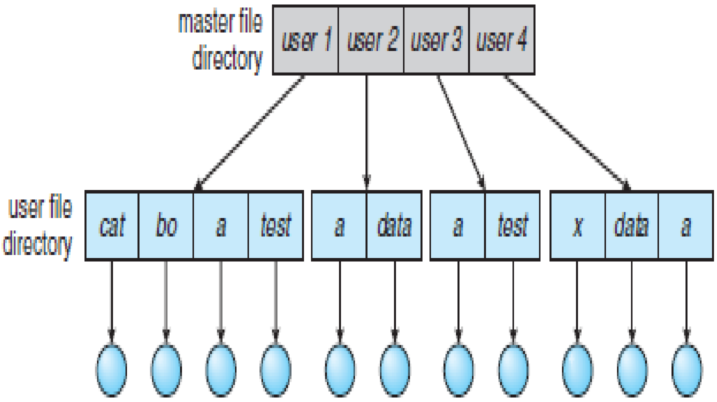
- The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand.
- A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user.

- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.
- It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system.
- Keeping track of so many files is a daunting task.



Two-Level Directory

- In the two-level directory structure, each user has his own **user file directory (UFD)**.
- The UFDs have similar structures, but each lists only the files of a single user.
- When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user.
- When a user refers to a particular file, only his own UFD is searched.
- Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.
- To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists.
- To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

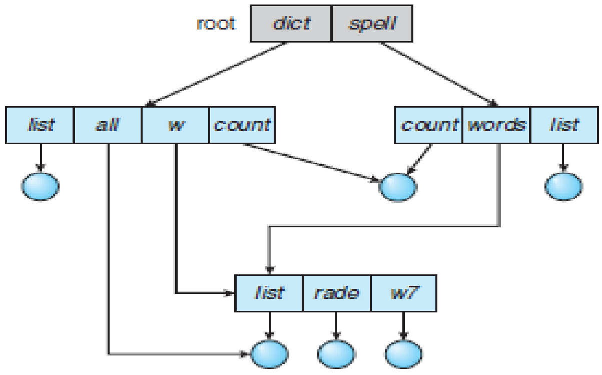


Tree-Structured Directories

- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.
- A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way.
- All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).
- Special system calls are used to create and delete directories.
- In normal use, each process has a current directory.
- The **current directory** should contain most of the files that are of current interest to the process.
- When reference is made to a file, the current directory is searched.
- If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.
- Path names can be of two types: absolute and relative.
- An **absolute path name** begins at the root and follows a path down to the specified file, giving the directory names on the path.
- A **relative path name** defines a path from the current directory.

Acyclic-Graph Directories

- A tree structure prohibits the sharing of files or directories.
- An **acyclic graph** is, a graph with no cycles allows directories to share subdirectories and files.
- The same file or subdirectory may be in two different directories.
- The acyclic graph is a natural generalization of the tree-structured directory scheme.


root
dict
spell
list
all
w
count
count
words
list
list
rade
w7