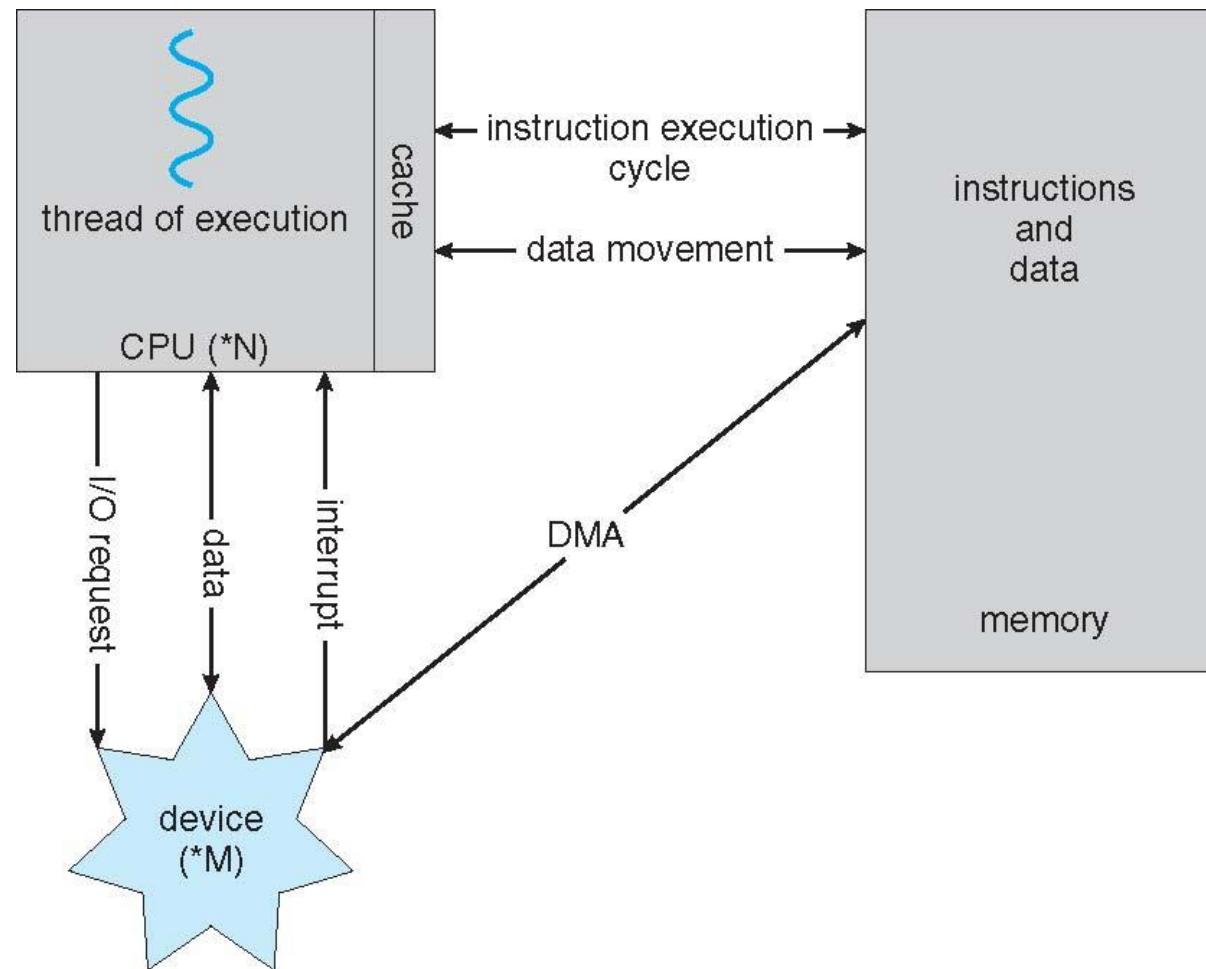


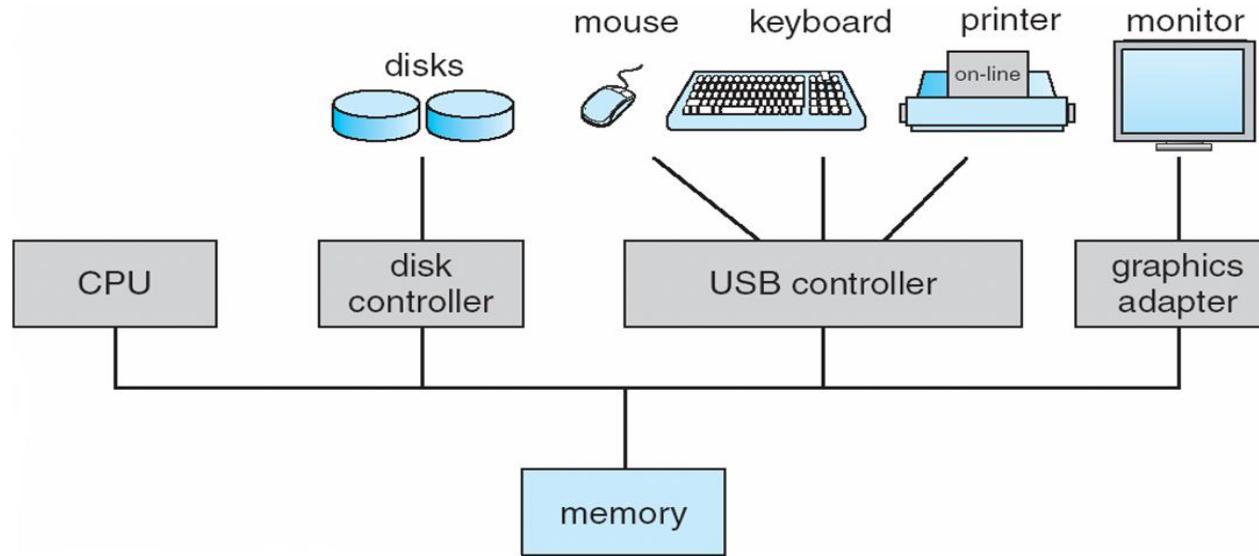
# OPERATING SYSTEMS

- A modern computer consists of one or more processors, main memory, storage devices, network interfaces and various input/output devices.
- Writing programs that keep track of all these components is an extremely difficult job.

# How a Modern Computer Works



# Computer System Organization



# Computer-System Operation

- 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*

# Components of Computer System

- **Hardware**
- **System Software**
- **Users**

## **Hardware**

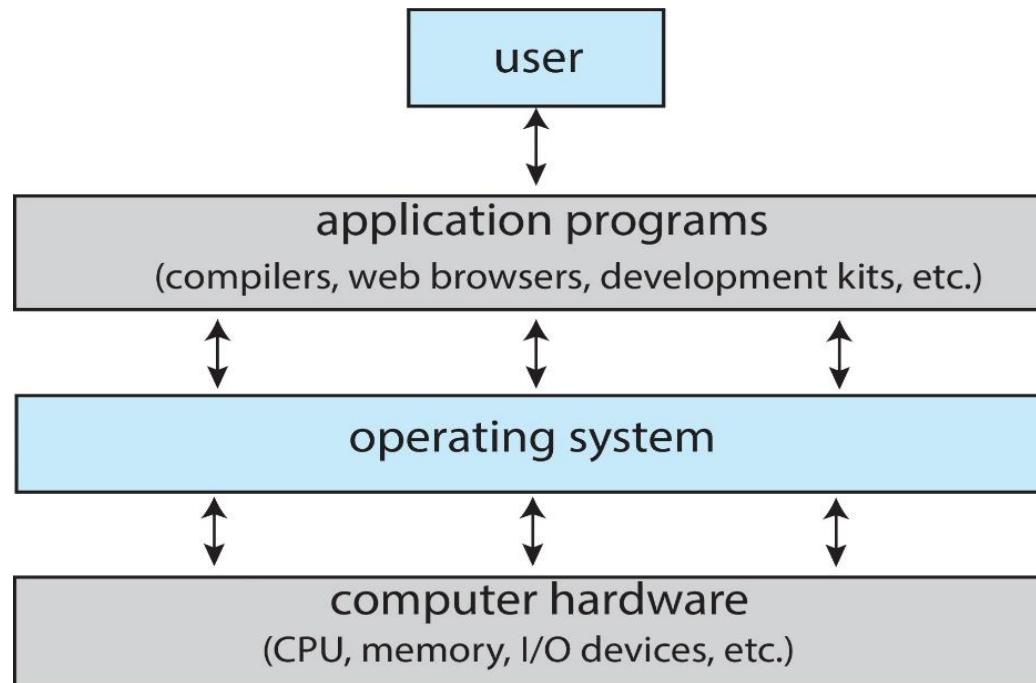
- It provides basic computing resources
  - CPU, memory, I/O devices
- It may be composed of two or more levels.
  - The lowest level consists of the basic resources.
  - The next level is the micro architecture level where physical resources are grouped together as functional units. The basic level operations involve internal registers, CPU and the operation of the data path is controlled either by micro program or by hardware control unit.
  - The next level is the Instruction set architecture level which deals with the execution of instructions.

## **System Software:**

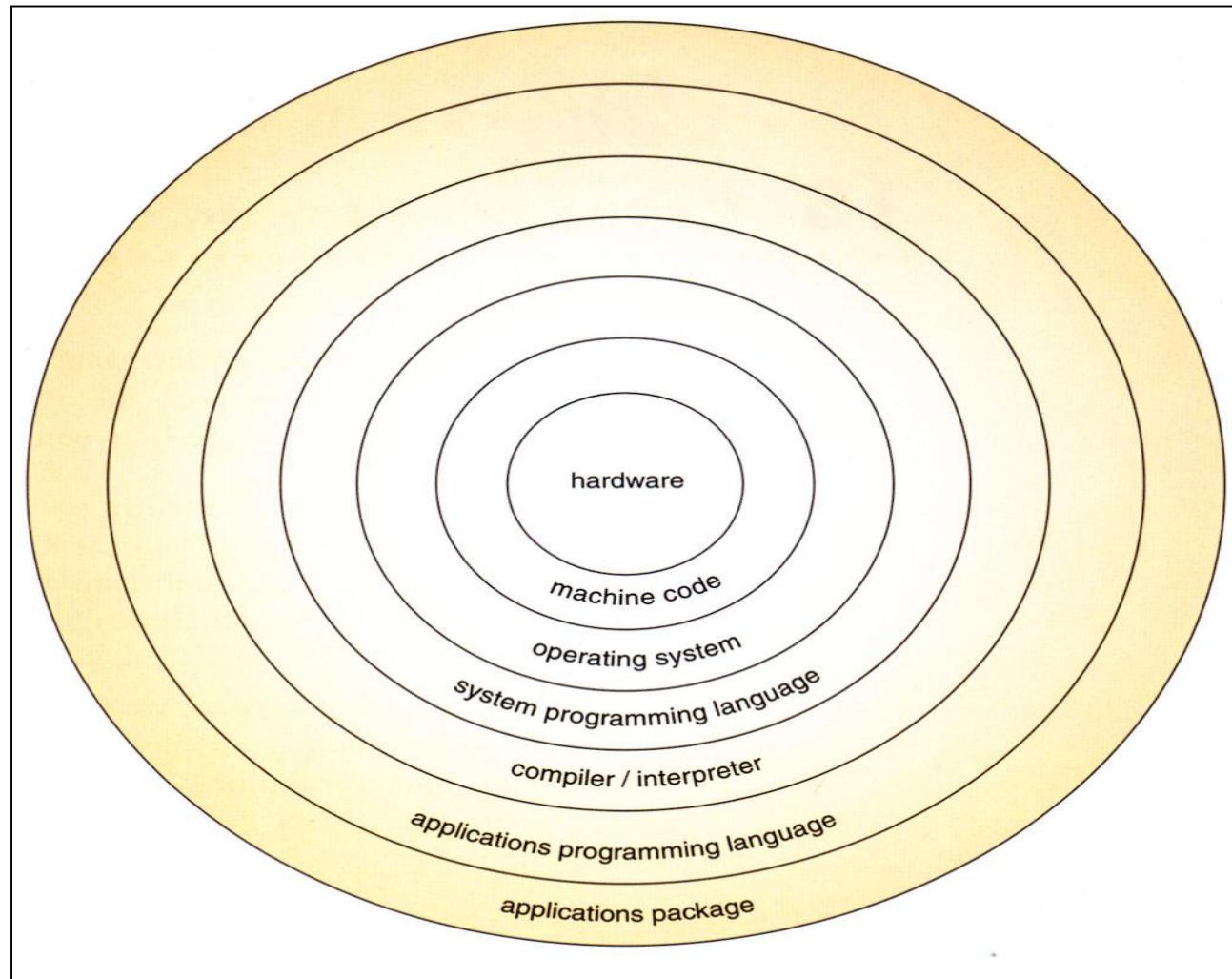
- Operating System: It hides the underlying complex hardware. It controls and coordinates use of hardware among various applications and users.
- Utilities and Application Software: define the ways of in which the system resources are used to solve the computing problems of the users. The system programs like compiler, assembler, editor etc. run in the user mode where the OS run in the supervisor mode or kernel mode.

**Users:** People, machines, other computers etc.

# Abstract View of Components of Computer



# Detail Layered View of Computer



# Three types of programs

## User / application programs

- programs used by the users to perform a task
- Performs specific tasks for users
  - Business application
  - Communications application
  - Multimedia application
  - Entertainment and educational software

## System programs

- an interface between user and computer
- Performs essential operation tasks
  - Operating system
  - Utility programs

## Driver programs (Device Drivers)

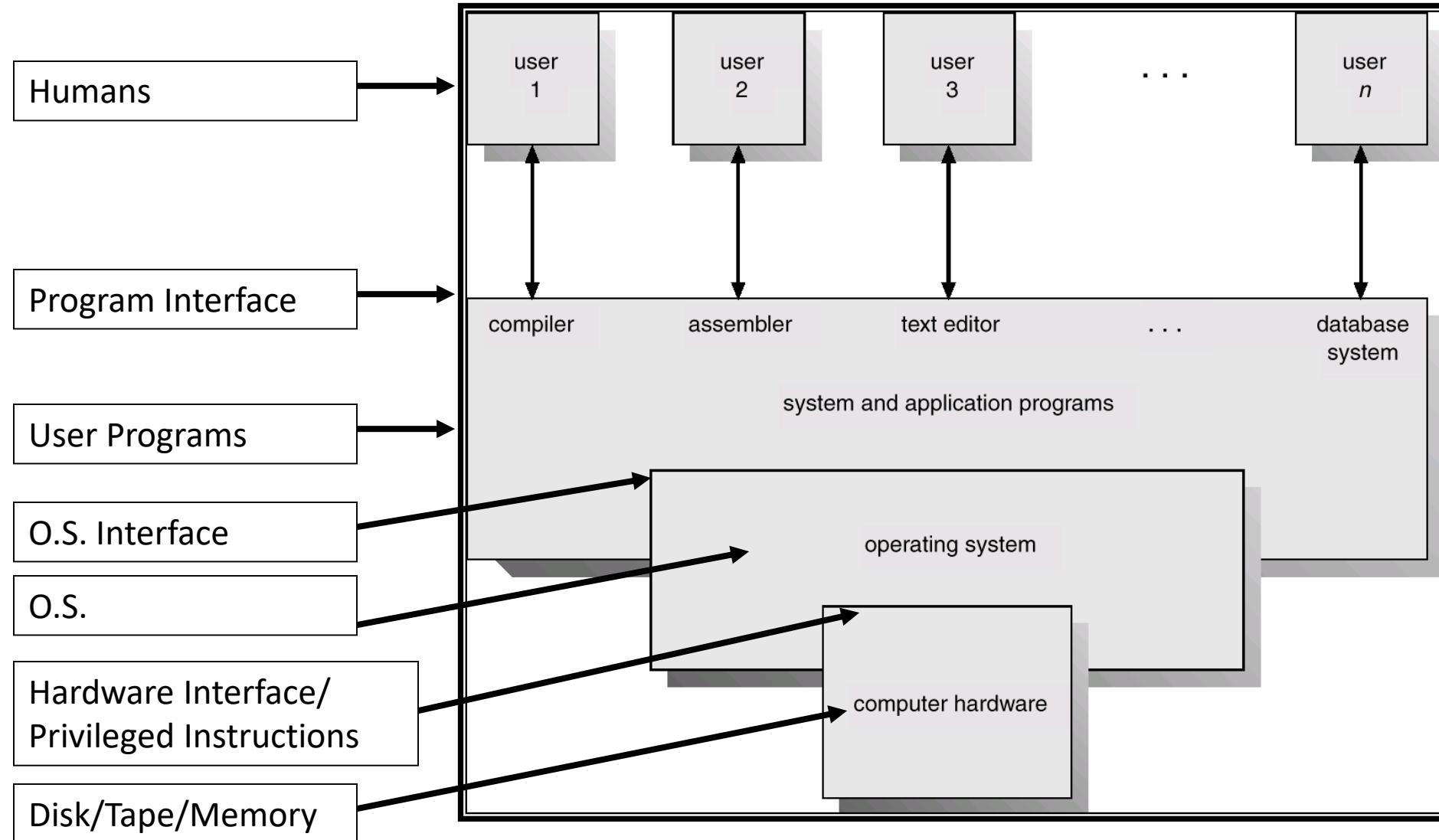
- small program that allows a specific input or output device to communicate with the rest of the computer system

# Computer Startup

- **bootstrap program** is loaded at power-up or reboot
  - Typically stored in ROM or EPROM, generally known as **firmware**
  - Initializes all aspects of system including the CPU registers, device controllers, and memory contents
  - Locates and loads operating system kernel and starts execution of the first process (such as “init”) and waits for events to occur.

# OPERATING SYSTEM OVERVIEW

## The Layers of a System



# OS

- Operating System (OS) is a program or set of programs, which acts as an interface between a user of the computer & the computer hardware.
- It is written in low-level languages (i.e. machine-dependent).
- When the computer is on, OS will first load into the main memory

# Definitions

## OS

- is a layer of software that manages all the hardware and software components in an effective and efficient manner.
- is a resource allocator that decides between conflicting requests for efficient and fair resource use.
- acts as a mediator, thereby making it easier for the programmer to access and use of the facilities and services of the computing system.
- implements a virtual machine that is easier to program than bare hardware.
- provides an abstraction of the hardware for all the user programs thereby hiding the complexity of the underlying hardware and gives the *user* a better view of the computer.
- is a control program that controls execution of programs to prevent errors and improper use of computer.

# Why do we need operating systems?

- The primary need for the OS arises from the fact that user needs to be provided with services and OS ought to facilitate the provisioning of these services.
- Convenience: OS provides a high-level abstraction of physical resources; make hardware usable by getting rid of warts and specifics; enables the construction of more complex software systems and enables portable code.
- Efficiency: It shares limited or expensive physical resources and provides protection.

# What is an Operating System?

- It is a *resource manager*
  - provides orderly and controlled allocation for programs in terms of time and space, *multiplexing*
  - Resource management keeps track of the resource, decides who gets the access and resolves conflicting requests for resources by enforcing policy, allocates the resource and after use reclaims the resource.

# What is an Operating System?

- It is an *extended*, or *virtual machine*
  - creates the functionalities of a computer in software i.e. creating copies of processors, memory etc. using software.
  - provides a simple, high-level abstraction, i.e., hides the “messy details” which must be performed
  - presents user with a virtual machine, easier to use
  - provides services; programs obtain these by *system calls*

# Services Provided by the OS

- An OS provides standard **services** (an interface), such as Processes, CPU scheduling, memory management, file system, networking, etc. In addition to this, OS provides services in the following areas:
- Program development: These services are in the form of utility programs such as editors, debuggers to assist programmers in creating programs. These are referred as application development tools.
- Program execution: The tasks that are performed to execute a program are loading of instructions and data into the main memory, initializing I/O devices and files and preparing other resources.

- Access to I/O devices: Each I/O device requires its own peculiar set of instructions or control signals for operation. OS provides a uniform interface that hides these details so that the programmer can access such devices using simple reads and writes.
- Controlled access to files: For accessing files, control must include a detailed understanding of the nature of I/O device as well as the structure of the data contained in the files on the storage medium. For system with multiple users, OS should provide protection mechanisms to control access to the files.

- System access: For shared or public systems, the access function must provide protection of resources and data from unauthorised users and must resolve conflicts for resource contention.
- Error Detection and Responses: OS must provide responses to runtime errors such as internal or external hardware errors, software errors like arithmetic overflow etc. The response may be terminating the program or reporting error to the application.
- Accounting: OS collects usage statistics for various resources and monitor performance parameters to improve performance. On a multiuser system, the information can be used for billing purposes.

# Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*
- A *trap* is a software-generated interrupt caused either by an error such as divide by 0 or a user request (system call).
- An operating system is **interrupt driven**

# Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter
- Separate segments of code determine what action should be taken for each type of interrupt

# I/O Structure

- After I/O starts, control returns to user program without waiting for I/O completion
  - **System call** – request to the operating system to allow user to wait for I/O completion
  - **Device-status table** contains entry for each I/O device indicating its type, address, and state
  - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt

# Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

# Storage Structure

- Main memory – only large storage media that the CPU can access directly
- Programs and data cannot reside in main memory permanently because:
  - Main memory is limited (too small) to store all programs and data permanently
  - Main memory is volatile
- So secondary storage is provided – extension of main memory that provides large nonvolatile storage capacity
- Magnetic 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

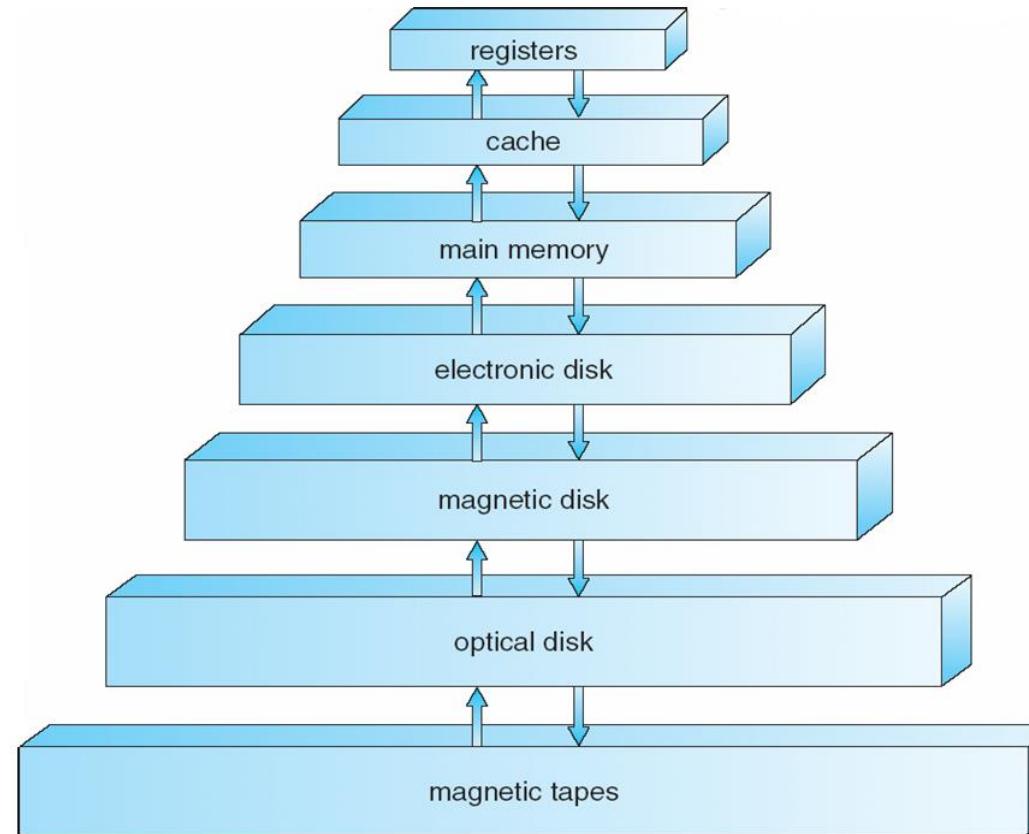
# Storage Hierarchy

- Storage systems organized in hierarchy by
  - Speed
  - Cost
  - Volatility
- The higher levels in the hierarchy are expensive but fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.
- **Caching** – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage

# Caching

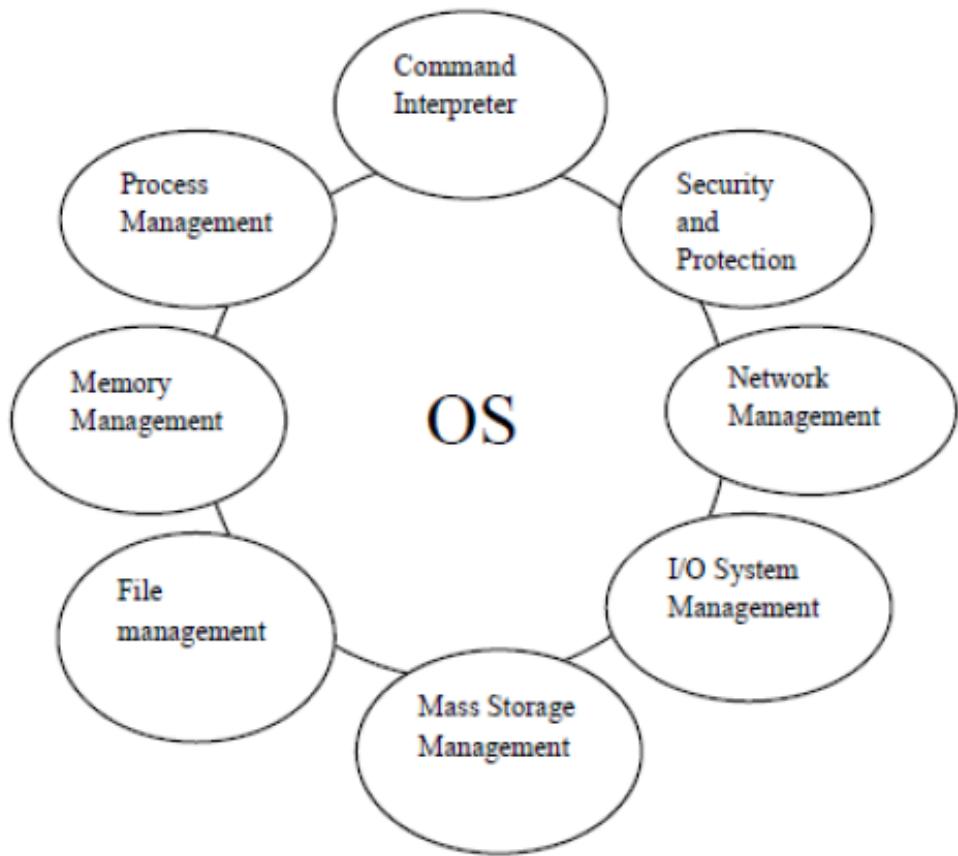
- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

# Storage-Device Hierarchy



# Major components of OS

1. Process Management
2. Main Memory Management
3. File Management
4. Mass Storage Management
5. I/O System Management
6. Network Management
7. Security and Protection system
8. Command Interpreter System



# Process Management

- A process is an instance of a program in execution and is the fundamental unit of computation in a computer.
- A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task. Processes can create subprocesses to execute concurrently.
- Programs can be executed concurrently as the number of processes active at a time may be more than one. These multiple processes which are active may communicate with each other and some may access resources which are mutually exclusive in nature.

## The activities of a process manager include

- creating and deleting of user and system processes,
- suspending and resuming processes,
- scheduling processes,
- providing mechanisms for process synchronization,
- providing mechanisms for inter process communication
- handling deadlocks during concurrent execution.

# Main Memory management

- Main memory is a volatile storage device. It loses its contents in the case of system failure.
- One can view memory as large array of words or bytes, each with its own address.
- It is storage of quickly accessible data shared by the CPU and I/O devices.
- A program resides on a disk as a binary executable file and it is brought to the main memory for execution.

## The functions of memory management are

- keeping track of memory in use,
- allocation of memory to processes by deciding which processes and data are to be moved into and out of memory and move processes between disk and memory.

# File Management

- The other names for this management are information management and storage management.
- OS provides uniform, logical view of information storage. It abstracts physical properties to logical storage unit called file.
- A file is a collection of related information defined by its creator. The information may be a sequence of bits, bytes, lines, or records whose meanings are defined by their creators. Commonly, files represent programs (both source and object forms) and data.

- The File management is responsible for allocation of space for programs and data on disk, creation and deletion of files, creation and deletion of directories, providing primitives to support for manipulating files and directories, mapping files onto secondary storage, maintaining file backup on stable storages.
- In addition, it keeps track of the information, its location, its usage, status using file system, decides who gets hold of information, enforce protection and provide access mechanism.
- Allocation and de-allocation of files are done using open and close system calls.

# Mass Storage Management

- Since main memory is volatile in nature and too small to accommodate all programs and data, disks are used to store data and programs that does not fit in main memory or data that must be kept for a long period of time.
- Most modern computer systems use disks as the principle on-line storage medium, for both programs and data.

- Functions of this storage management includes free space management, storage allocation, disk scheduling, disk buffer management to reduce the disk access time as some storage devices are slow in nature.
- It also manages the tertiary storage such as optical storage devices, magnetic tapes for back up purpose. These devices vary between WORM (Write Once Read Many times) and RW (Read Write).

# I/O System management

- The I/O system consists of a buffer-caching system, a general device-driver interface and drivers for specific hardware devices.
- The buffer caching system is to reduce the I/O access time.
- When a process wishes to access an I/O device it must issue a system call to OS which has device drivers to facilitate I/O functions involving I/O devices. These device drivers are software routines that control respective I/O devices through their controllers.
- The OS hides the peculiarities of specific hardware devices from the user.

- The I/O system management functions are keeping track of the I/O devices, I/O channels, etc. using I/O traffic controller.
- It performs I/O scheduling by deciding what is an efficient way to allocate the I/O resource and if it is to be shared, then deciding who gets it, how much of it is to be allocated, and for how long.
- It allocates the I/O device and initiates the I/O operation and when the use of the device is through reclaiming is done by the I/O system or in some I/O operation it terminates automatically.

# Network Management

- An OS is responsible for the computer system networking via a distributed environment which allows computers to work together.
- A distributed system is a collection of autonomous computers and do not have a common clock or shared memory. Each computer has their own clock and memory and communicates with each other through networks.
- Several networking protocols such as TCP/IP (Transmission Control Protocol/ Internet Protocol), UDP (User Datagram Protocol), FTP (File Transfer Protocol), HTTP (Hyper Text Transfer protocol), NFS (Network File System) etc. are used.
- The resources like processors, memory etc., are shared and access to these shared resources improves the computation speed-up, increases availability and enhances reliability.

# Security and Protection System

- Security refers to defense of the system against internal and external attacks which is of huge range including worms, viruses, Trojan horses, denial of services, identity theft, etc.
- OS is responsible for detecting and preventing these attacks.
- Protection refers to a mechanism for controlling access of processes or users to resources defined by OS.
- The protection mechanism provided by OS must distinguish between authorized and unauthorized usage, specify the controls to be imposed and provide a means of enforcement.

# Command-Interpreter System

- A user interacts with OS via one or more user applications through a special application called a shell or command interpreter which acts as an interface between the user and the operating system.
- Today almost all OS provides a user friendly interface, namely Graphical User Interface (GUI).
- The commands given to OS are control statements that deal with process creation and management, I/O handling, secondary-storage management, main memory management, file system access, protection and networking.

<i>From Architecture to OS to Application, and Back</i>			
	<b>Hardware</b>	<b>Example OS Services</b>	<b>User Abstraction</b>
Processor	Process management, Scheduling, Traps, Protections, Billing, Synchronization		Process
Memory	Management, Protection, Virtual memory		Address space
I/O devices	Concurrency with CPU, Interrupt handling		Terminal, Mouse, Printer, (System Calls)
File system	Management, Persistence		Files
Distributed systems	Network security, Distributed file system		RPC system calls, Transparent file sharing

*From Architectural to OS to Application, and Back*

<b>OS Service</b>	<b>Hardware Support</b>
Protection	Kernel / User mode Protected Instructions Base and Limit Registers
Interrupts	Interrupt Vectors
System calls	Trap instructions and trap vectors
I/O	Interrupts or Memory-Mapping
Scheduling, error recovery, billing	Timer
Synchronization	Atomic instructions
Virtual Memory	Translation look-aside buffers Register pointing to base of page table

# Types of OS

# Evolution of OS:

- The evolution of operating systems went through seven *major phases*.
- Six of them significantly changed the ways in which users accessed computers through the open shop, batch processing, multiprogramming, timesharing, personal computing, and distributed systems.
- In the seventh phase the foundations of concurrent programming were developed and demonstrated in model operating systems.

## Evolution of OS (contd..):

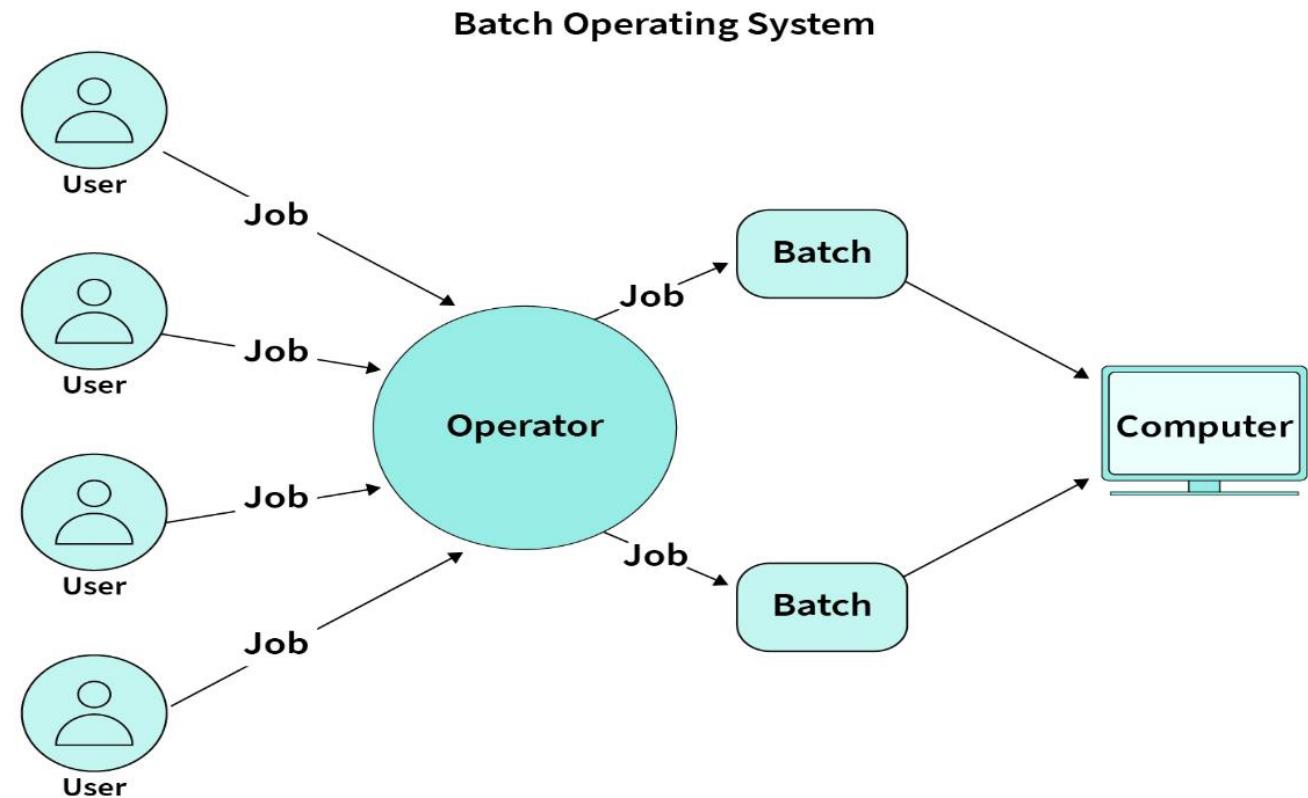
Major Phases	Technical Innovations	Operating Systems
Open Shop	The idea of OS Programmers write programs and submit tape/cards to operator. Operator feeds cards, collects output from printer.	IBM 701 open shop (1954)
Batch Processing	Tape batching, First-in, first-out scheduling.	BKS system (1961)
Multi-programming	Processor multiplexing, Indivisible operations, Demand paging, Input/output spooling, Priority scheduling, Remote job entry	Atlas supervisor (1961), Exec II system (1966)

# Evolution of OS (contd..):

Timesharing	Simultaneous user interaction, On-line file systems	Multics file system (1965), Unix (1974)
Concurrent Programming	Hierarchical systems, Extensible kernels, Parallel programming concepts, Secure parallel languages	RC 4000 system (1969), 13 Venus system (1972), 14 Boss 2 system (1975).
Personal Computing	Graphic user interfaces	OS 6 (1972) Pilot system (1980)
Distributed Systems	Remote servers	WFS file server (1979) Unix United RPC (1982) 24 Amoeba system (1990)

# Batch Systems

- Introduction of tape drives allow batching of jobs
- In this type of system, there is **no direct interaction between user and the computer.**
- The user has to submit a job (written on cards or tape) to a computer operator.
- Then computer operator places a batch of several jobs on an input device.
- Jobs are batched together by type of languages and requirement.



Computer now has a resident monitor :

- initially control is in monitor.
- monitor reads job and transfer control.
- at end of job, control transfers back to monitor.
- The monitor is always in the main memory and available for execution.



Even better: spooling systems.

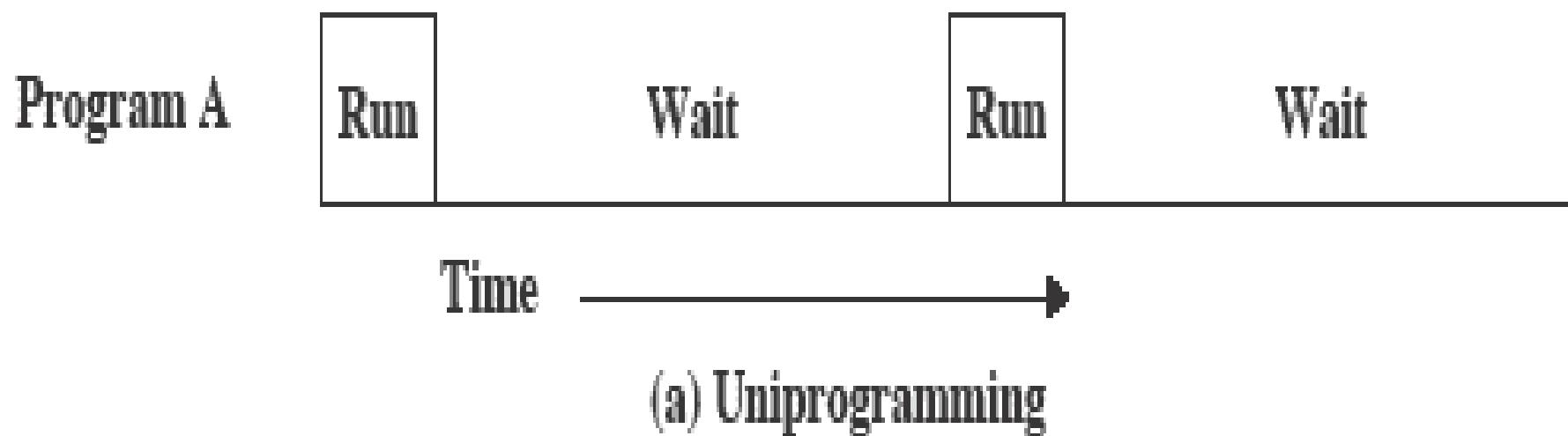
- use interrupt driven I/O.
- use magnetic disk to cache input tape.
- Monitor now schedules jobs. . .

## **Advantages of Simple Batch Systems**

- No interaction between user and computer.
- No mechanism to prioritize the processes.

# Uniprogramming

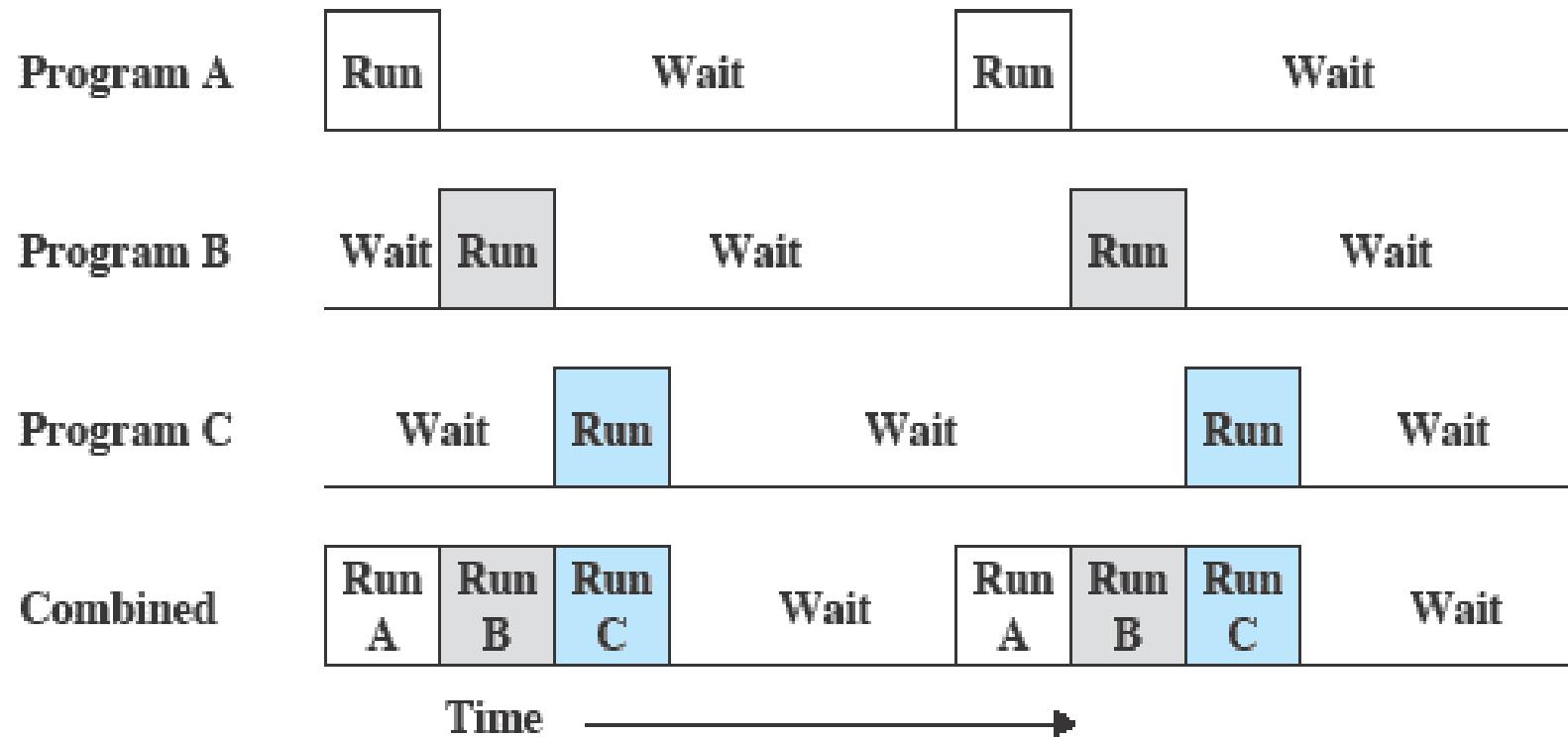
- Processor must wait for I/O instruction to complete before preceding



# Multiprogramming Batch Systems

- Single user cannot keep CPU and I/O devices busy at all times
- Multiprogramming organizes jobs (code and data) so CPU always has one to execute
- A subset of total jobs in system is kept in memory
- In this, the operating system picks up and begins to execute one of the jobs from memory.
- Once this job needs an I/O operation operating system switches to another job (CPU and OS always busy).
- Jobs in the memory are always less than the number of jobs on disk(Job Pool).

# Multiprogramming

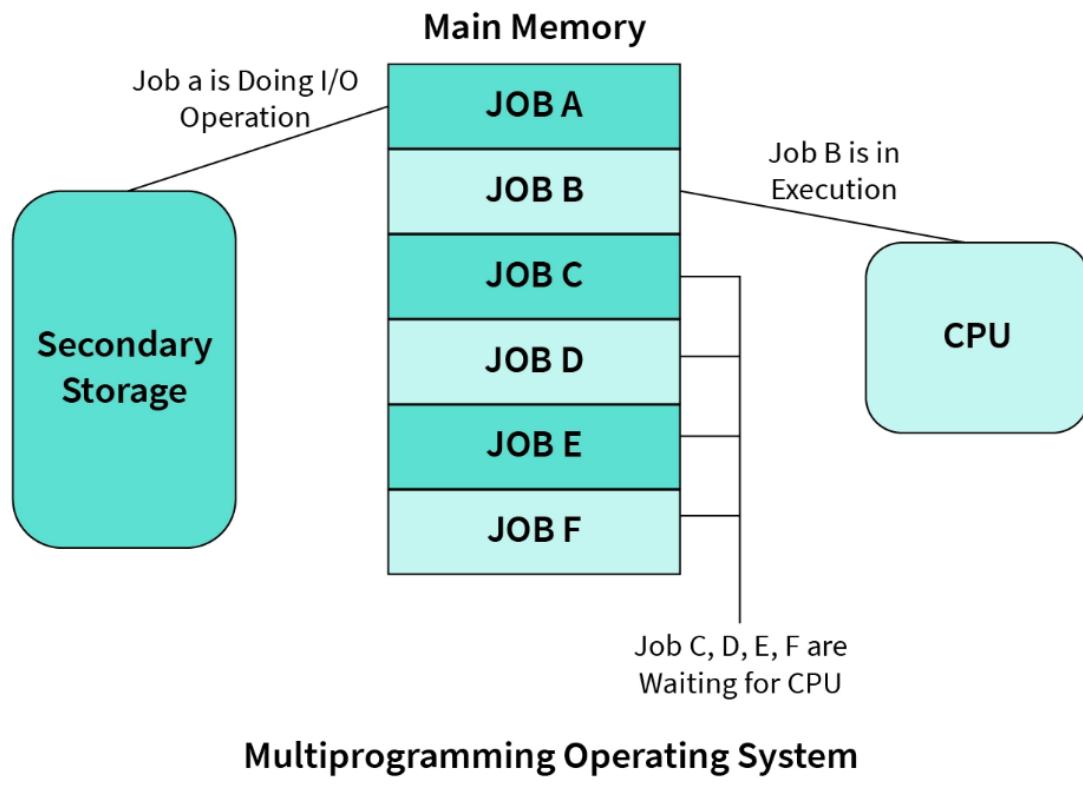


(c) Multiprogramming with three programs

- Use memory to cache jobs from disk i.e., more than one job active simultaneously.

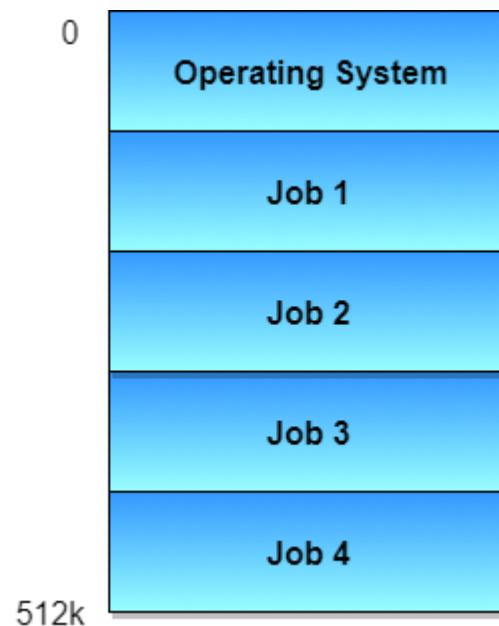
Two stage scheduling:

- select jobs to load: job scheduling.
- select resident job to run: CPU scheduling.
- If several jobs are ready to run at the same time, then the system chooses which one to run through the process of **CPU Scheduling**.
- In Non-multiprogrammed system, there are moments when CPU sits idle and does not do any work.
- In Multiprogramming system, CPU will never be idle and keeps on processing.



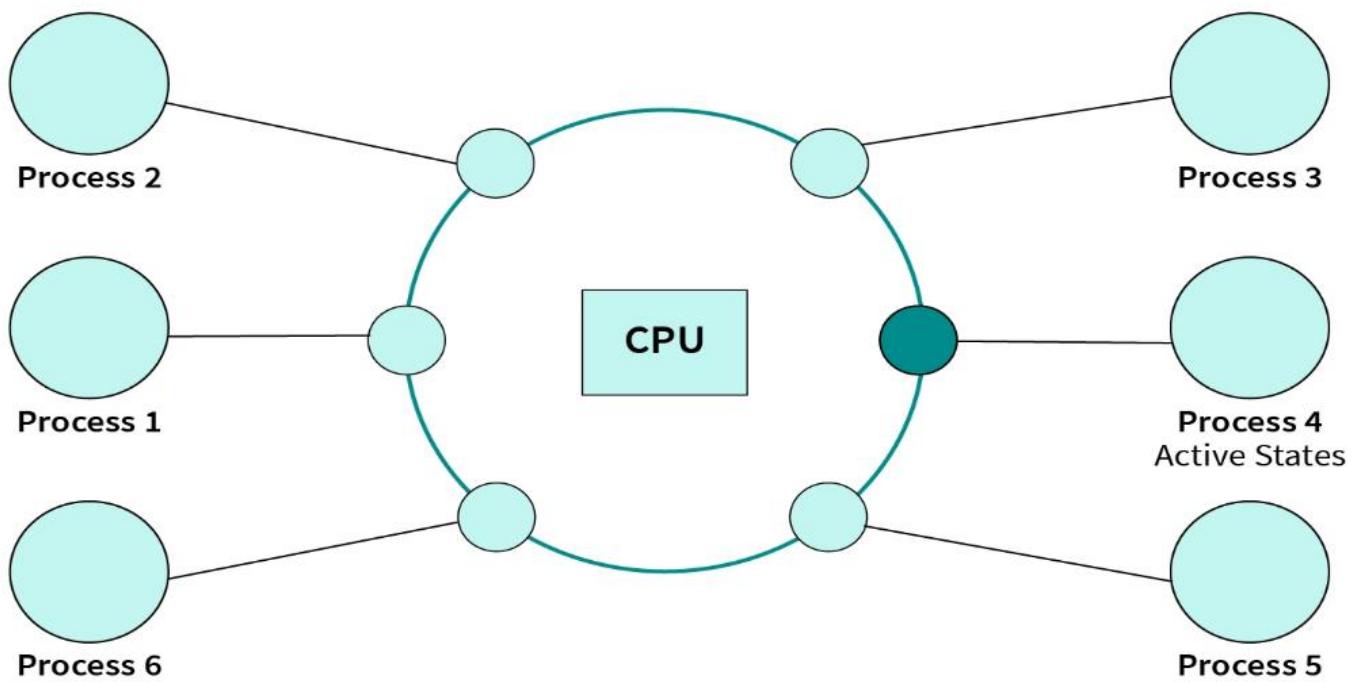
## Advantages:

- Efficient memory utilization
- Throughput increases
- CPU is never idle, so performance increases.



# Time Sharing Systems

- **Time Sharing Systems** are very similar to Multiprogramming batch systems.
- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
- Time slice is defined by the OS, for sharing CPU time between processes.
- Examples: Multics, Unix, etc.,
- In Time sharing systems the prime focus is on **minimizing the response time**, while in multiprogramming the prime focus is to maximize the CPU usage.
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory  $\Rightarrow$  **process**
  - If several jobs ready to run at the same time  $\Rightarrow$  **CPU scheduling**
  - If processes don't fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory



# Multiprocessor Systems

- Also known as **parallel systems, tightly-coupled systems**
- A Multiprocessor system consists of several processors that share a common physical memory.
- Multiprocessor system provides higher computing power and speed.
- In multiprocessor system all processors operate under single operating system. Multiplicity of the processors and how they do act together are transparent to the others.

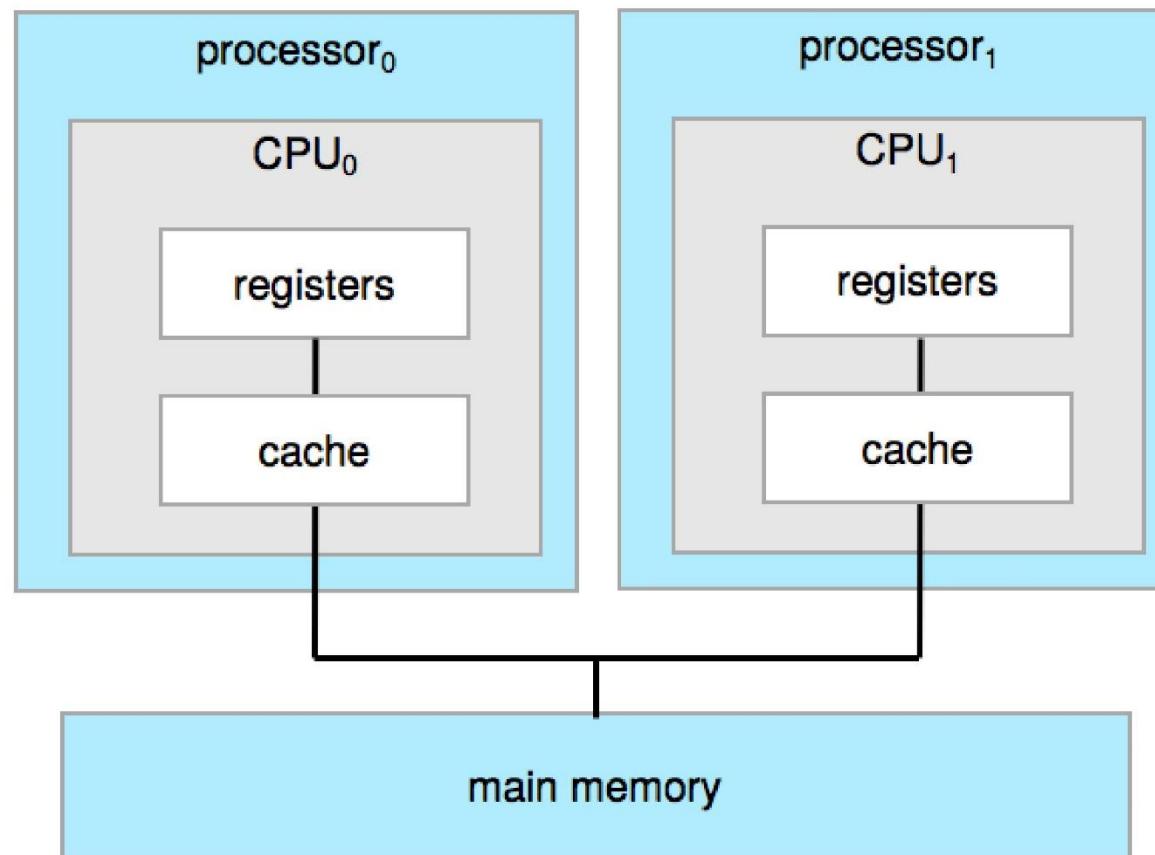
## **Advantages of Multiprocessor Systems**

- Enhanced performance
  - Increased throughput - Execution of several tasks by different processors concurrently, increases the system's throughput without speeding up the execution of a single task.
  - If possible, system divides task into many subtasks and then these subtasks can be executed in parallel in different processors. Thereby speeding up the execution of single tasks.
- Economy of scale
- Increased reliability – graceful degradation or fault tolerance

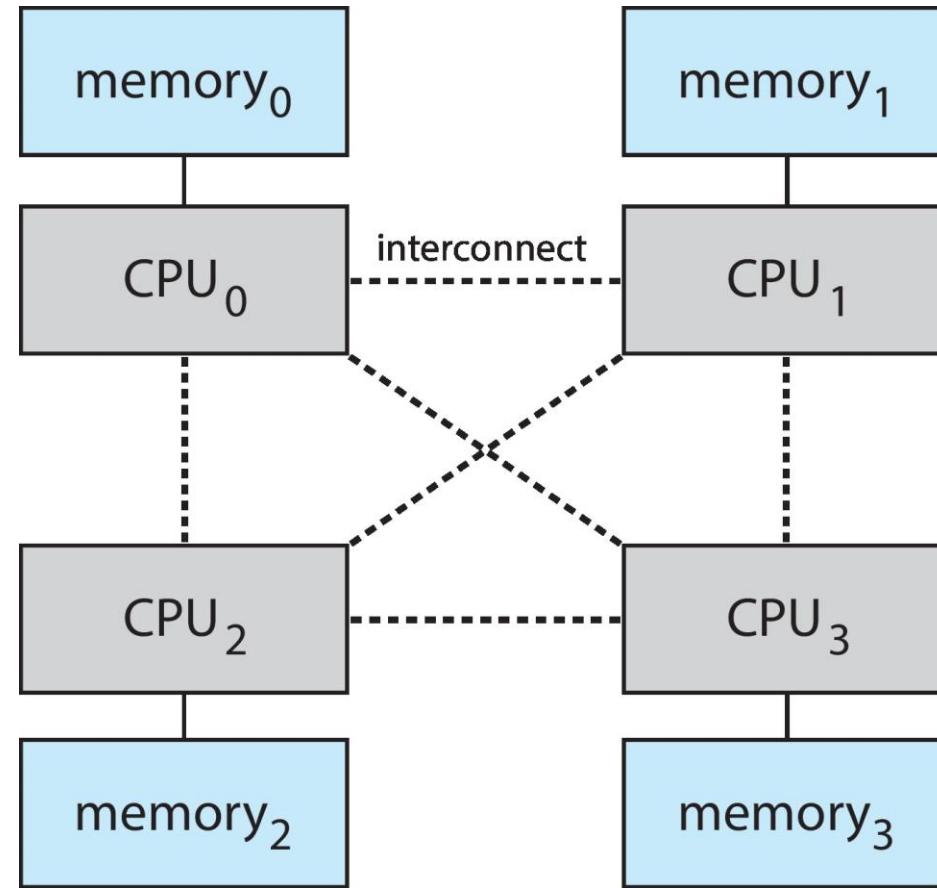
Two types:

- 1. Asymmetric Multiprocessing** – each processor is assigned a specific task.
- 2. Symmetric Multiprocessing** – each processor performs all tasks

# Symmetric Multiprocessing Architecture



# Non-Uniform Memory Access System

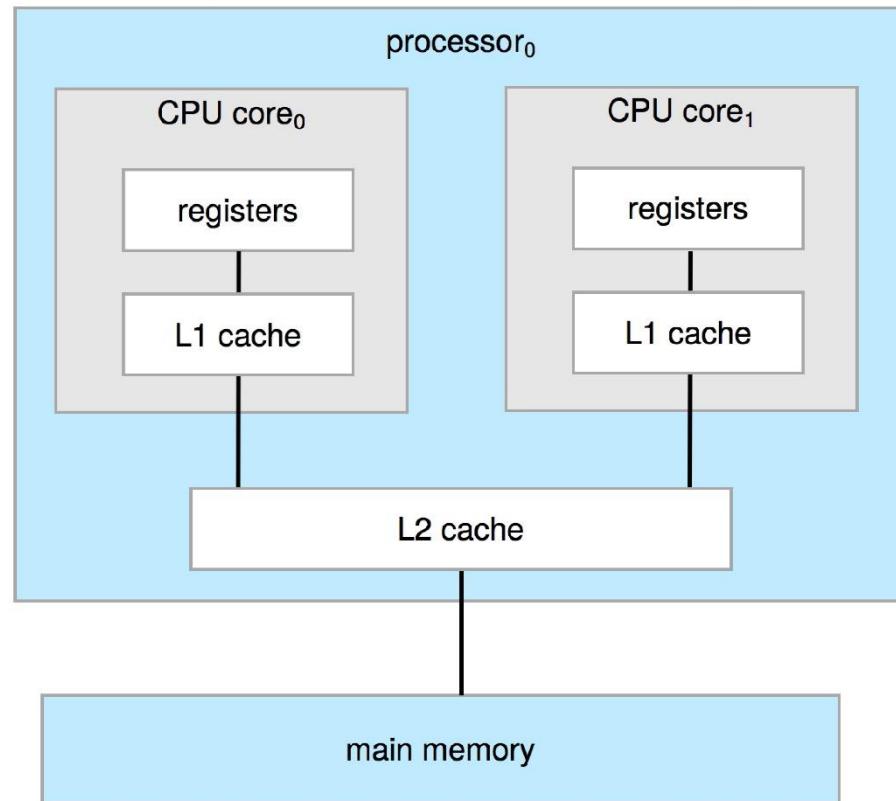


# A Dual-Core Design

- A recent trend in CPU design is to include multiple computing **cores** on a single chip.
- Multi-chip and **multicore**
- Systems containing all chips
  - Chassis containing multiple separate systems

Advantages:

- on-chip communication is faster than between-chip communication.
- One chip with multiple cores uses significantly less power than multiple single-core chips.

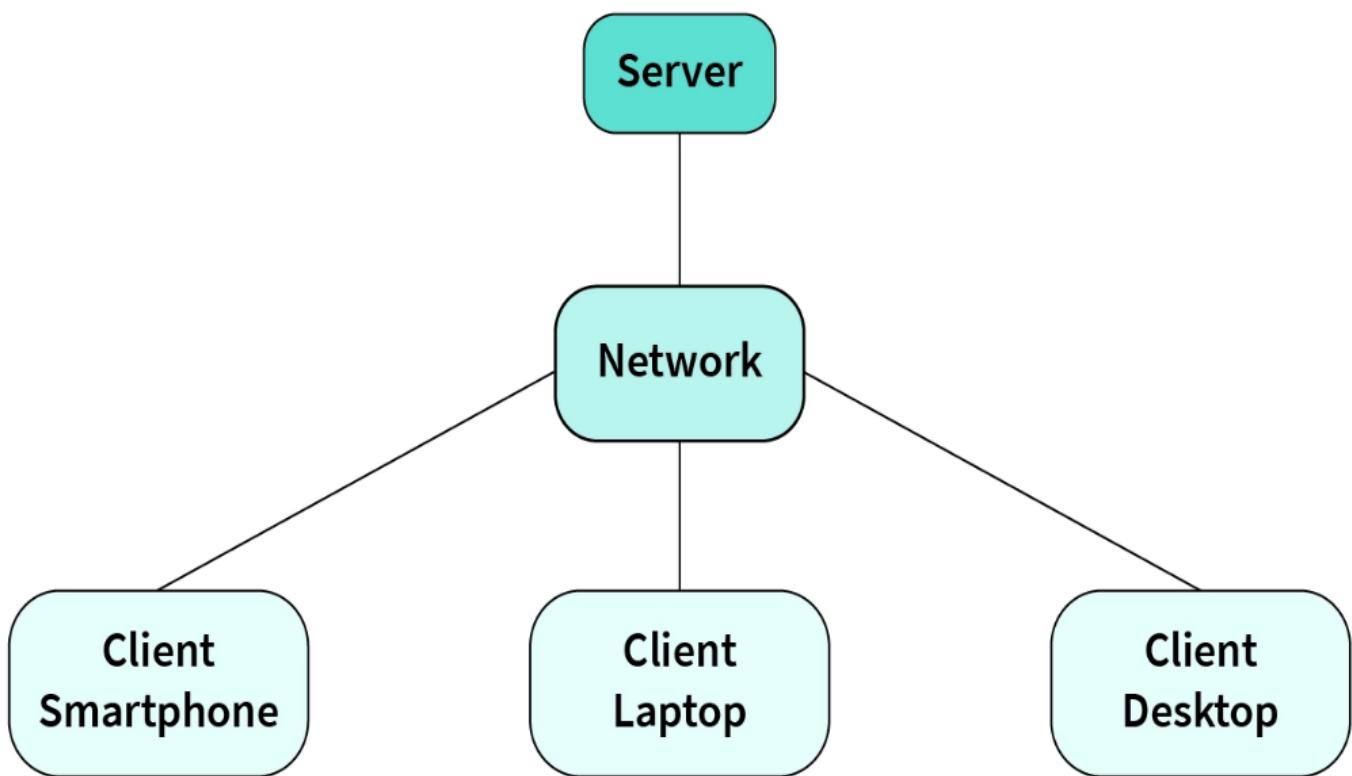


# Blade servers

- These are a relatively recent development in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis.
- The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system.
- These servers consist of multiple independent multiprocessor systems.

# Desktop Systems

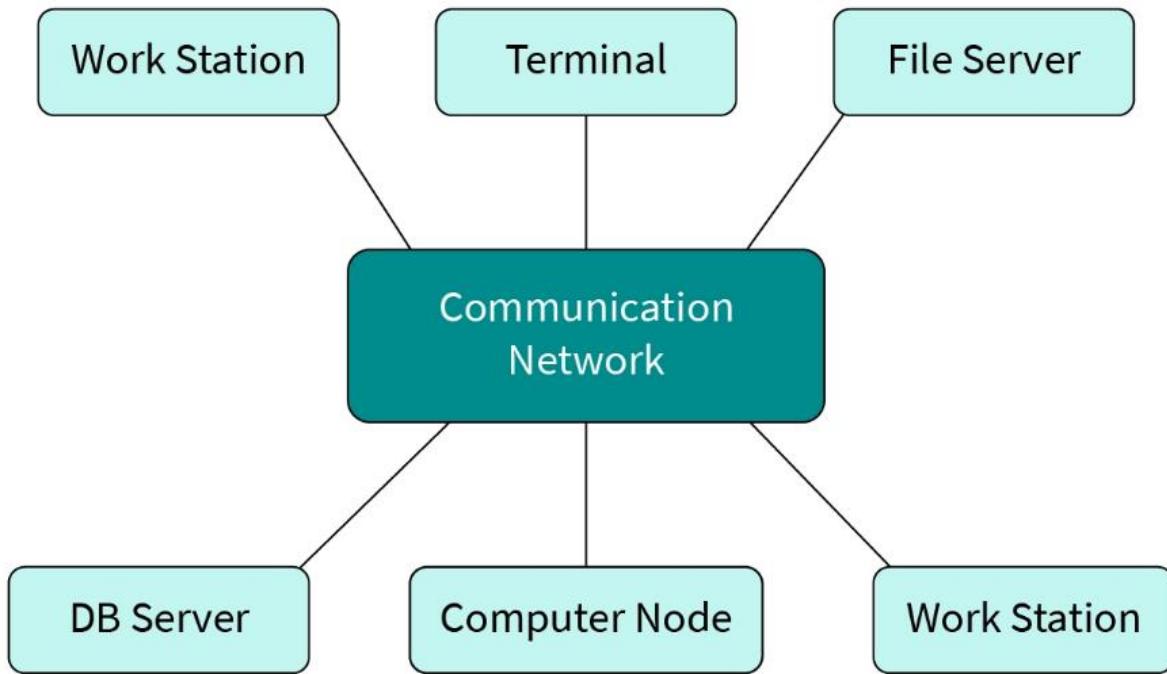
- PC operating systems were neither **multiuser** nor **multitasking**.
- However, the goals of these operating systems have changed with time; instead of maximizing CPU and peripheral utilization, the systems opt for maximizing user convenience and responsiveness. These systems are called **Desktop Systems** and include PCs running Microsoft Windows and the Apple Macintosh.



- Operating systems for these computers have benefited in several ways from the development of operating systems for **mainframes**.
- **Microcomputers** were immediately able to adopt some of the technology developed for larger operating systems.
- On the other hand, the hardware costs for microcomputers are sufficiently **low** that individuals have sole use of the computer, and CPU utilization is no longer a prime concern. Thus, some of the design decisions made in operating systems for mainframes may not be appropriate for smaller systems.

# Distributed Operating System

- The motivation behind developing distributed operating systems is the availability of powerful and inexpensive microprocessors and advances in communication technology.
- These advancements in technology have made it possible to design and develop distributed systems comprising of many computers that are interconnected by communication networks. The main benefit of distributed systems is its low price/performance ratio.
- **Advantages Distributed Operating System**
- As there are multiple systems involved, user at one site can utilize the resources of systems at other sites for resource-intensive tasks.
- Fast processing.
- Less load on the Host Machine.

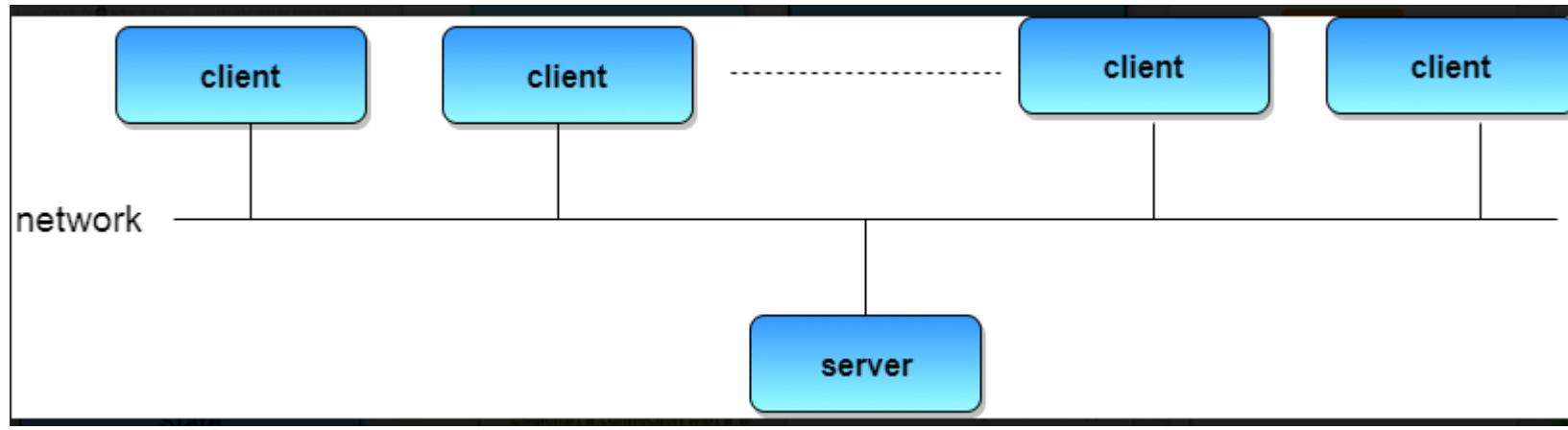


# Types of Distributed Operating Systems

- Following are the two types of distributed operating systems used:
  - Client-Server Systems
  - Peer-to-Peer Systems

## Client-Server Systems

- **Centralized systems** today act as **server systems** to satisfy requests generated by **client systems**.
- Server Systems can be broadly categorized as: **Compute Servers** and **File Servers**.
- **Compute Server systems**, provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.
- **File Server systems**, provide a file-system interface where clients can create, update, read, and delete files.



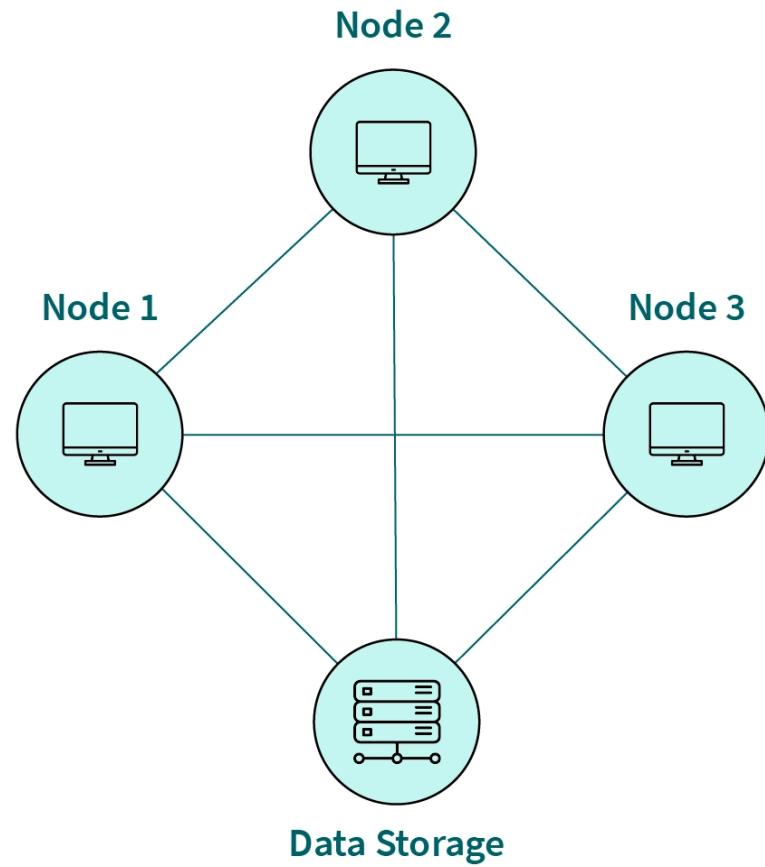
# Peer-to-Peer Systems

- The growth of computer networks - especially the Internet and World Wide Web (WWW) – has had a profound influence on the recent development of operating systems.
- When PCs were introduced in the 1970s, they were designed for **personal** use and were generally considered standalone computers.
- With the beginning of widespread public use of the Internet in the 1990s for electronic mail and FTP, many PCs became connected to computer networks.

- In contrast to the **Tightly Coupled** systems, the computer networks used in these applications consist of a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory.
- The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines. These systems are usually referred to as loosely coupled systems ( or distributed systems).

# Clustered Systems

- Like parallel systems, clustered systems gather together multiple CPUs to accomplish computational work.
- Clustered systems differ from parallel systems, however, in that they are composed of two or more individual systems coupled together.
- The definition of the term clustered is **not concrete**; the general accepted definition is that clustered computers share storage and are closely linked via LAN networking.
- Clustering is usually performed to provide **high availability**.



- A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others.
- If the monitored machine fails, the monitoring machine can take ownership of its storage, and restart the application(s) that were running on the failed machine.
- The failed machine can remain down, but the users and clients of the application would only see a brief interruption of service.

- **Asymmetric Clustering** - In this, 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.
- **Symmetric Clustering** - In this, two or more hosts are running applications, and they are monitoring each other. This mode is obviously more efficient, as it uses all of the available hardware.
- **Parallel Clustering** - Parallel clusters allow multiple hosts to access the same data on the shared storage. Because most operating systems lack support for this simultaneous data access by multiple hosts, parallel clusters are usually accomplished by special versions of software and special releases of applications.

- Clustered technology is rapidly changing. Clustered system's usage and it's features should expand greatly as **Storage Area Networks(SANs)**. SANs allow easy attachment of multiple hosts to multiple storage units.
- Current clusters are usually limited to two or four hosts due to the complexity of connecting the hosts to shared storage.

# Real Time Operating System

- It is defined as an operating system known to give maximum time for each of the critical operations that it performs, like OS calls and interrupt handling.
- The Real-Time Operating system which guarantees the maximum time for critical operations and complete them on time are referred to as **Hard Real-Time Operating Systems**.
- While the real-time operating systems that can only guarantee a maximum of the time, i.e. the critical task will get priority over other tasks, but not assured of completing it in a defined time. These systems are referred to as **Soft Real-Time Operating Systems**.

# Handheld Systems

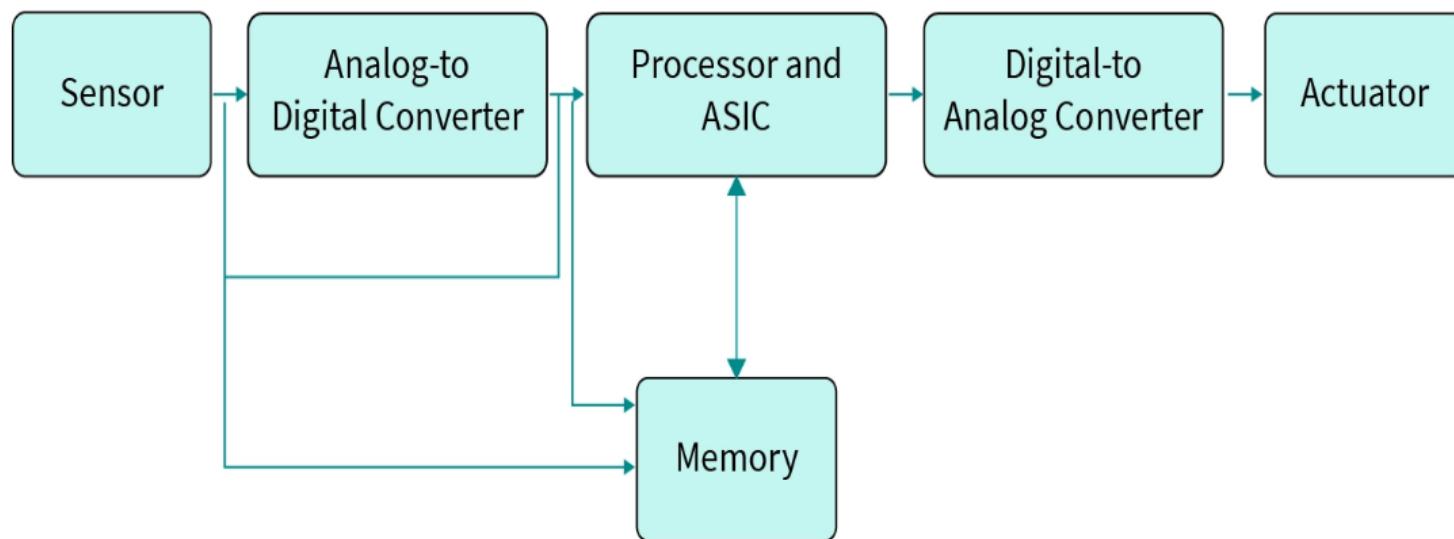
- Handheld systems include **Personal Digital Assistants(PDAs)**, such as Palm-Pilots or Cellular Telephones with connectivity to a network such as the Internet.
- They are usually of limited size due to which most handheld devices have a small amount of memory, include slow processors, and feature small display screens.

- Processors for most handheld devices often run at a fraction of the speed of a processor in a PC. Faster processors require **more power**. To include a faster processor in a handheld device would require a **larger battery** that would have to be replaced more frequently.
- The last issue confronting program designers for handheld devices is the small display screens typically available. One approach for displaying the content in web pages is **web clipping**, where only a small subset of a web page is delivered and displayed on the handheld device.

# Embedded Operating Systems

- An embedded operating system is a specialized OS for embedded systems. It aims to perform with certainty specific tasks regularly that help the device operate.
- An embedded operating system often has limited features and functions. The OS may perform only a single action that allows the device to work, but it must execute that action consistently and timely.
- Embedded operating systems are built into Internet of Things devices. They are also part of many other devices and systems. In most cases, embedded hardware doesn't have much capacity and has fewer resources. So, the amount of processing power and memory is limited.

## Embedded System Structure Diagram

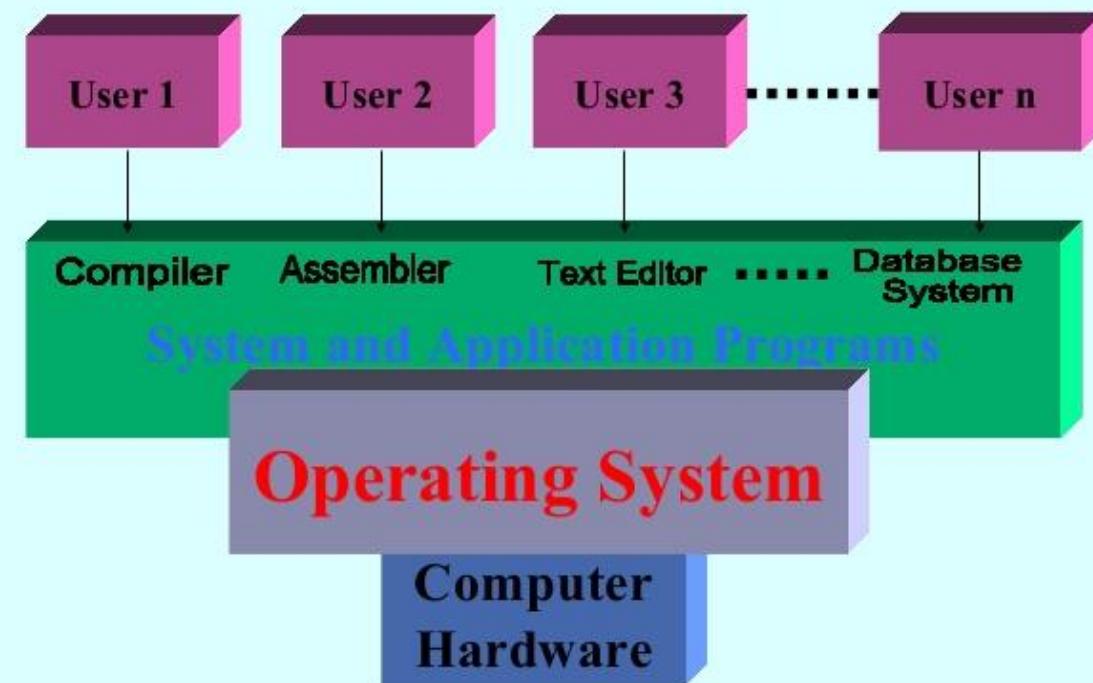


# Mobile Operating System

- A mobile operating system is an operating system that helps run application software on mobile devices. It is the same kind of software as the famous computer operating systems Linux and Windows, but they are light and simple to some extent.
- The operating systems found on smartphones include Symbian OS, IOS, BlackBerryOS, Windows Mobile, Palm WebOS, Android, and Maemo.
- Android, WebOS, and Maemo are all derived from Linux. The iPhone OS originated from BSD and NeXTSTEP, which are related to Unix. It combines the power of a computer and the experience of a hand-held device. It typically contains a cellular built-in modem and SIM tray for telephony and internet connections.

# Operating System Structure

## Abstract View of a Computer System



# Views of OS

Three views of an operating system

- Application View: what services does it provide?
- System View: what problems does it solve?
- Implementation View: how is it built

# Application View of an Operating System

The OS provides an execution environment for running programs.

## The execution environment

- provides a program with the processor time and memory space that it needs to run.
- provides interfaces through which a program can use networks, storage, I/O devices, and other system hardware components.
  - Interfaces provide a simplified, abstract view of hardware to application programs.
- isolates running programs from one another and prevents undesirable interactions among them.

# System View

The OS manages the hardware resources of a computer system. Resources include processors, memory, disks and other storage devices, network interfaces, I/O devices such as keyboard, mouse and monitor, and so on.

The operating system allocates resources among running programs.

It controls the sharing of resources among programs.

The OS itself also uses resources, which it must share with application programs.

# Implementation View

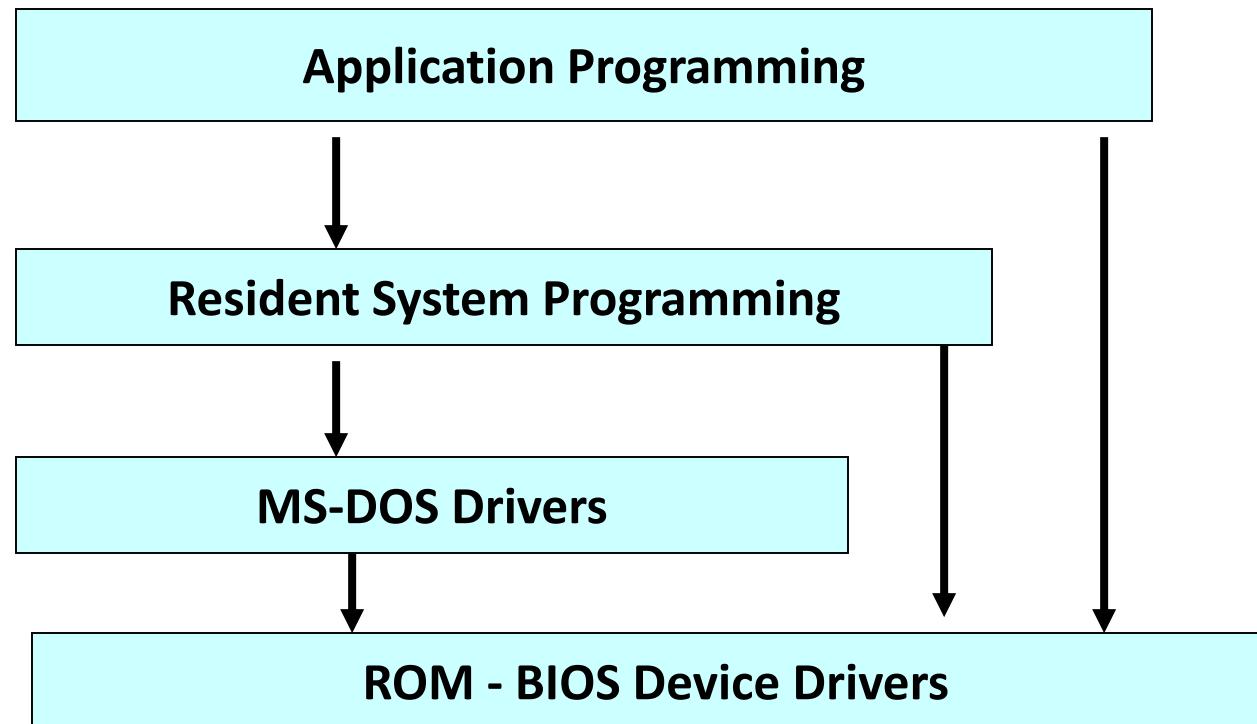
- The OS is a concurrent, real-time program.
- Concurrency arises naturally in an OS when it supports concurrent applications, and because it must interact directly with the hardware.
- Hardware interactions also impose timing constraints

# OS Implementation

# How An Operating System Is Put Together

A SIMPLE STRUCTURE:

Example of MS-DOS.



# Simple Structure

- Operating systems such as MS-DOS and the original UNIX did not have well-defined structures.
- There was no CPU Execution Mode (user and kernel), and so errors in applications could cause the whole system to crash.

# Monolithic Approach

- Functionality of the OS is invoked with simple function calls within the kernel, which is one large program.
- Device drivers are loaded into the running kernel and become part of the kernel.

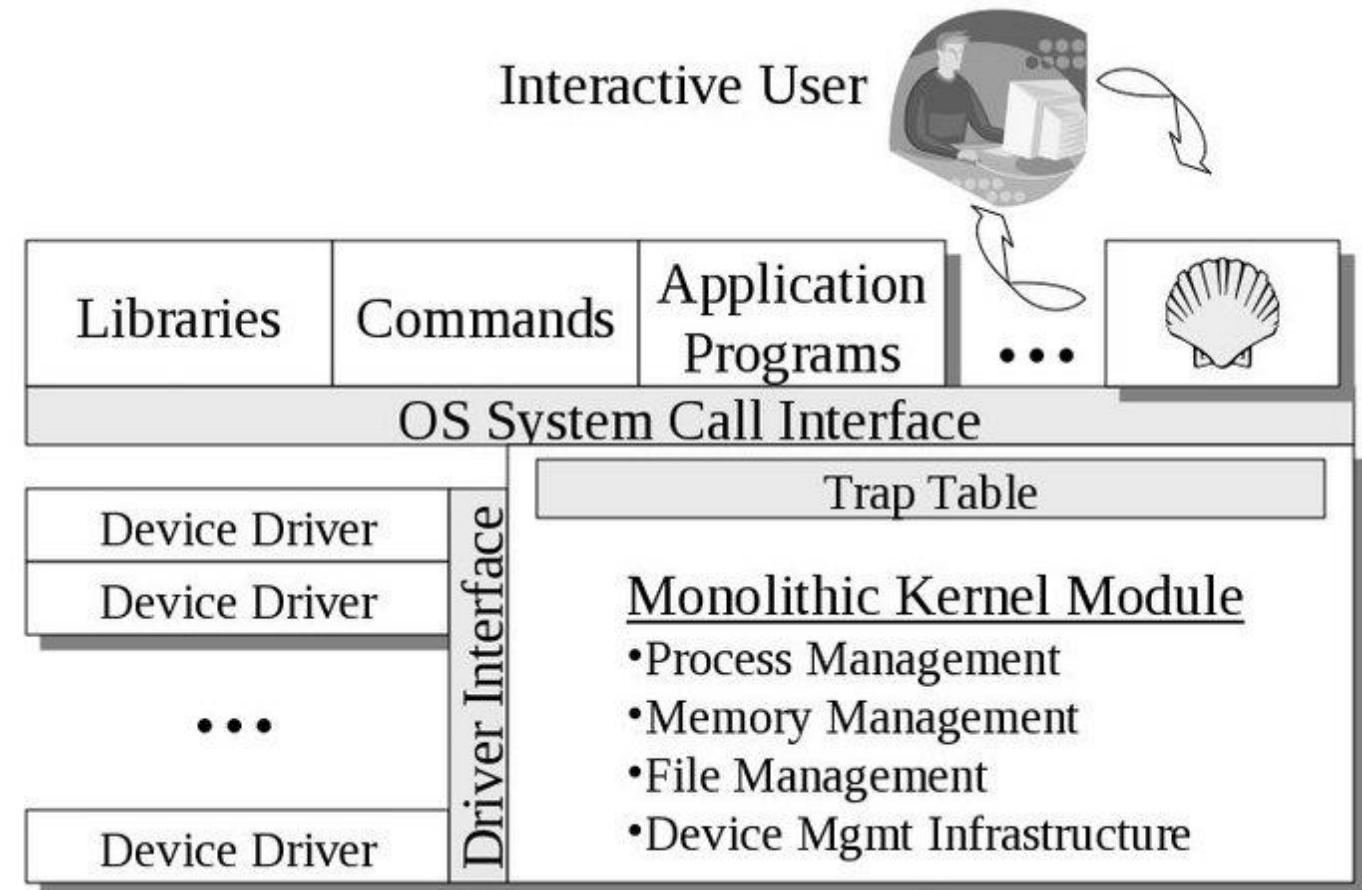
e.g. OS/360, VMS and Linux.

Advantage:

Direct interaction between components makes monolithic OS highly efficient.

Disadvantages:

It is difficult to isolate the source of bugs and other errors because monolithic kernels group components together and also all code executes with unrestricted access to the system.



# Layered Approach

- The layered approach to OS attempts to address the issue of OS becoming larger and more complex by grouping components that perform similar functions into layers.
- This approach breaks up the operating system into different layers.
- Each layer communicates exclusively with those immediately above and below it. Lower level layers provide services to higher level ones using an interface that hides their implementation.

- Layered OS are more modular than monolithic OS because the implementation of each layer can be modified without requiring any modification to other layers. A modular system has self contained components that can be reused throughout the system.
  - Each component hides how it performs its job and presents a standard interface that other components can use to request its services.
  - With the layered approach, the bottom layer is the hardware, while the highest layer is the user interface.
- 
- The main *advantage* is simplicity of construction and debugging.
  - The main *difficulty* is defining the various layers.
  - The main *disadvantage* is that the OS tends to be less efficient than other implementations.

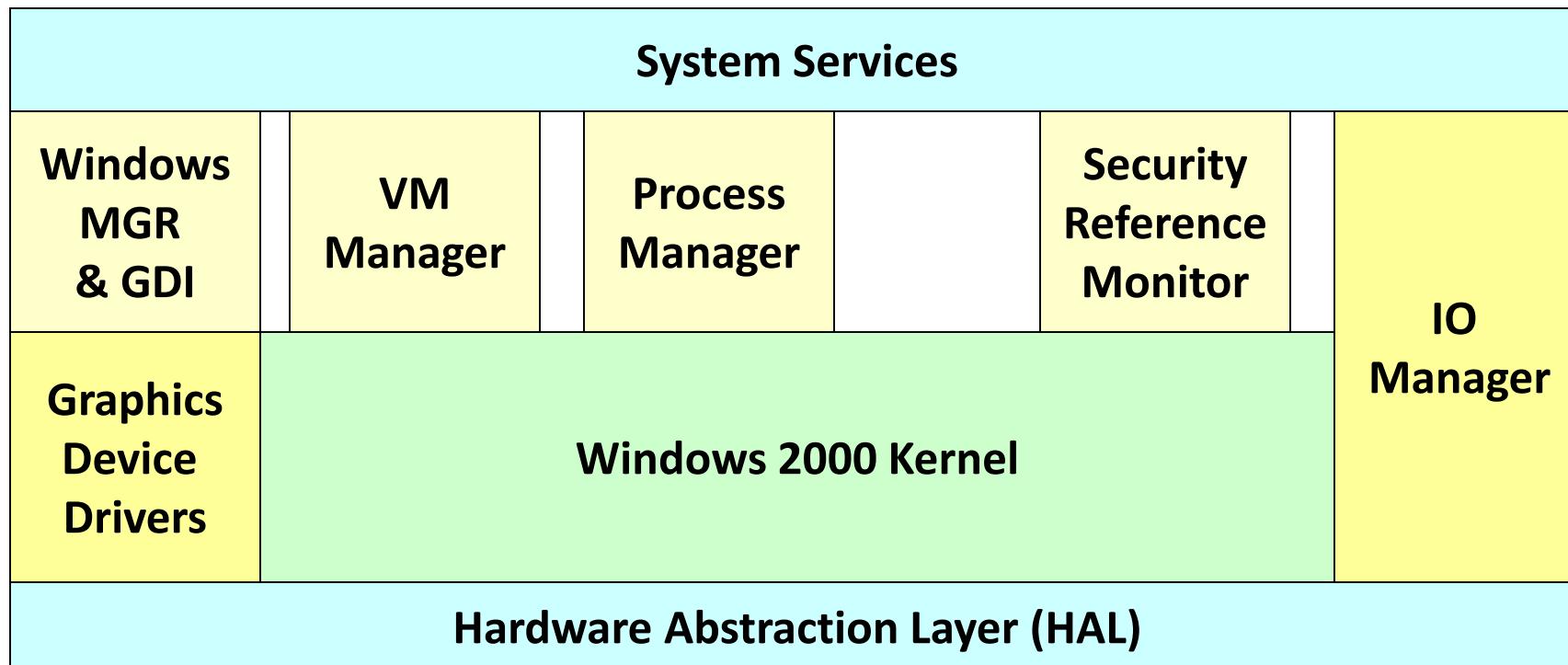
# Hierarchical OS Model

Level	Name	Objects	Typical operations
5	Command Language interpreter	Environmental data	Statements in Command language
4	File system	Files, devices	Create, destroy, open, close, read and write
3	Memory management	Segments, pages	read, write, fetch
2	Basic I/O	Data blocks	read, write, allocate, free
1	Kernel	Process, semaphore	create, destroy, suspend, resume, signal, wait

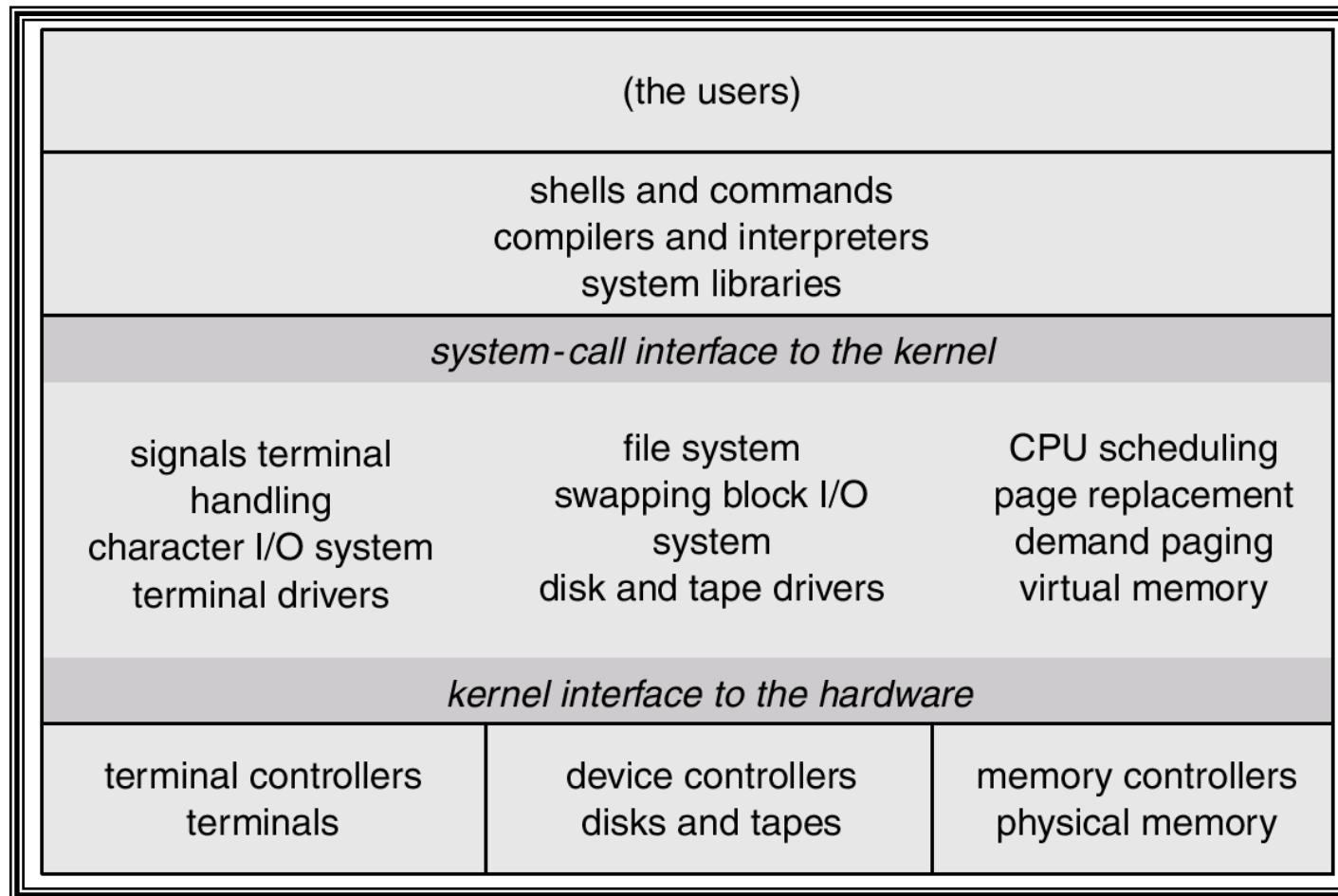
# Example - Windows 2000

MGR – ManaGeR

GDI – Graphical Data Interface



# Unix



# Microkernels

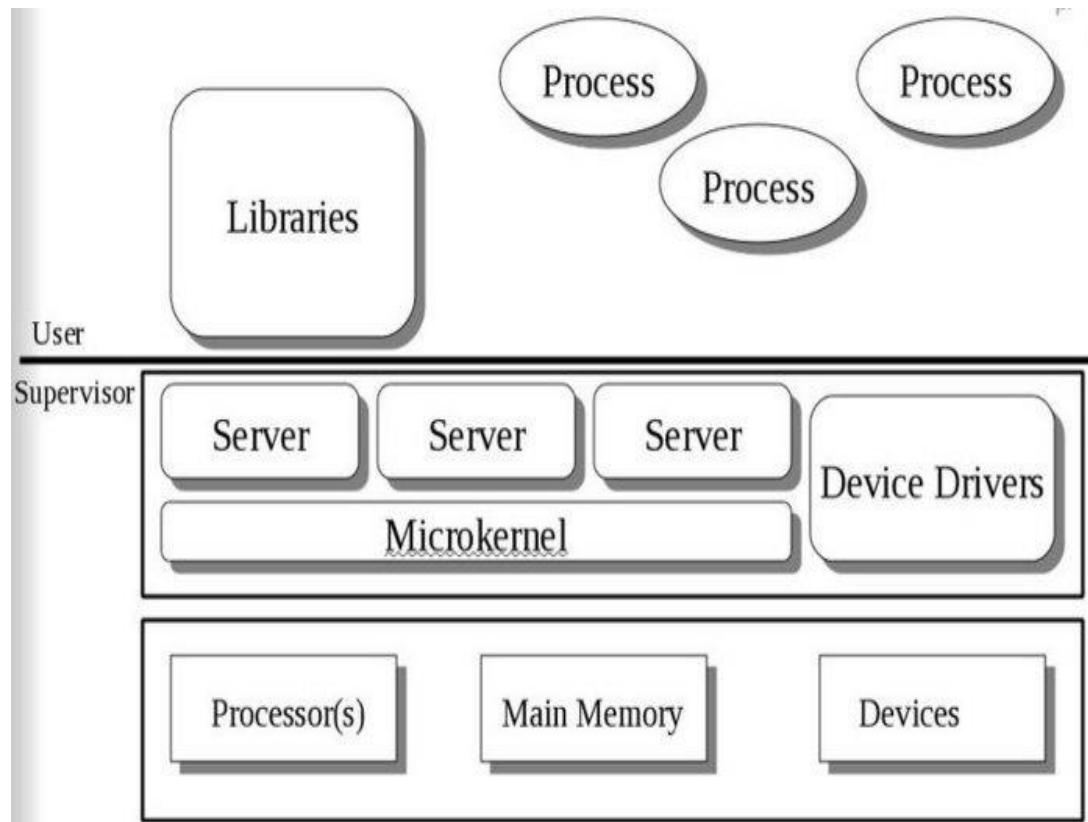
This structures the operating system by removing all nonessential portions of the kernel and implementing them as system and user level programs.

- Generally they provide minimal process and memory management, and a communications facility.
- Communication between components of the OS is provided by message passing.

The *benefits* of the microkernel are as follows:

- Extending the operating system becomes much easier.
- Any changes to the kernel tend to be fewer, since the kernel is smaller.
- The microkernel also provides more security and reliability.

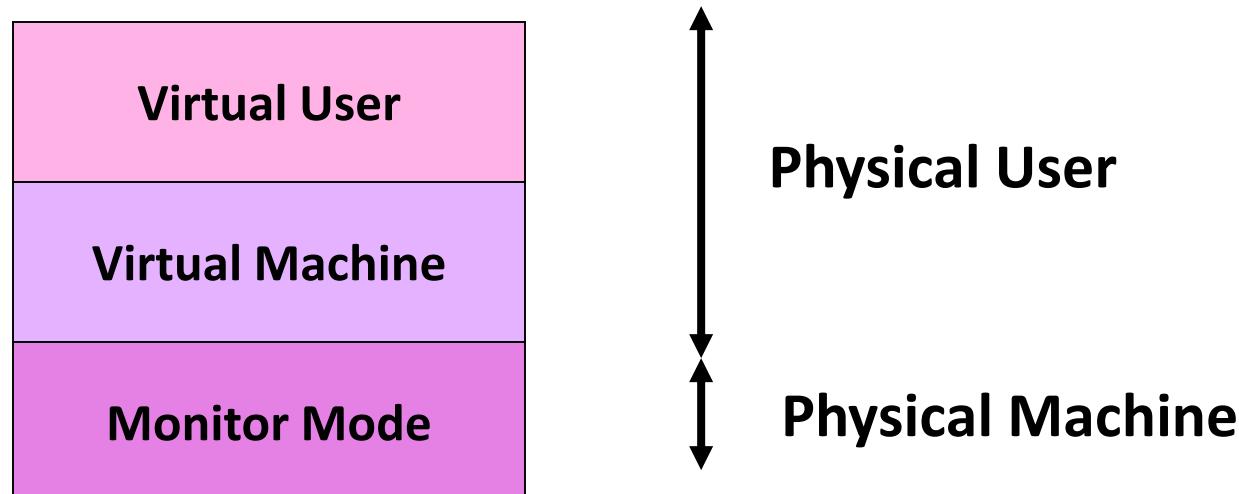
Main *disadvantage* is poor performance due to increased system overhead from message passing.



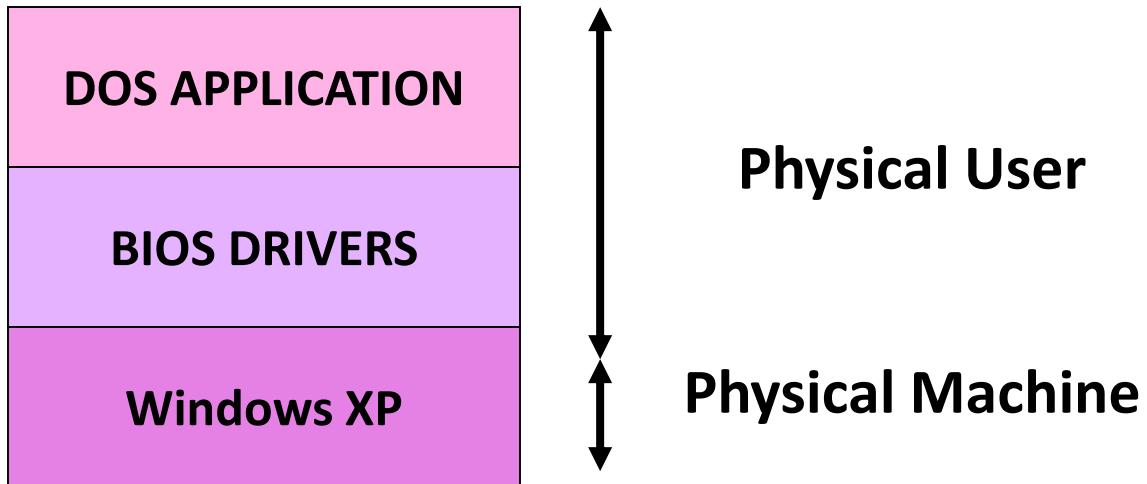
# Virtual Machine

In a Virtual Machine - each process "seems" to execute on its own processor with its own memory, devices, etc.

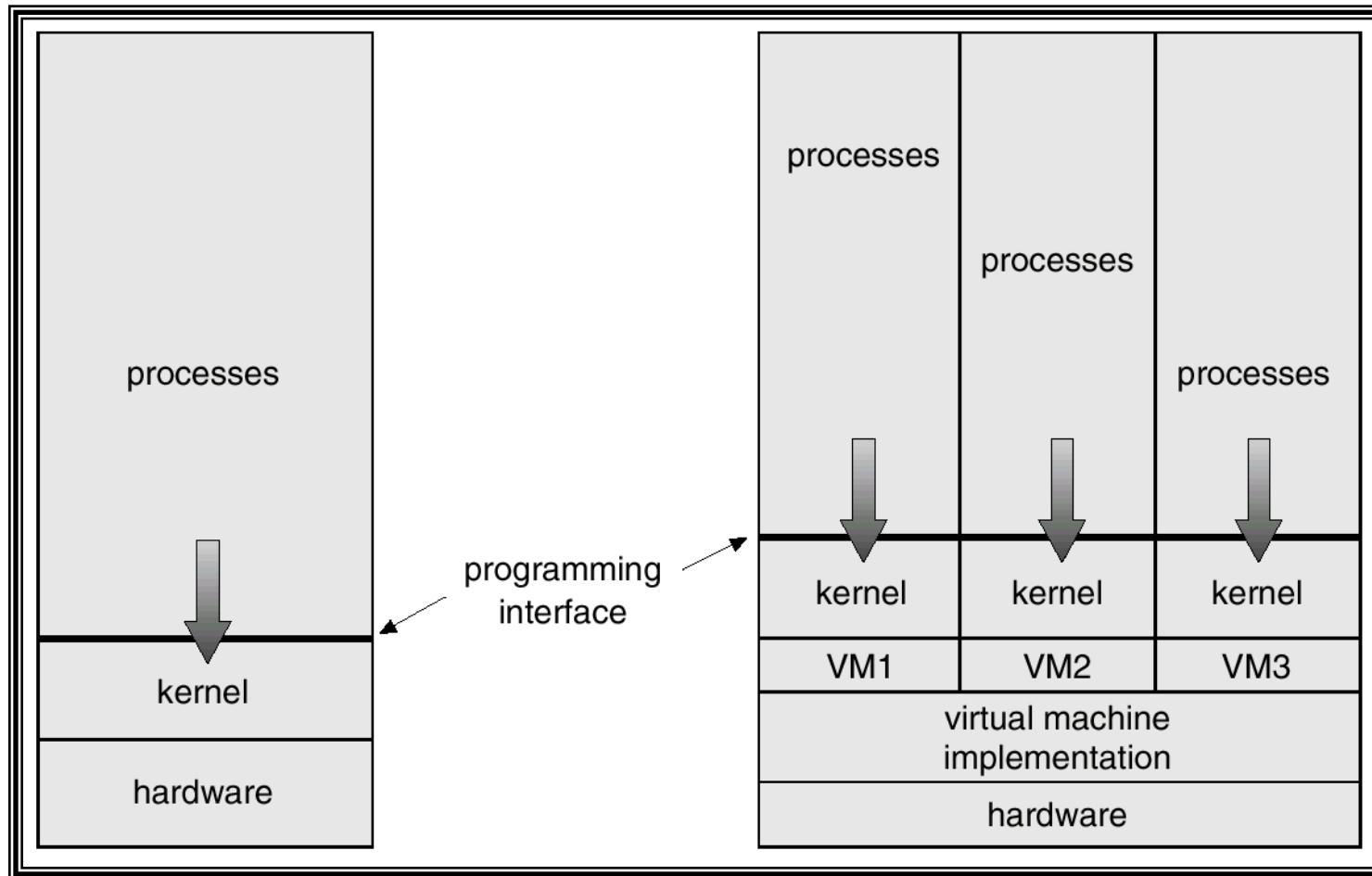
- The resources of the physical machine are shared. Virtual devices are sliced out of the physical ones. Virtual disks are subsets of physical ones.
- Useful for running different OS simultaneously on the same machine.
- Protection is excellent, but no sharing possible.
- Virtual privileged instructions are trapped.



# Example of MS-DOS on top of Windows XP.

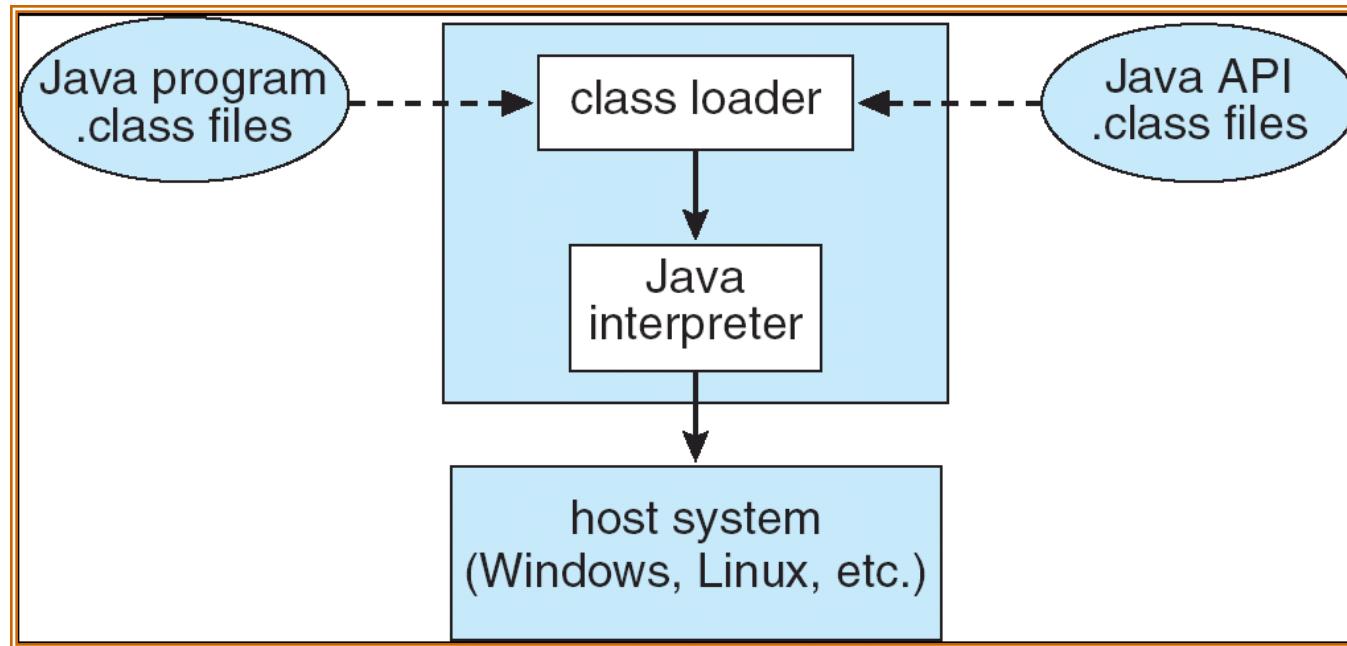


# Virtual Machine



# Example of Java Virtual Machine

- The Java Virtual Machine allows Java code to be portable between various hardware and OS platforms.



# Computer System Operation

# System Startup

- On power up
  - everything in system is in random, unpredictable state
  - special hardware circuit raises RESET pin of CPU
    - sets the program counter to 0xffffffff0
      - this address is mapped to ROM (Read-Only Memory)
- BIOS (Basic Input/Output Stream)
  - set of programs stored in ROM
  - some OS's use only these programs
    - MS DOS
  - many modern systems use these programs to load other system programs
    - Windows, Unix, Linux

# BIOS

- General operations performed by BIOS
  - 1) find and test hardware devices
    - POST (Power-On Self-Test)
  - 2) initialize hardware devices
    - creates a table of installed devices
  - 3) find *boot sector*
    - may be on floppy, hard drive, or CD-ROM
  - 4) load boot sector into memory location 0x00007c00
  - 5) sets the program counter to 0x00007c00
    - starts executing code at that address

# Boot Loader

- Small program stored in boot sector
- Loaded by BIOS at location 0x00007c0
- Configure a basic file system to allow system to read from disk
- Loads kernel into memory
- Also loads another program that will begin kernel initialization

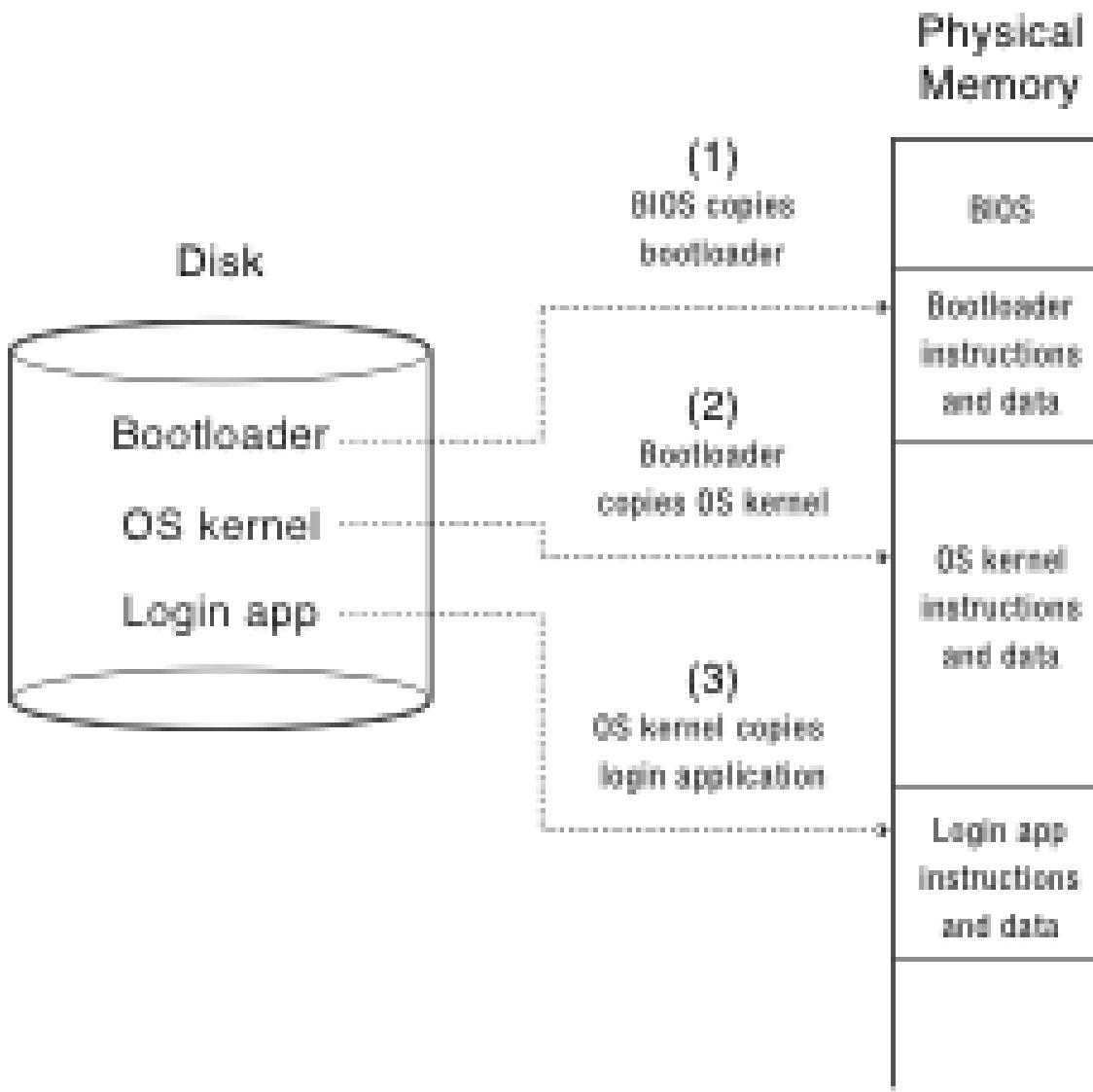
# Initial Kernel Program

- Determines amount of RAM in system
  - uses a BIOS function to do this
- Configures hardware devices
  - video card, mouse, disks, etc.
  - BIOS may have done this but usually redo it
    - portability
- Switches the CPU from *real* to *protected* mode
  - real mode: fixed segment sizes, 1 MB memory addressing, and no segment protection
  - protected mode: variable segment sizes, 4 GB memory addressing, and provides segment protection
- Initializes paging (virtual memory)

# Final Kernel Initialization

- Sets up page tables and segment descriptor tables
  - these are used by virtual memory and segmentation hardware
- Sets up interrupt vector and enables interrupts
- Initializes all other kernel data structures ( Linked lists, Binary search trees, Bitmaps etc.)
- Creates initial process and starts it running
  - *init* in Linux
  - *smss* (Session Manager SubSystem) in NT

# Booting



# System Programs

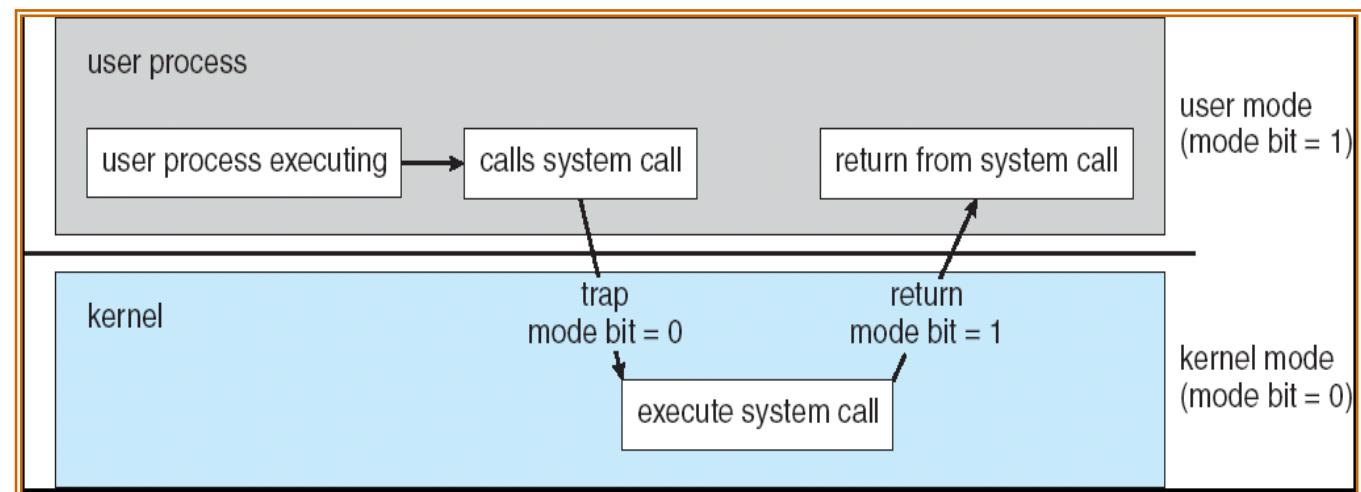
- Application programs included with the OS
- Highly trusted programs
- Perform useful work that most users need
  - listing and deleting files, configuring system
  - ls, rm, Windows Explorer and Control Panel
  - may include compilers and text editors
- Not part of the OS
  - run in user space
- Very useful

# Operating-System Operations

- Interrupt driven by hardware
- Software error or request creates **exception** or **trap**
  - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
- Hardware provides at least two modes:
  - “Kernel” mode (or “supervisor” or “protected”)
  - “User” mode: Normal programs executed
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user

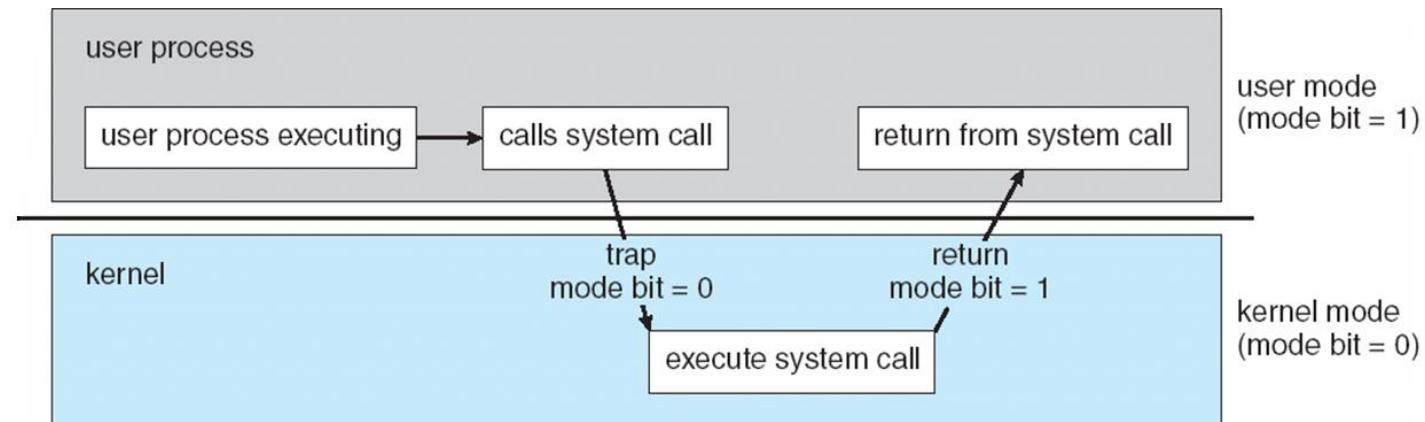
# Dual Mode Operation

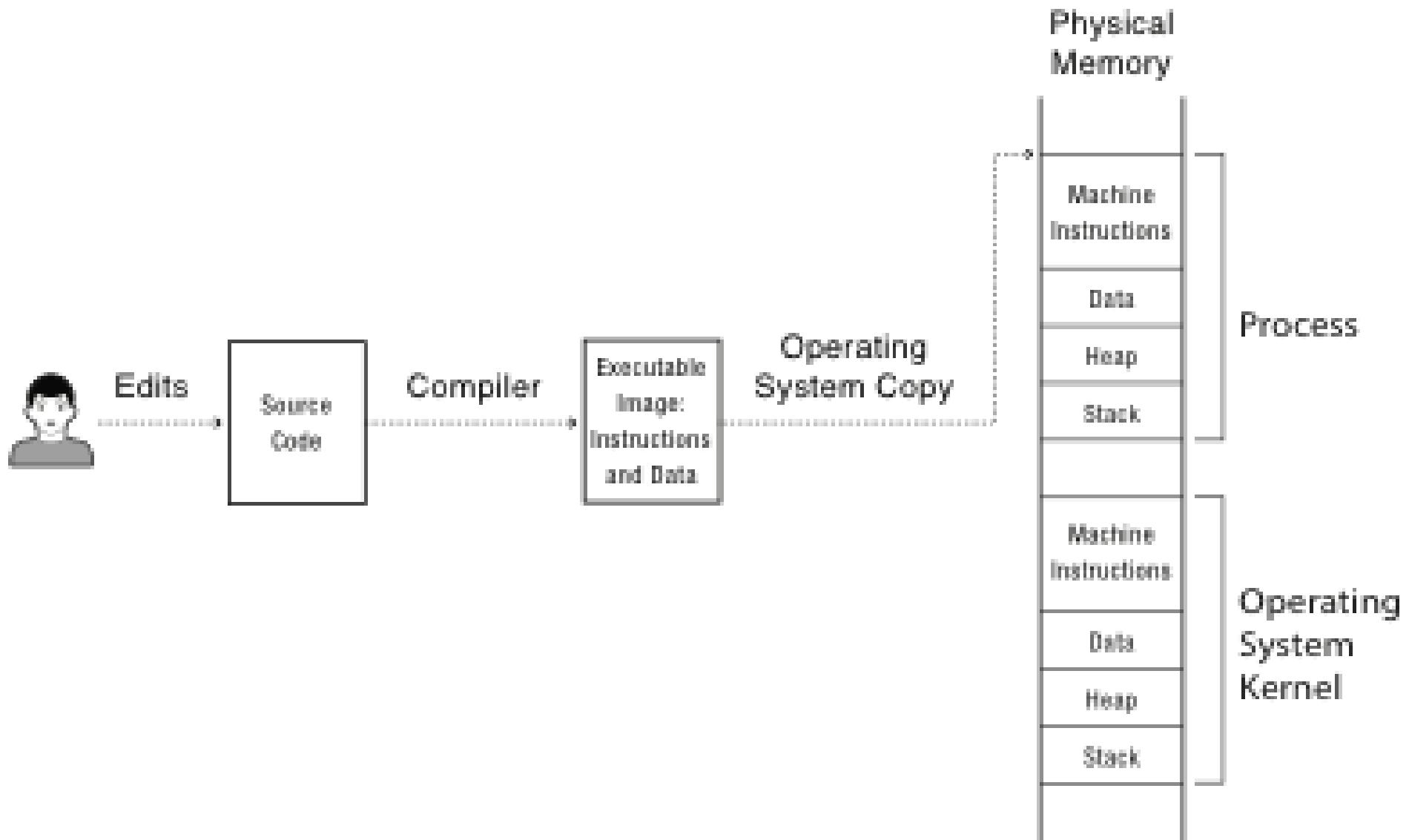
- Some instructions/ops prohibited in user mode:
  - Example: cannot modify page tables in user mode
    - Attempt to modify  $\Rightarrow$  Exception generated
- Transitions from user mode to kernel mode:
  - System Calls, Interrupts, Other exceptions



# Transition from User to Kernel Mode

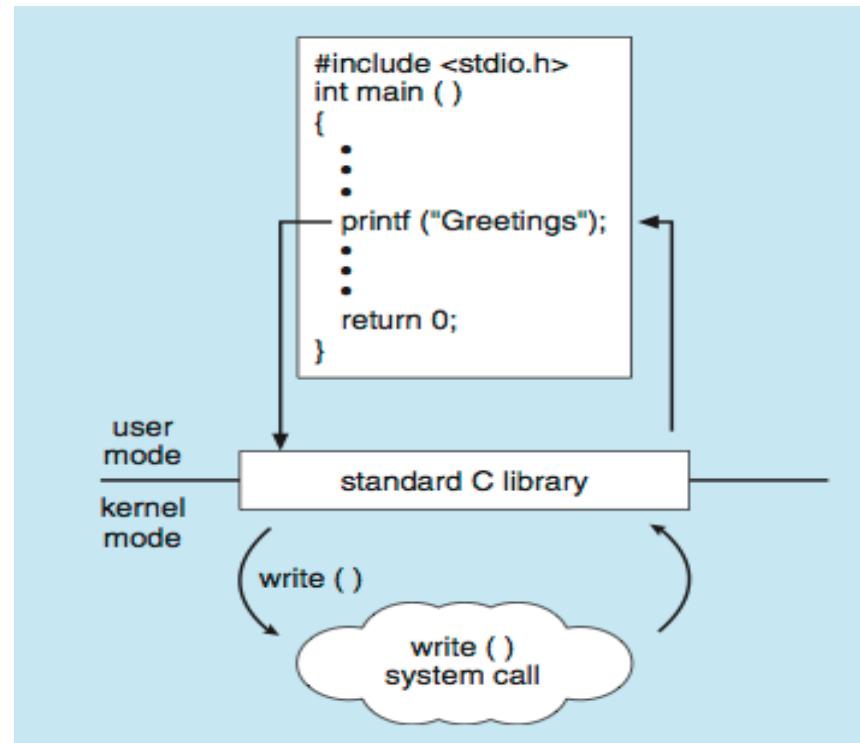
- Timer to prevent infinite loop / process hogging resources
  - Set interrupt after specific period
  - Operating system decrements counter
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time





# Standard C Library Example

- C program invoking printf() library call, which calls write() system call



# System Calls

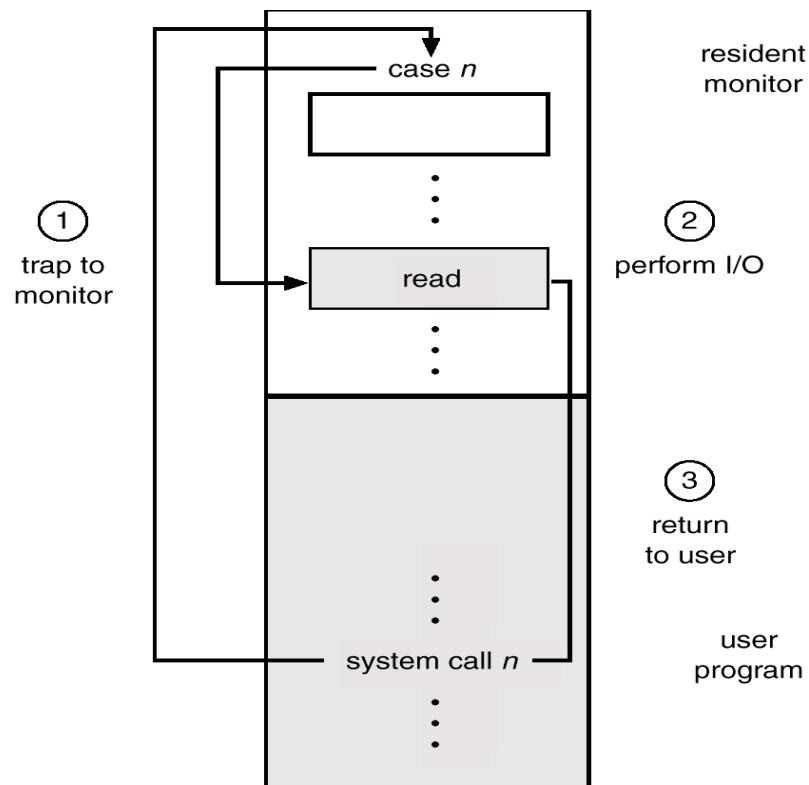
- System calls provide the interface between a running program and the operating system.
  - Generally available as assembly-language instructions.
  - Languages have been defined to replace assembly language for systems programming; allow system calls to be made directly (e.g., C, C++)
- Three general methods are used to pass parameters between a running program and the operating system.
  - Pass parameters in *registers*.
  - Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
  - *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system.

# System Calls

- System calls are routines run by the OS on behalf of the user
- Allow user to access I/O, create processes, get system information, etc.
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

# System Calls

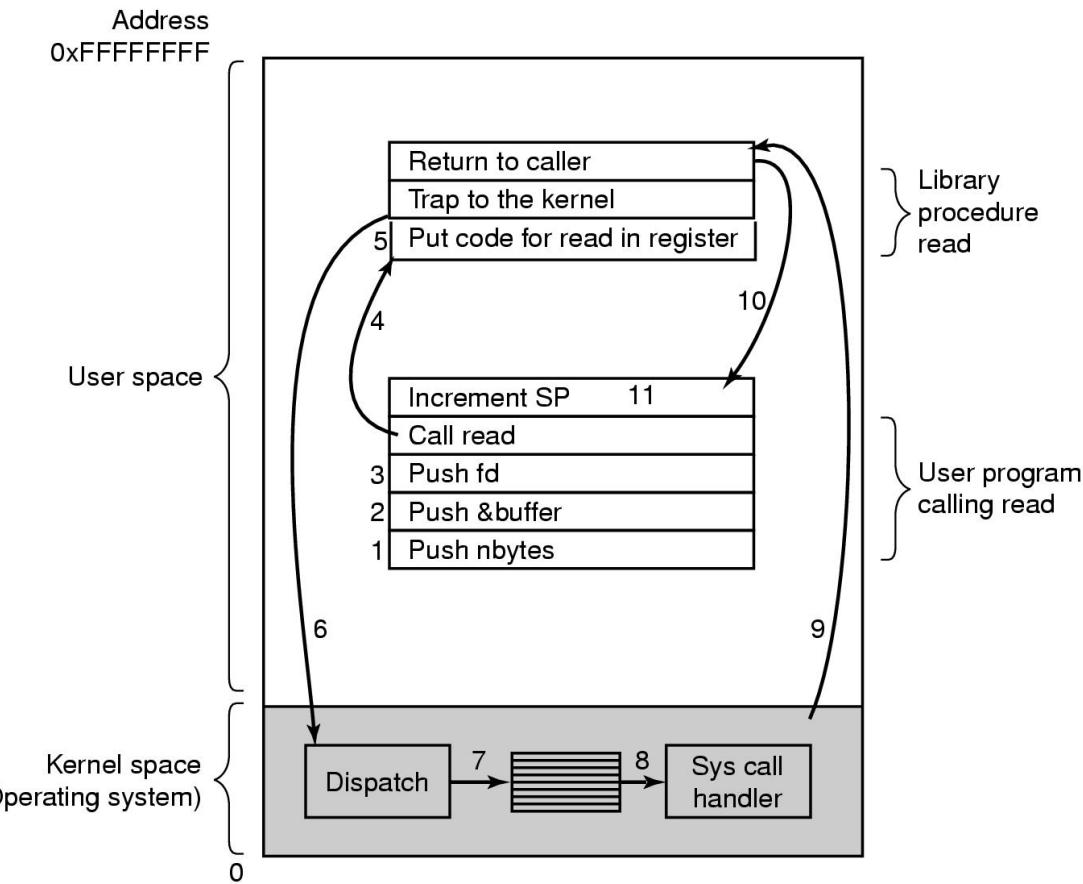
- A System Call is the main way a user program interacts with the Operating System.



# System Calls

## HOW A SYSTEM CALL WORKS

- Obtain access to system space
- Do parameter validation
- System resource collection ( locks on structures )
- Ask device/system for requested item
- Suspend waiting for device
- Interrupt makes this thread ready to run
- Wrap-up
- Return to user



There are 11 (or more) steps in making the system call  
**read (fd, buffer, nbytes)**

Linux API

# System Call Implementation

- 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)

# Example of Standard API

## EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t      read(int fd, void *buf, size_t count)
```

return value      function name      parameters

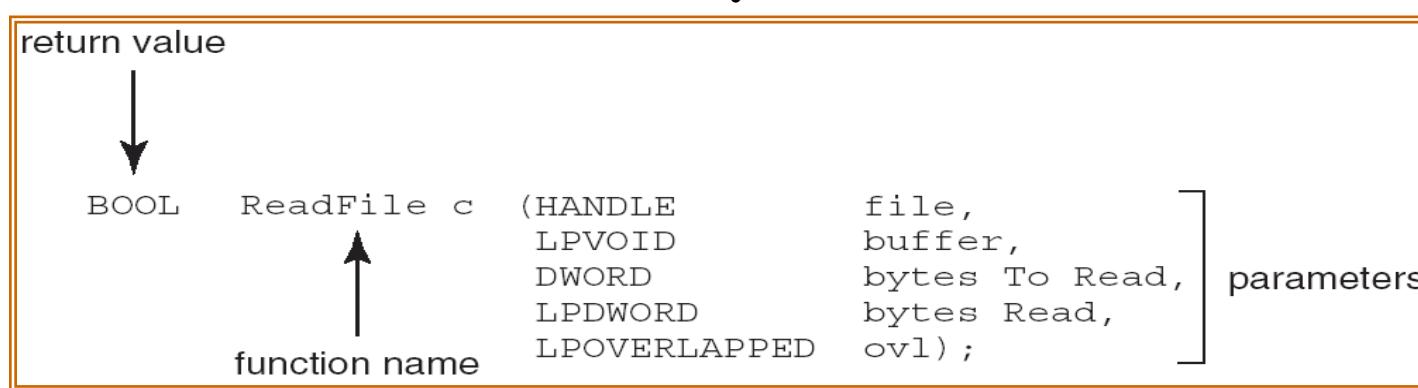
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

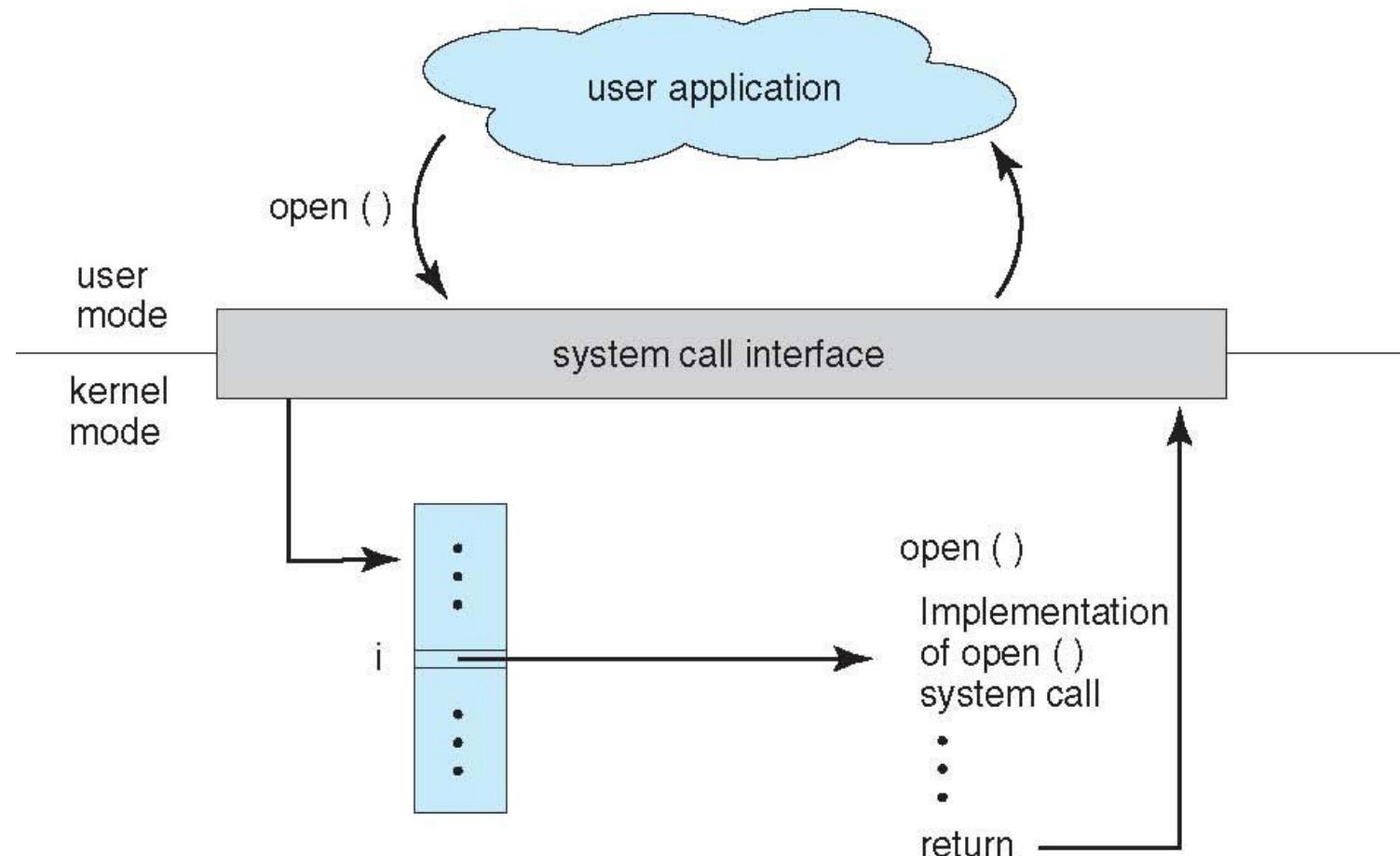
# Example of Windows API

- Consider the ReadFile() function in the program
- Win32 API—a function for reading from a file.



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

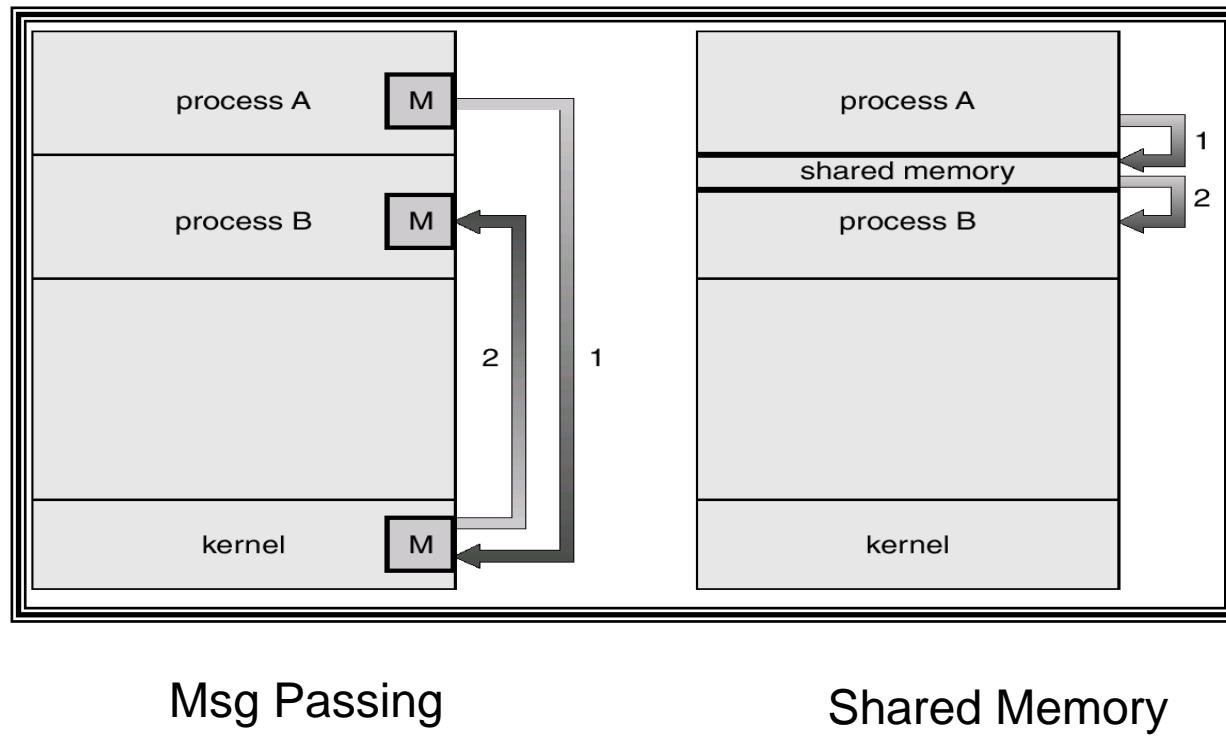
# API – System Call – OS Relationship



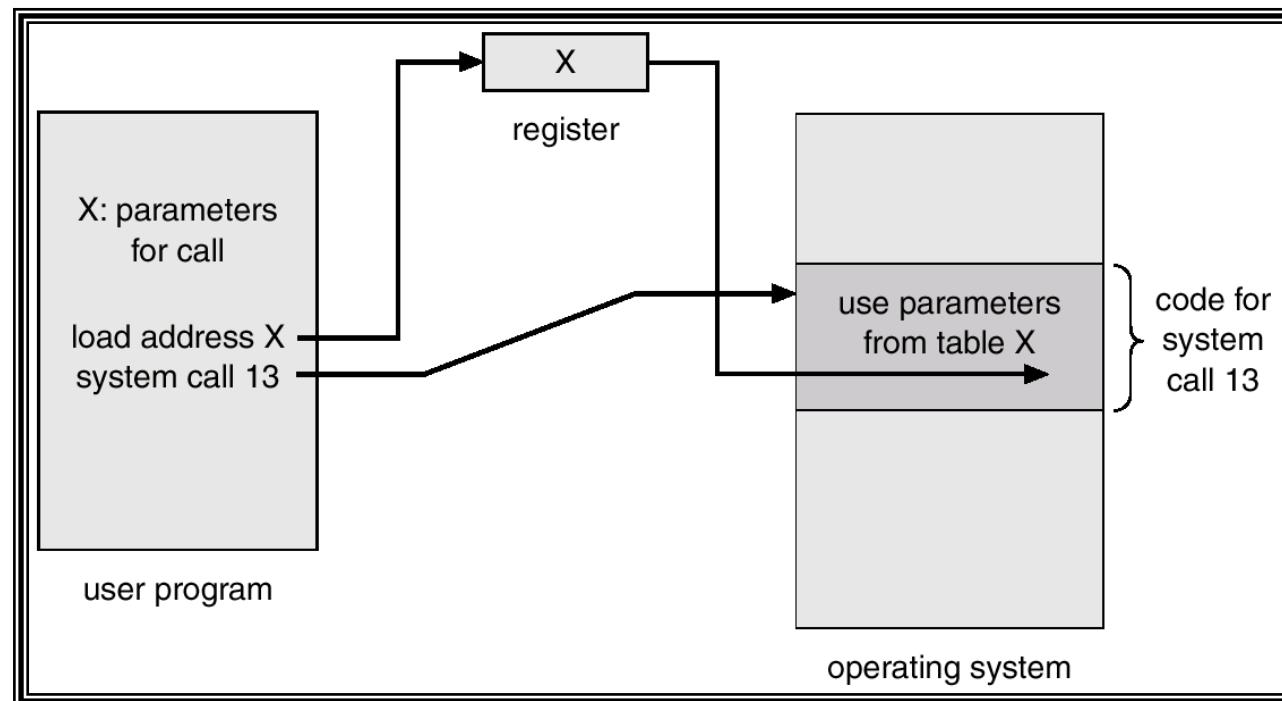
# System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

# passing data between programs

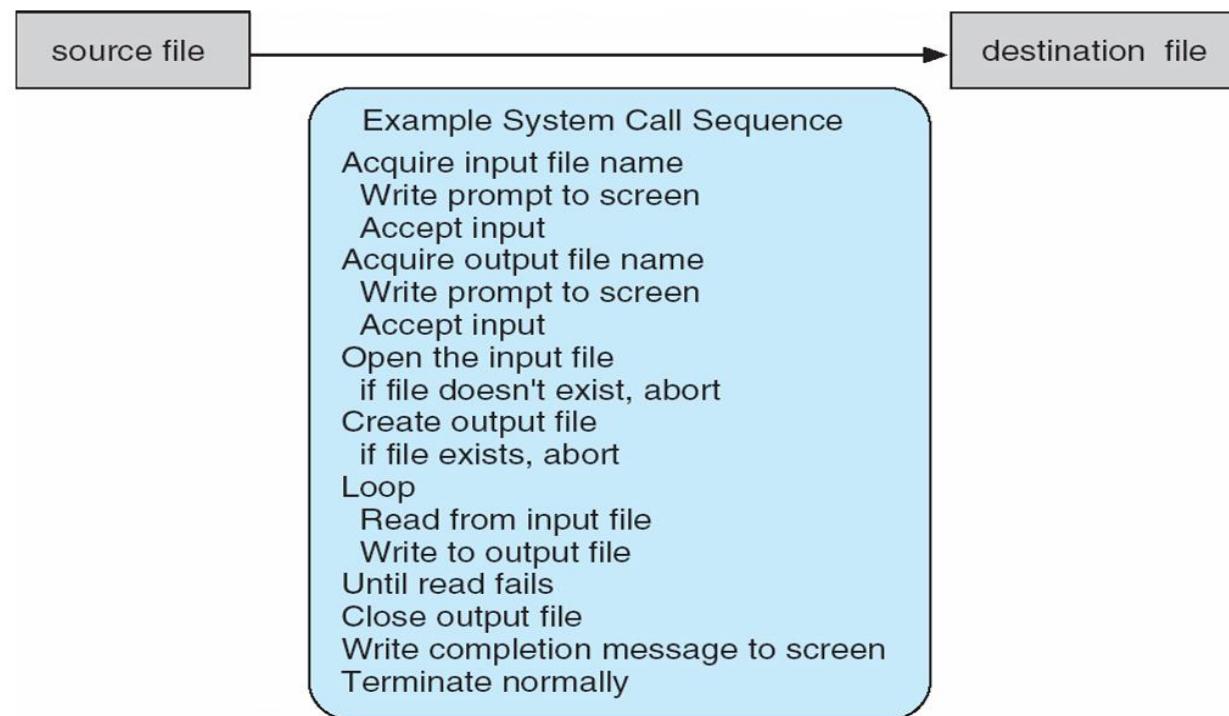


# Passing of Parameters As A Table



# Example of System Calls

- System call sequence to copy the contents of one file to another file



# Send message from one processor to another

Operations to be performed:

- Check Permissions, Format Message

- Enforce forward progress,

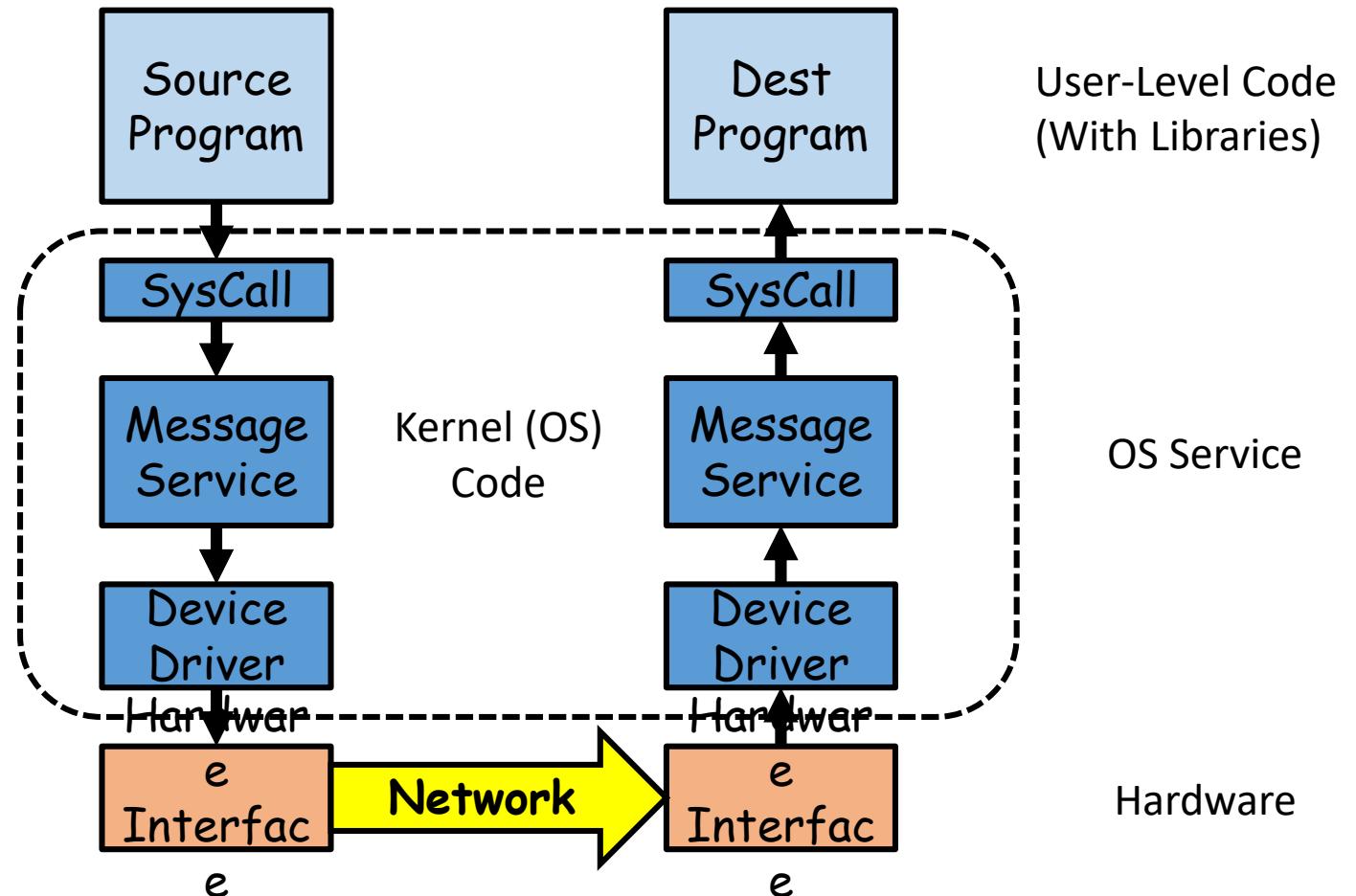
Handle interrupts

- Prevent Denial Of Service (DOS)

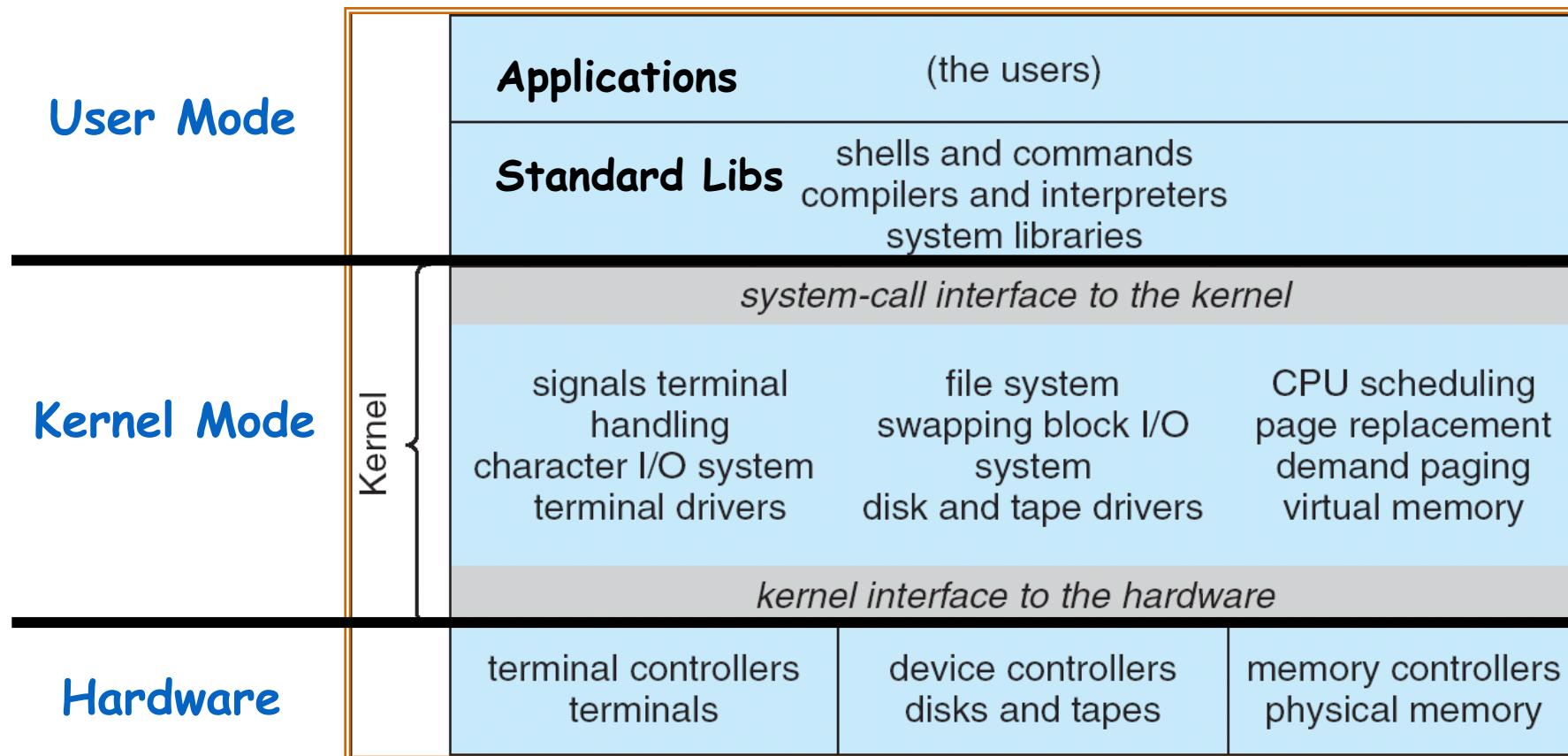
and/or Deadlock

Traditional Approach:

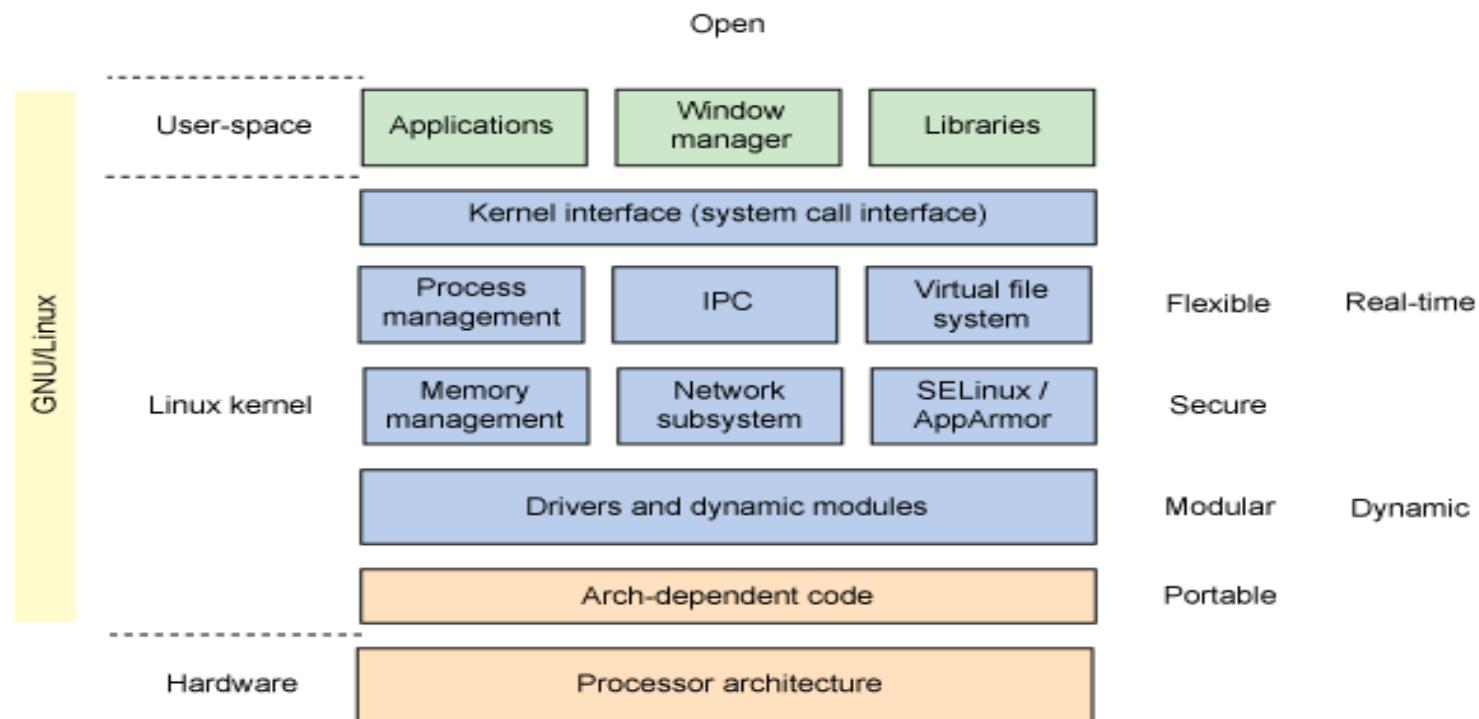
Use a system call + OS Service



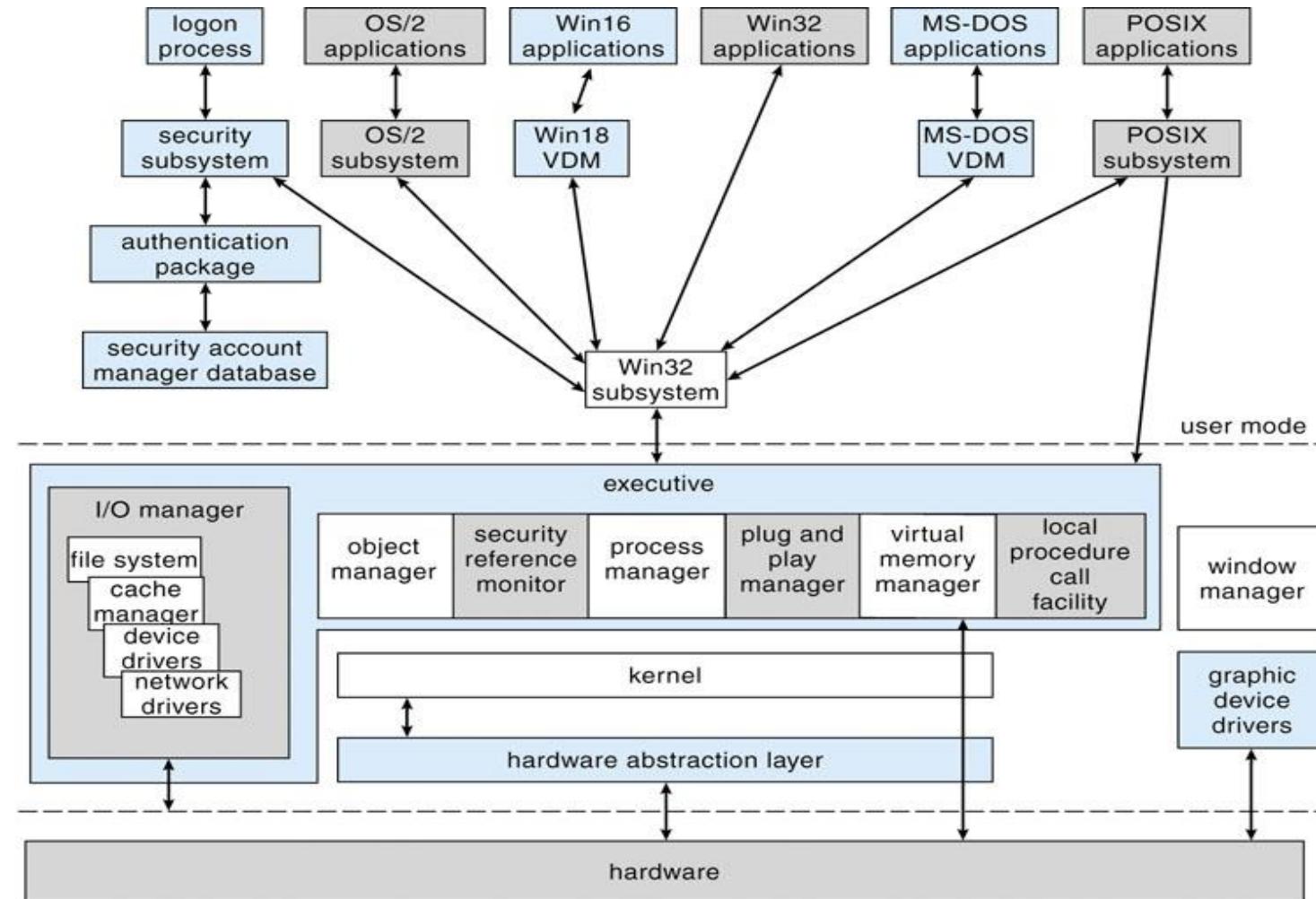
# UNIX System Structure



# Linux Structure



# Microsoft Windows Structure



# Major Windows Components

- Hardware Abstraction Layer
  - Hides hardware chipset differences from upper levels of OS
- Kernel Layer
  - Thread Scheduling
  - Low-Level Processors Synchronization
  - Interrupt/Exception Handling
  - Switching between User/Kernel Mode.
- Executive
  - Set of services that all environmental subsystems need
    - Object Manager
    - Virtual Memory Manager
    - Process Manager
    - Advanced Local Procedure Call Facility
    - I/O manager
    - Cache Manager
    - Security Reference Monitor
    - Plug-and-Plan and Power Managers
    - Registry
    - Booting
- Programmer Interface: Win32 API

# Types of System Calls

- Process control
- File manipulation
- Device manipulation
- Information maintenance
- Communications
- Protection

# Types of System Calls

- 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

# Types of System Calls

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

# Types of System Calls

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

# Types of System Calls

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

# Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

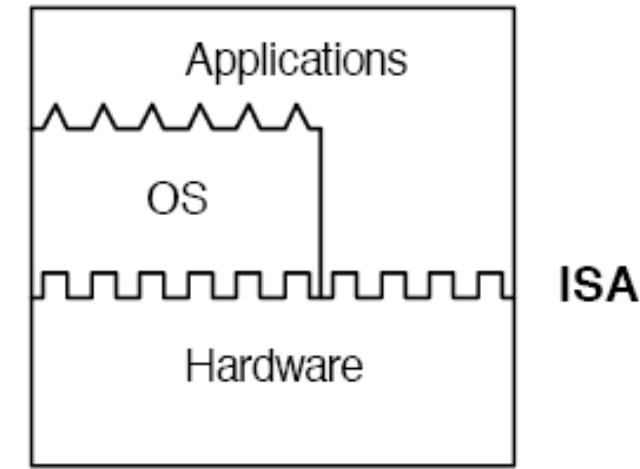
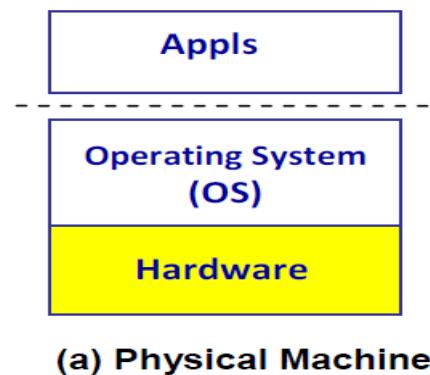
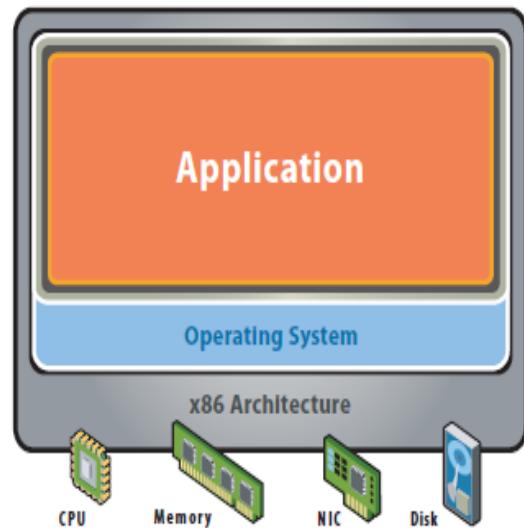
<b>UNIX</b>	<b>Win32</b>	<b>Description</b>
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

# Initial Hardware Model

The host machine is equipped with the physical hardware

e.g. a desktop with x-86 architecture running its installed Windows OS

All applications access hardware resources (i.e. memory, i/o) through system calls to operating system (privileged instructions)



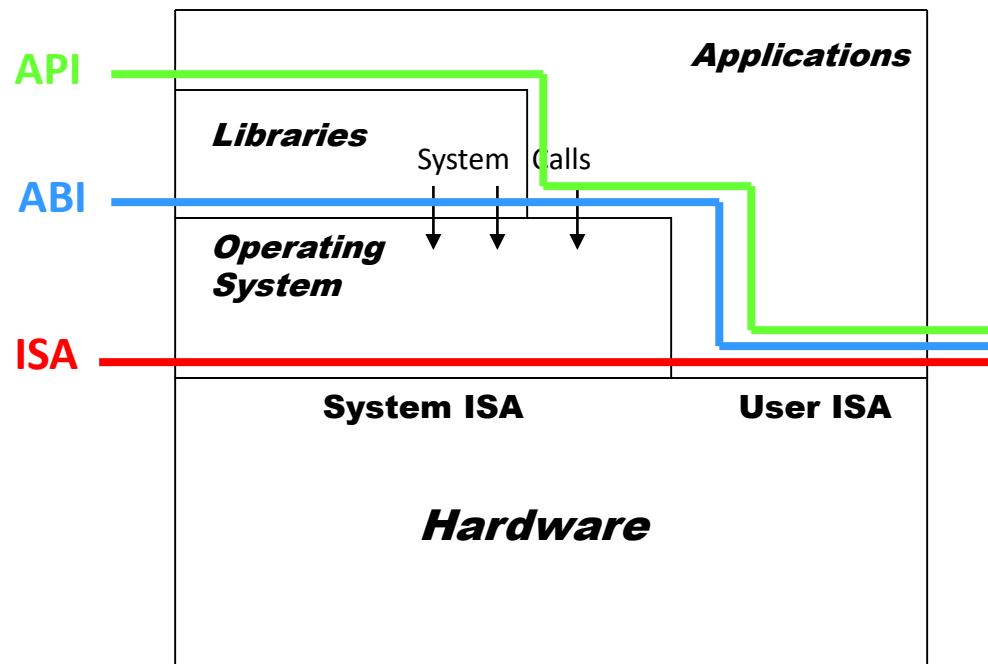
- Single OS image per machine
- Software and hardware tightly coupled
- Running multiple applications on same machine
- Underutilized resources
- Inflexible and costly infrastructure

- Advantages
  - Design is decoupled (i.e. OS people can develop OS separate of Hardware people developing hardware)
  - Hardware and software can be upgraded without notifying the Application programs

- Disadvantages
  - Application compiled on one ISA will not run on another ISA..
    - Applications compiled for Mac use different operating system calls than application designed for windows.
  - ISA's must support old software
    - Can often be inhibiting in terms of performance
  - Since software is developed separately from hardware..
  - Software is not necessarily optimized for hardware.

# Architecture & Interfaces

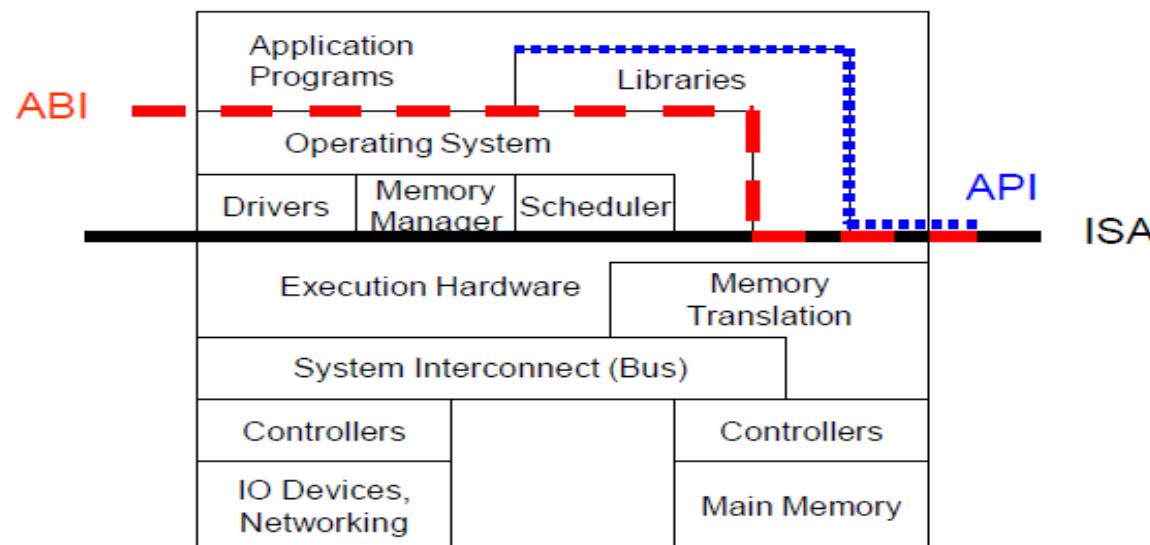
Architecture: formal specification of a system's interface and the logical behavior of its visible resources.



- **API** – Application Programming Interface
- **ABI** – Application Binary Interface
- **ISA** – Instruction Set Architecture

# Architecture, Implementation Layers

- Architecture
  - Functionality and Appearance of a computer system but not implementation details
  - Level of Abstraction = Implementation layer ISA, ABI, API



# Architecture, Implementation Layers

- Implementation Layer : ISA
  - Instruction Set Architecture
  - Divides hardware and software
  - Concept of ISA originates from IBM 360
    - Various prices, processing power, processing unit, devices
    - But guarantee a *software compatibility*
  - User ISA and System ISA

# Architecture, Implementation Layers

- Implementation Layer : ABI
  - Application Binary Interface
  - Provides a program with access to the hardware resource and services available in a system
  - Consists of User ISA and System Call Interfaces

# Architecture, Implementation Layers

- Implementation Layer : API
  - Application Programming Interface
  - Key element is Standard Library ( or Libraries )
  - Typically defined at the source code level of High Level Language
  - **libc** in Unix environment : supports the UNIX/C programming language

# Multimode System

- The concept of modes of operation in operating system can be extended beyond the dual mode. This is known as the multimode system. In those cases more than 1 bit is used by the CPU to set and handle the mode.
- An example of the multimode system can be described by the systems that support virtualization. These CPU's have a separate mode that specifies when the virtual machine manager (VMM) and the virtualisation management software is in control of the system.
- For these systems, the virtual mode has more privileges than user mode but less than kernel mode.

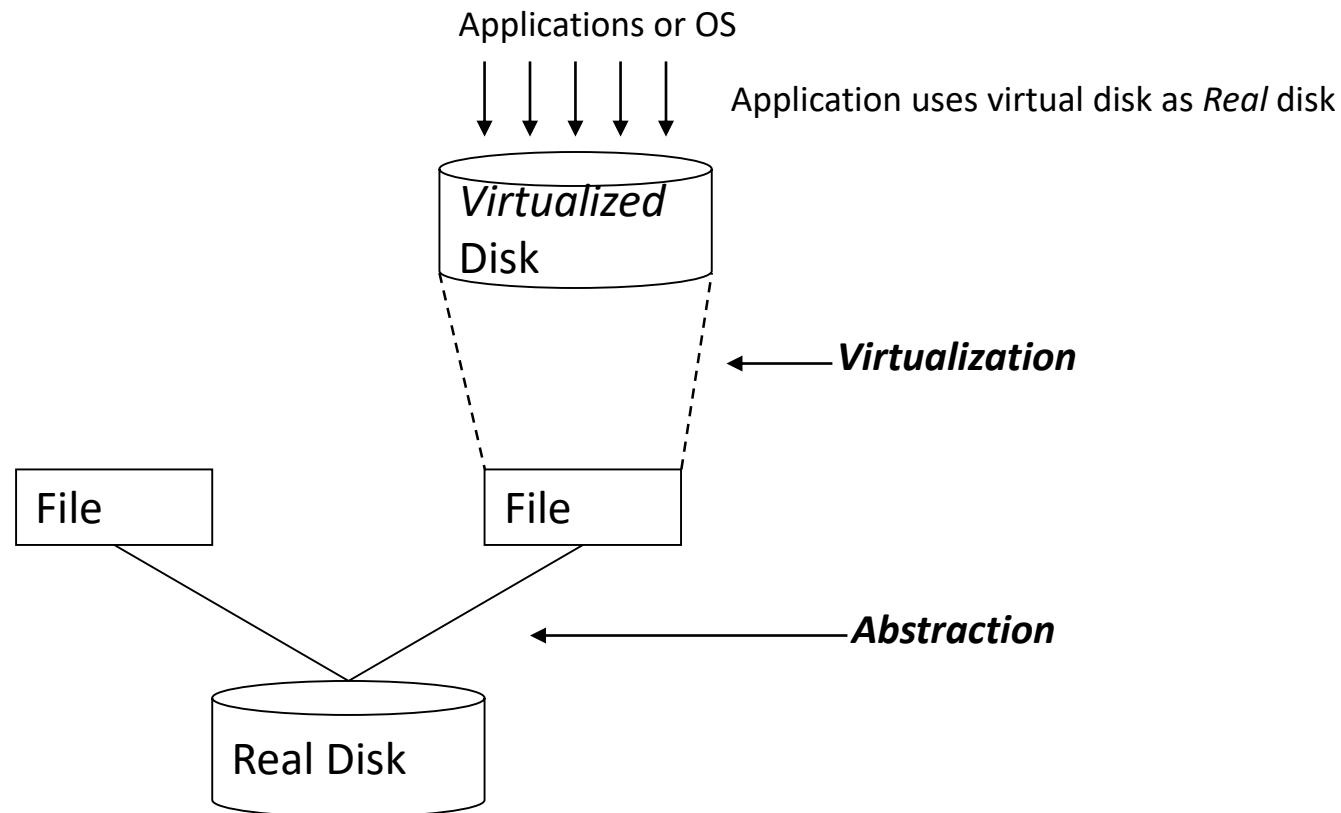
# What is virtualization?

- **Virtualization** -- the abstraction of computer resources.
- Virtualization hides the physical characteristics of computing resources from their users, be they applications, or end users.
- This includes making a single physical resource (such as a server, an operating system, an application, or storage device) appear to function as multiple virtual resources; it can also include making multiple physical resources (such as storage devices or servers) appear as a single virtual resource.

# Virtualization

- Similar to Abstraction but doesn't always hide low layer's details
- Real system is transformed so that it appears to be different
- Virtualization can be applied not only to subsystem, but to an *Entire Machine*  
→ **Virtual Machine**

# Virtualization



- **Virtualization**, in computing, is the creation of a virtual (rather than actual) version of something, such as a hardware platform, operating system, storage device, or network resources.
- Virtualization is the process by which one computer hosts the appearance of many computers.
- It is used to improve IT throughput and costs by using physical resources as a pool from which virtual resources can be allocated.

# VIRTUALIZATION BENEFITS

- Sharing of resources helps cost reduction
- Isolation: Virtual machines are isolated from each other as if they are physically separated
- Encapsulation: Virtual machines encapsulate a complete computing environment
- Hardware Independence: Virtual machines run independently of underlying hardware
- Portability: Virtual machines can be migrated between different hosts.
- Cross platform compatibility
- Increase Security
- Enhance Performance
- Simplify software migration

# What is “*Machine*”?

- 2 perspectives
- From the perspective of a process
  - ABI provides interface between process and machine
- From the perspective of a system
  - Underlying hardware itself is a machine.
  - ISA provides interface between system and machine

# System/Process Virtual Machines

Can view virtual machine as:

- Process virtual machine
  - Virtual machines can be instantiated for a single program (i.e. similar to Java)
  - Virtual machine terminates when process terminates.
- System virtual machine (i.e. similar to cygwin)
  - Full execution environment that can support multiple processes
  - Support I/O devices
  - Support GUI

# Virtual Machines

- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface *identical* to the underlying bare hardware.
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.

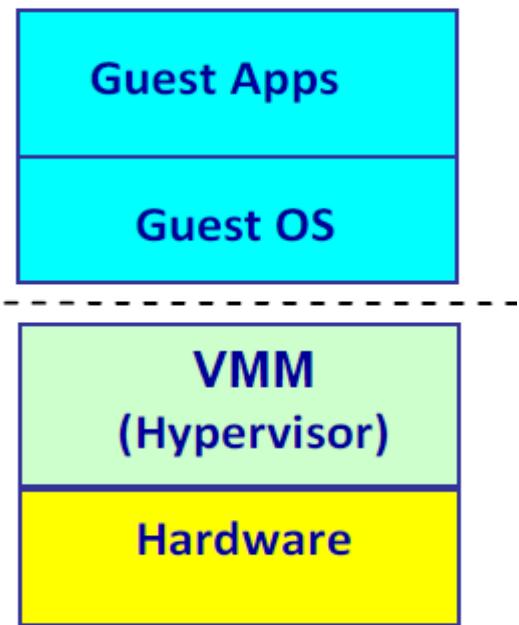
- The resources of the physical computer are shared to create the virtual machines.
  - CPU scheduling can create the appearance that users have their own processor.
  - Spooling (simultaneous peripheral operations online) and a file system can provide virtual card readers and virtual line printers.
  - A normal user time-sharing terminal serves as the virtual machine operator's console.

# VM types

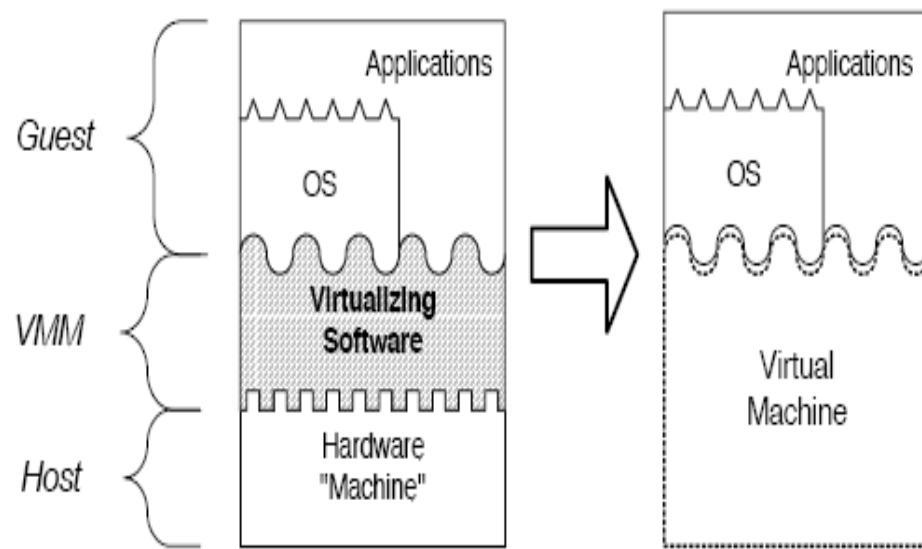
- Native VM
- Hosted VM
- Dual mode VM

# Native VM

- The VM can be provisioned to any hardware system.
- Virtual software placed between underlying machine and conventional software
  - Conventional software sees different ISA from the one supported by the hardware
- The VM is built with virtual resources managed by a guest OS to run a specific application. Between the VMs and the host platform, we need to deploy a middleware layer called a *Virtual Machine Monitor (VMM)* .



**(b) Native VM**

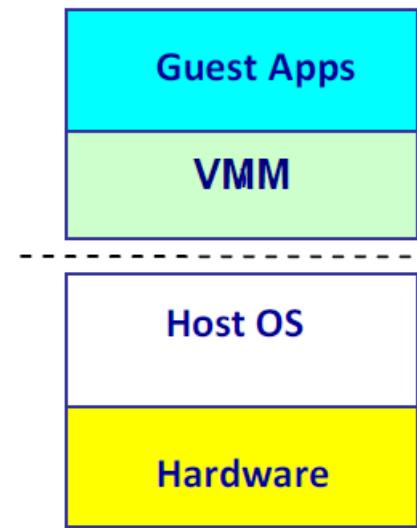


- Figure b shows a native VM installed with the use a VMM called a *hypervisor at the privileged mode*.
- For example, the hardware has a x-86 architecture running the Windows system. The guest OS could be a Linux system and the hypervisor is the XEN system developed at Cambridge University.
- This hypervisor approach is also called bare-metal VM, because the hypervisor handles the bare hardware (CPU, memory, and I/O) directly.

- Virtualization process involves:
  - Mapping of virtual resources (registers and memory) to real hardware resources
  - Using real machine instructions to carry out the actions specified by the virtual machine instructions

# Hosted VM

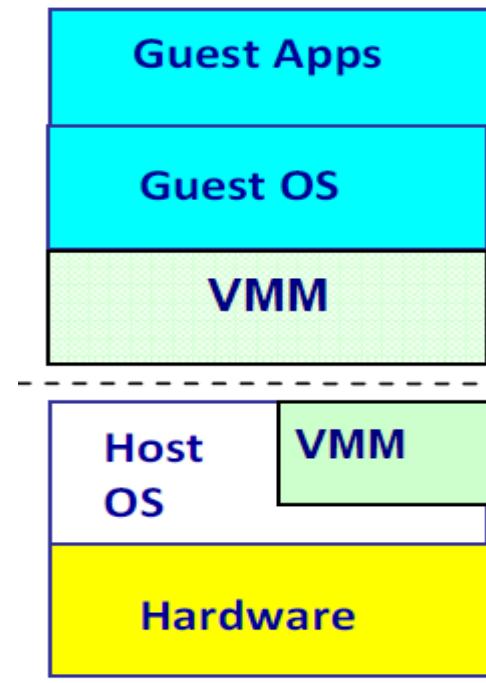
Another architecture is the host VM shown in Fig. (c). Here the VMM runs with a non-privileged mode. The host OS need not be modified.



(c) Hosted VM

# Dual Mode VM

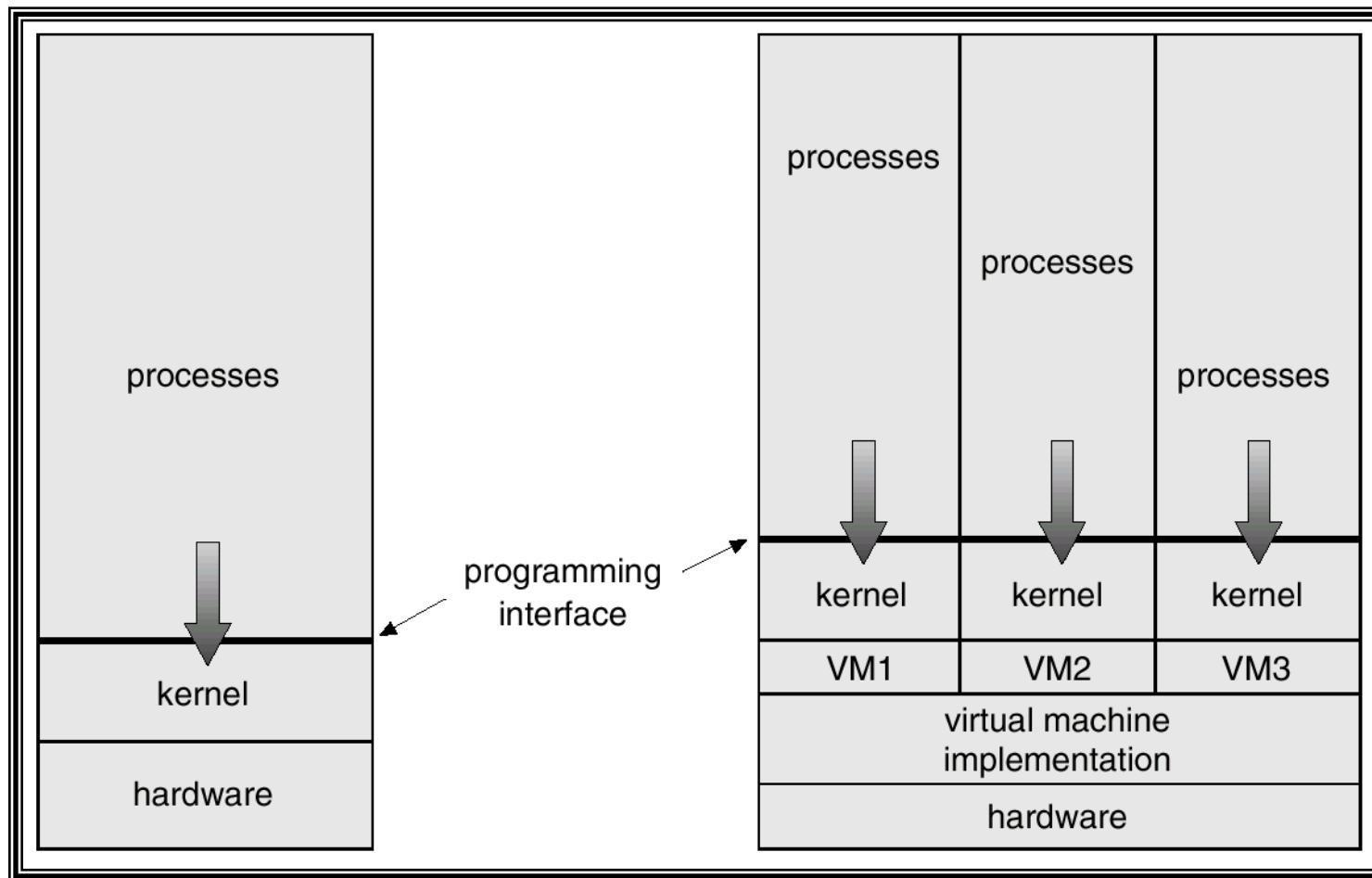
The VM can be also implemented with a dual mode as shown in Fig. (d).



(d) Dual-mode VM

- Part of VMM runs at the user level and another portion runs at the supervisor level. In this case, the host OS may have to be modified to some extent.
- Multiple VMs can be ported to one given hardware system, *to support the virtualization process*.
- *The VM approach offers hardware-independence of the OS and applications.*
- The user application and its dedicated OS could be bundled together as a virtual appliance, that can be easily ported on various hardware platforms.

# System Models



# Advantages/Disadvantages of Virtual Machines

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine.

# Process Management

# Process Concepts

- The notion of a process is fundamental to OS and it defines the fundamental unit of computation for the computer and used by OS for concurrent program execution.

What is a process?

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

The components of process are:

- Object Program: Code to be executed
- Data: Data used for executing the program
- Resources: Resources needed for execution of the program
- Status of the process execution: Used for verifying the status of the process execution.
- A process is allocated with resources such as memory and is available for scheduling. It can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources.

# Difference between a program and a process

A program is not the same as a process.

- A program is a static object which contains instructions that can exist in a file.
  - Program = static file (image)
- A process is dynamic object that is a program in execution.
  - Process = executing program = program + execution state.
- A program is a sequence of instructions whereas process is a sequence of instruction executions.
- A program exists at a single place in space and continues to exist as time goes forward whereas a process exists in a limited span of time.
- A program does not perform the action by itself whereas different processes may run different instances of the same program

# Running a program

A program consists of code and data

- On running a program, the loader:
  - reads and interprets the executable file
  - sets up the process's memory to contain the code & data from executable file
  - pushes “argc”, “argv” on the stack
  - sets the CPU registers properly & calls “`_start()`”

- Program starts running at `_start()`

```
_start(args) {  
    initialize_java();  
    ret = main(args);  
    exit(ret)  
}
```

- Now “process” is running, and no longer it is “program”
- When `main()` returns, OS calls “`exit()`” which destroys the process and returns all resources

A process includes: program counter; stack; data section

### Process image

- Collection of programs, data, stack, and attributes that form the process
- User data
  - Modifiable part of the user space
  - Program data, user stack area, and modifiable code
- User program
  - Executable code
- System stack
  - Used to store parameters and calling addresses for procedure and system calls
- Process control block
  - Data needed by the OS to control the process
- Location and attributes of the process
  - Memory management aspects: contiguous or fragmented allocation

# Keeping track of a process

- A process has code.
  - OS must track program counter (code location).
- A process has a stack.
  - OS must track stack pointer.
- OS stores state of processes' computation in a Process Control Block (PCB).
  - E.g., each process has an identifier (process identifier, or PID)
  - Data (program instructions, stack & heap) resides in memory, metadata is in PCB (which is a kernel data structure in memory)

# Implementation of a Process

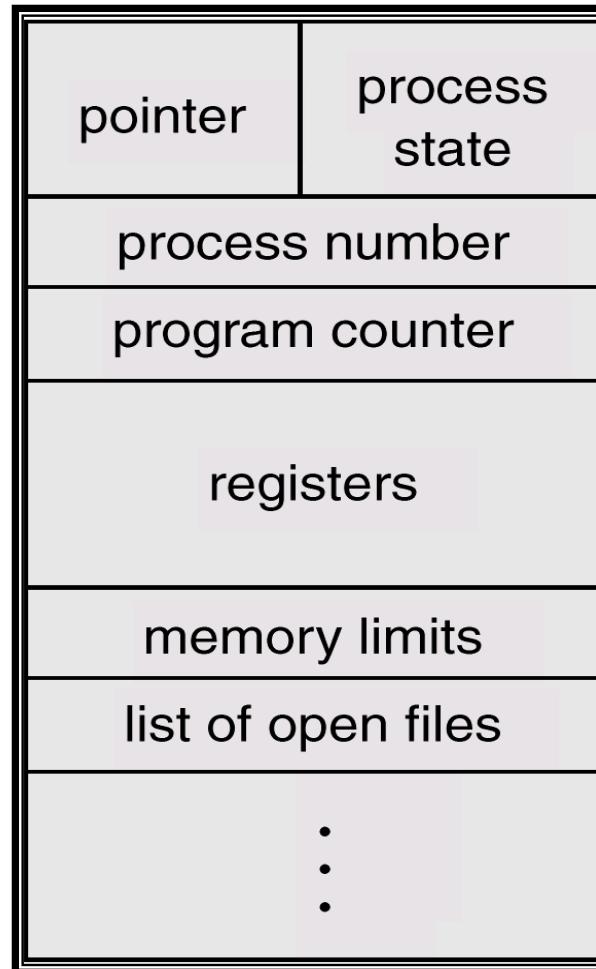
## Process Control Block (PCB)

- Each process contains the process control block (PCB). PCB is the data structure used by the OS and all information about a particular process such as Process id, process state, priority, privileges, memory management information, accounting information etc. are grouped by OS.
- PCB also includes the information about CPU scheduling, I/O resource management, file management information, priority and so on.
- The PCB simply serves as the repository for any information that may vary from process to process.

# Contents of PCB are

1. Pointer: Pointer points to another process control block. Pointer is used for maintaining the scheduling list.
2. Process id
3. Process State: Process state may be new, ready, running, waiting and so on.
4. Program Counter: It indicates the address of the next instruction to be executed for this process.
5. Event information: For a process in the blocked state this field contains information concerning the event for which the process is waiting.
6. CPU register: It indicates general purpose register, stack pointers, index registers and accumulators etc. number of register and type of register totally depends upon the computer architecture.
7. Memory Management Information: This information may include the value of base and limit register. This information is useful for deallocating the memory when the process terminates.
8. Accounting Information: This information includes the amount of CPU and real time used, time limits, job or process numbers, account numbers etc.

# Process Control Block (PCB)



- When a process is created, hardware registers and flags are set to the values provided by the loader or linker.
- Whenever that process is suspended, the contents of the processor register are usually saved on the stack and the pointer to the related stack frame is stored in the PCB.
- In this way, the hardware state can be restored when the process is scheduled to run again.

# Operations on Processes

- The OS as well as other processes can perform operations on a process. Several operations are possible on the process such as create, kill, signal, suspend, schedule, change priority, resume etc.
- OS must provide the environment for the process operations.
- Two main operations that are to be performed are :
  - process creation
  - process deletion

# Process Creation

OS creates the very first process when it initializes. That process then starts all other processes in the system using the create system calls.

OS creates a new process with the specified or default attributes and identifier.

A process may create several new sub-processes.

Syntax for creating new process is :

- CREATE ( processid, attributes )

- When OS issues a CREATE system call, it obtains a new process control block from the pool of free memory, fills the fields with provided and default parameters, and insert the PCB into the ready list.
- Thus it makes the specified process eligible for running.
- When a process is created, it requires some parameters. These are priority, level of privilege, requirement of memory, access right, memory protection information etc.
- Process will need certain resources, such as CPU time, memory, files and I/O devices to complete the operation.

## Process Hierarchy

- When one process creates another process, the creator is referred as parent and the created process is the child process. The Child process may create another process. So it forms a tree of processes which is referred as process hierarchy.
- When process creates a child process, that child process may obtain its resources directly from the OS, otherwise it uses the resources of parent process. Generally a parent is in control of a child process.

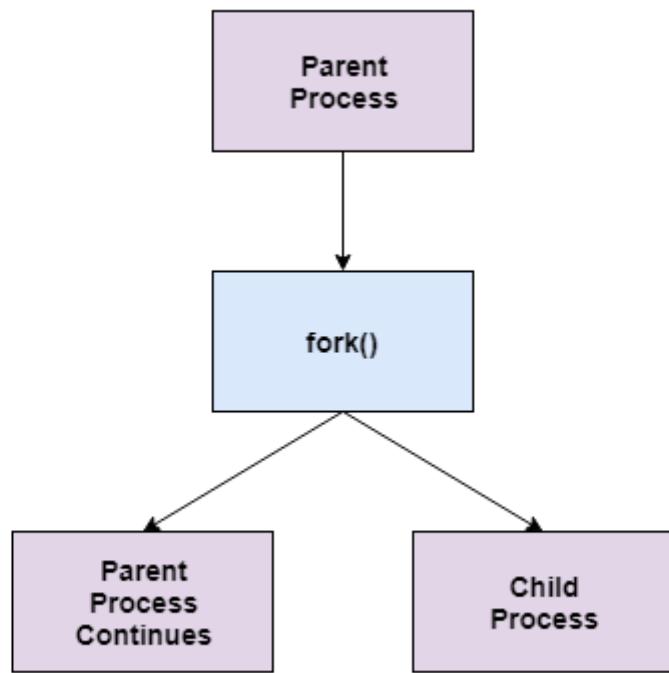
- In Operating System, the fork() system call is used by a process to create another process. The process that used the fork() system call is the parent process and process consequently created is known as the child process.

### **Parent Process**

- All the processes in operating system are created when a process executes the fork() system call except the startup process. The process that used the fork() system call is the parent process. In other words, a parent process is one that creates a child process. A parent process may have multiple child processes but a child process only one parent process.
- On the success of a fork() system call, the PID of the child process is returned to the parent process and 0 is returned to the child process. On the failure of a fork() system call, -1 is returned to the parent process and a child process is not created.

## **Child Process**

- A child process is a process created by a parent process in operating system using a fork() system call. A child process may also be called a subprocess or a subtask.
- A child process is created as its parent process's copy and inherits most of its attributes. If a child process has no parent process, it was created directly by the kernel.
- If a child process exits or is interrupted, then a SIGCHLD signal is send to the parent process.



When a process creates a new process, two possibilities exist in terms of execution.

1. Concurrent : The parent continues to execute concurrently with its children.

2. Sequential :The parent waits until some or all of its children have terminated.

- For address space, two possibilities:

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

- Resource sharing possibilities

1. Parent and child share all resources
2. Children share subset of parent's resources
3. Parent and child share no resources

# Process Termination

- DELETE system call is used for terminating a process.
- A process may delete itself or by another process. A process can cause the termination of another process via an appropriate system call.
- The OS reacts by reclaiming all resources allocated to the specified process, closing files opened by or for the process. PCB is also removed from its place of residence in the list and is returned to the free pool.
- The DELETE service is normally invoked as a part of orderly program termination. Parent may terminate execution of children processes (abort) if Child has exceeded allocated resources or Task assigned to the child is no longer required or parent is exiting.
- OS may not allow child to continue if its parent terminates. This may result in cascading termination.

**Zombie Process:**

A child process which has finished the execution but still has entry in the process table goes to the zombie state.

A child process always first becomes a zombie before being removed from the process table.

The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

**Orphan Process:**

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

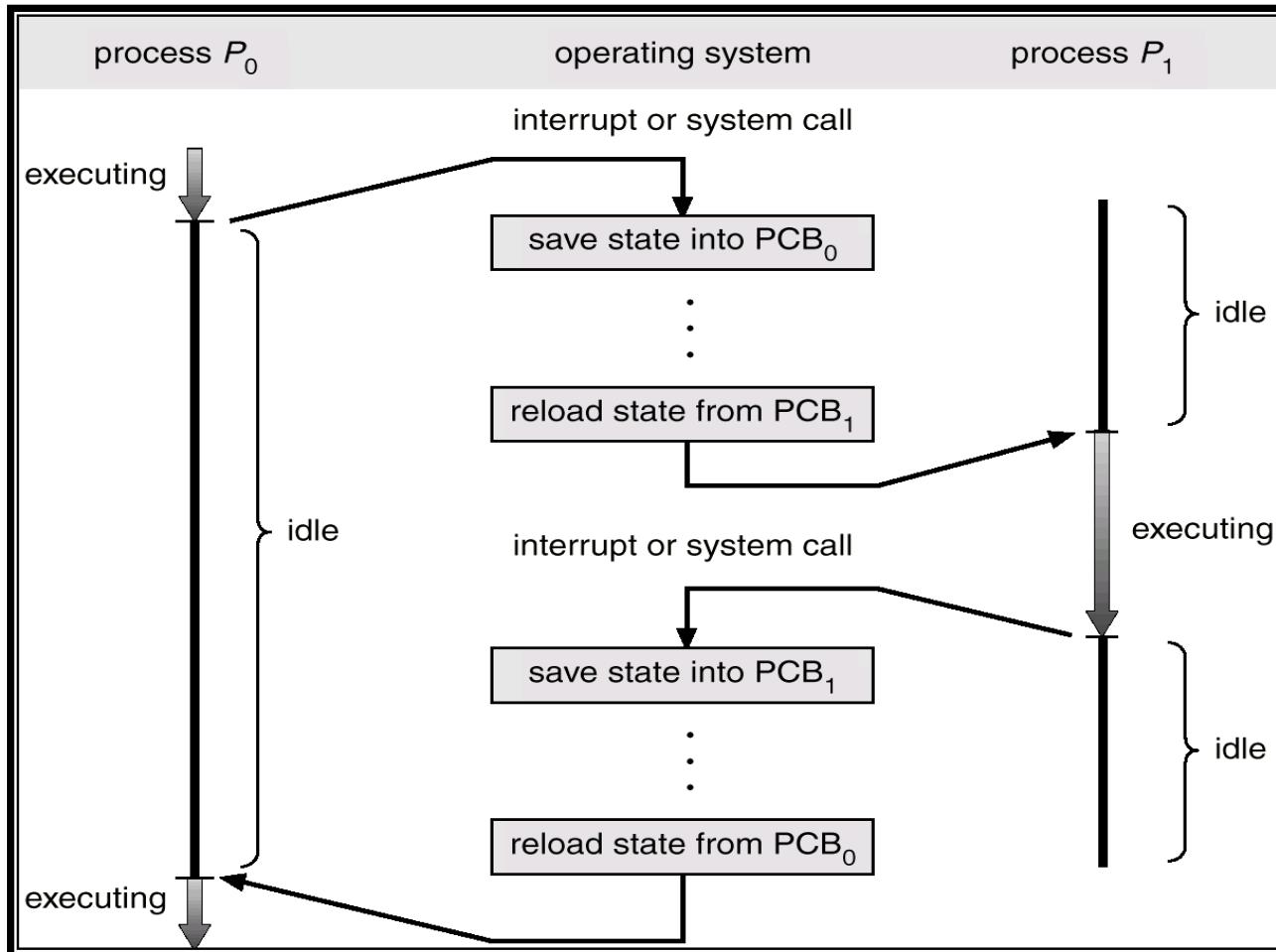
## Static and Dynamic Process:

- A process that does not terminate while the OS is functioning is called static.
- A process that may terminate is called dynamic.

# Context Switch

- When the scheduler switches the CPU from executing one process to executing another, the context switcher saves the content of all processor registers for the process being removed from the CPU in its process descriptor.
- The context of a process is represented in the PCB of a process. Context switch time is purely an overhead.
- Context switching can significantly affect performance, since modern computers have a lot of general and status registers to be saved.
- Context switch times are highly dependent on hardware support. Some hardware systems employ two or more sets of processor registers to reduce the amount of context switching time.
- When the process is switched, the information stored are Program Counter value, Scheduling Information, Base and limit register value, Currently used register, Changed State, I/O State and accounting.

# CPU Switch from Process to Process

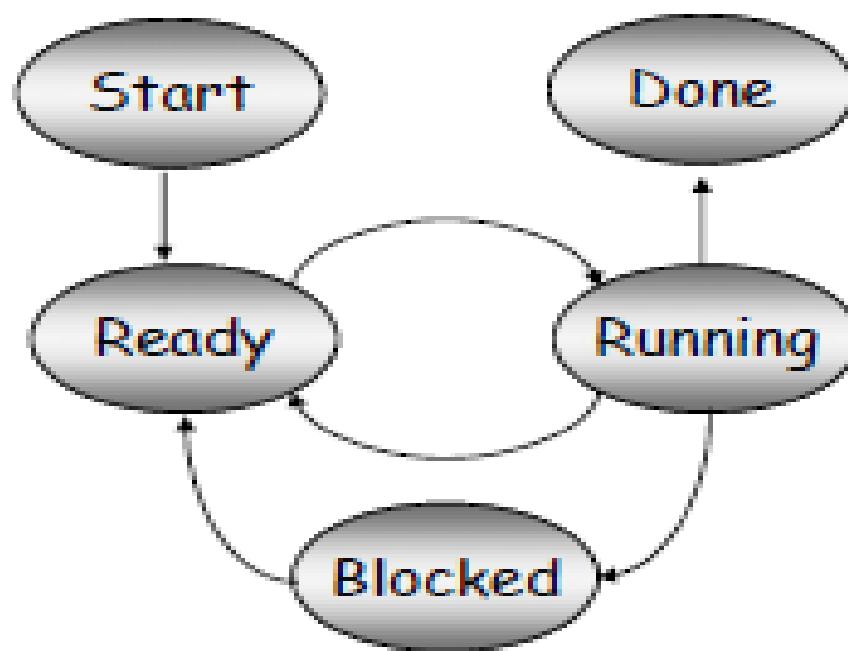


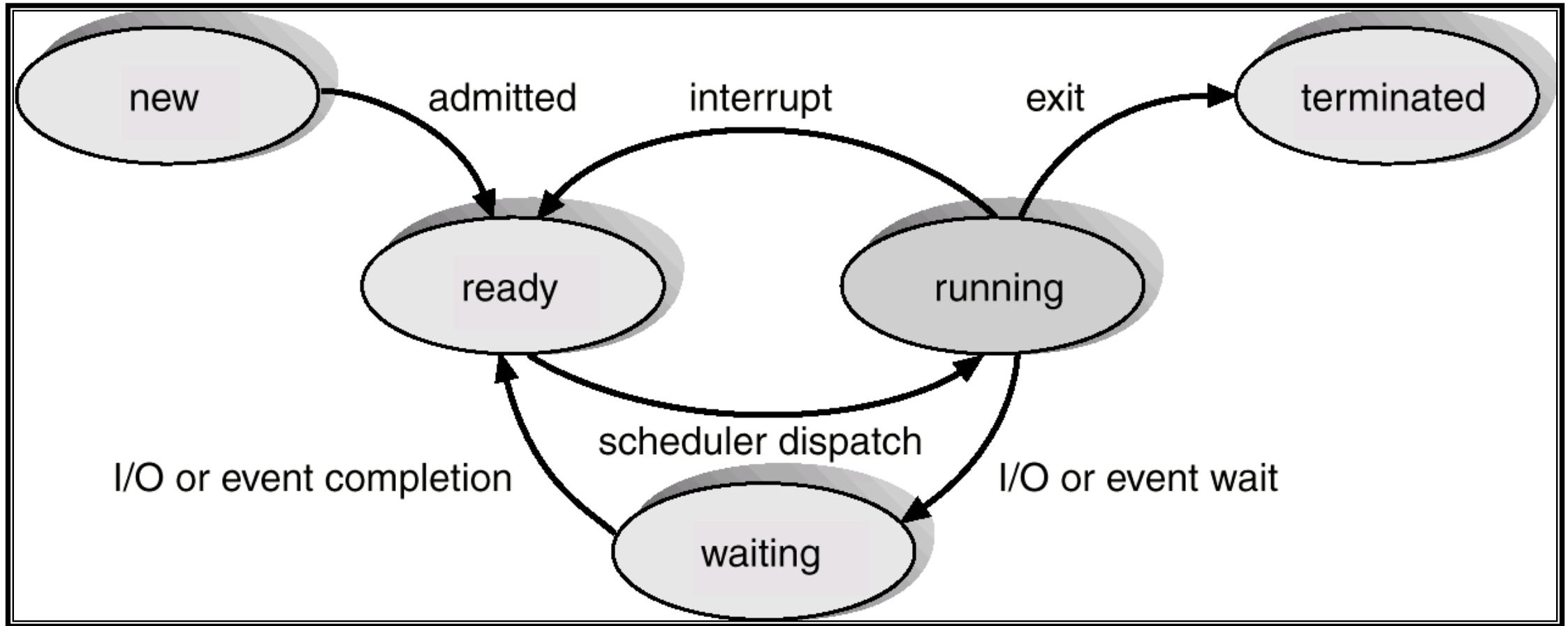
# Life Cycle of a Process

When process executes, it changes state. Process state is defined as the current activity of the process.

Once created a process can be in any of the following three basic states:

- Ready: The process is ready to be executed, but CPU is not available for the execution of this process.
- Running: The CPU is executing the instructions of the corresponding process. A running process possesses all the resources needed for its execution, including the processor.
- Blocked/ Waiting: The process is waiting for an event to occur.
  - The event can be I/O operation to be completed
  - Memory to be made available
  - A message to be received etc.





- A running process gets blocked because of a requested resource is not available or can become ready because of the CPU decides to execute another process.
- A blocked process becomes ready when the needed resource becomes available to it. A ready process starts running when the CPU becomes available to it.
- Whenever processes changes state, the OS reacts by placing the process PCB in the list that corresponds to its new state. Only one process can be running on any processor at any instant and many processes may be in ready and waiting state.

# Threads

# Thread

- Thread: single sequential flow of control within a program
- Single-threaded program can handle one task at any time.
- Multitasking allows single processor to run several concurrent threads.
- Most modern operating systems support multitasking.

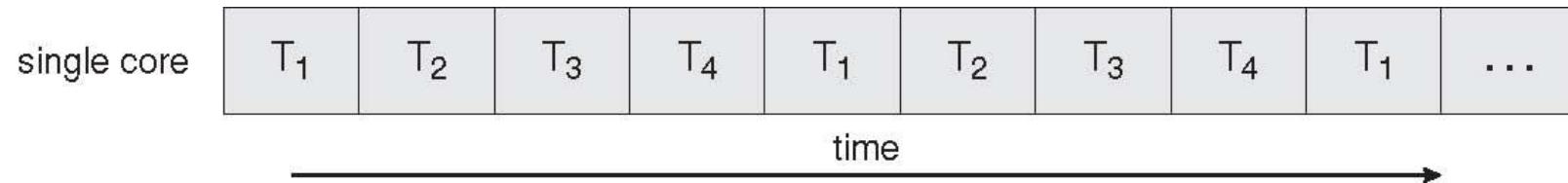
# Thread Concepts

- Traditionally a process has a single address space and a single thread of control to execute a program within that address space.
- To execute a program, a process has to initialize and maintain state information.
- The state information is comprised of page tables, swap image, file descriptors, outstanding I/O requests, saved register values etc. This information is maintained on a per program basis and thus a per process basis.
- The volume of this information makes it expensive to create and maintain processes as well as to switch between them.
- Threads or light weight processes have been proposed to handle situations where creating, maintaining and switching between processes occur frequently.

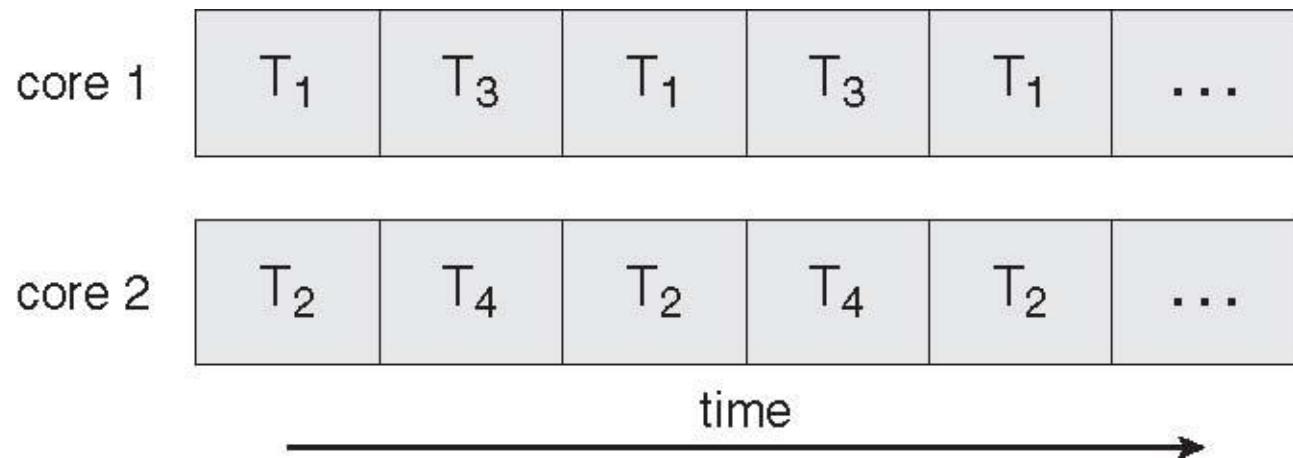
# Multicore Programming

- Multicore systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

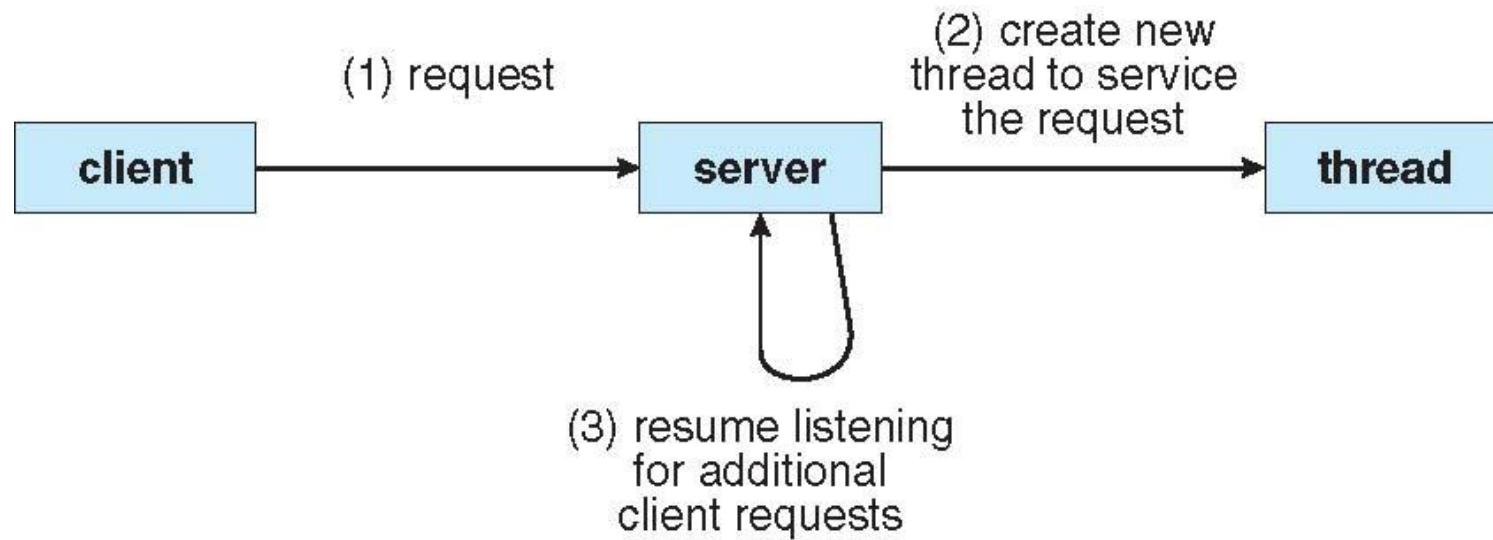
# Concurrent Execution on a Single-core System



# Parallel Execution on a Multicore System

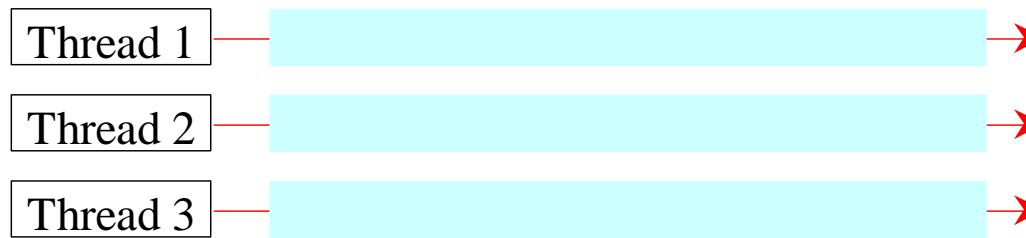


# Multithreaded Server Architecture

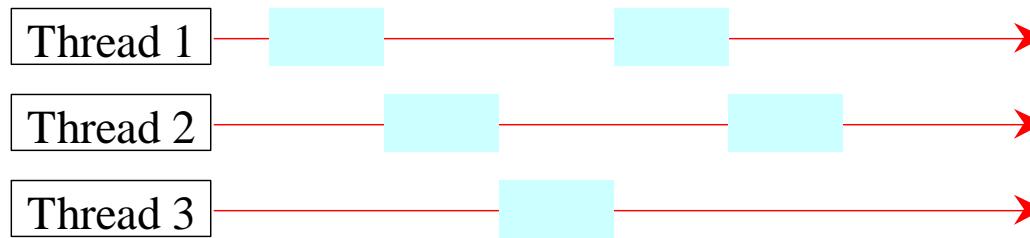


# Threads Concept

Multiple threads on multiple CPUs

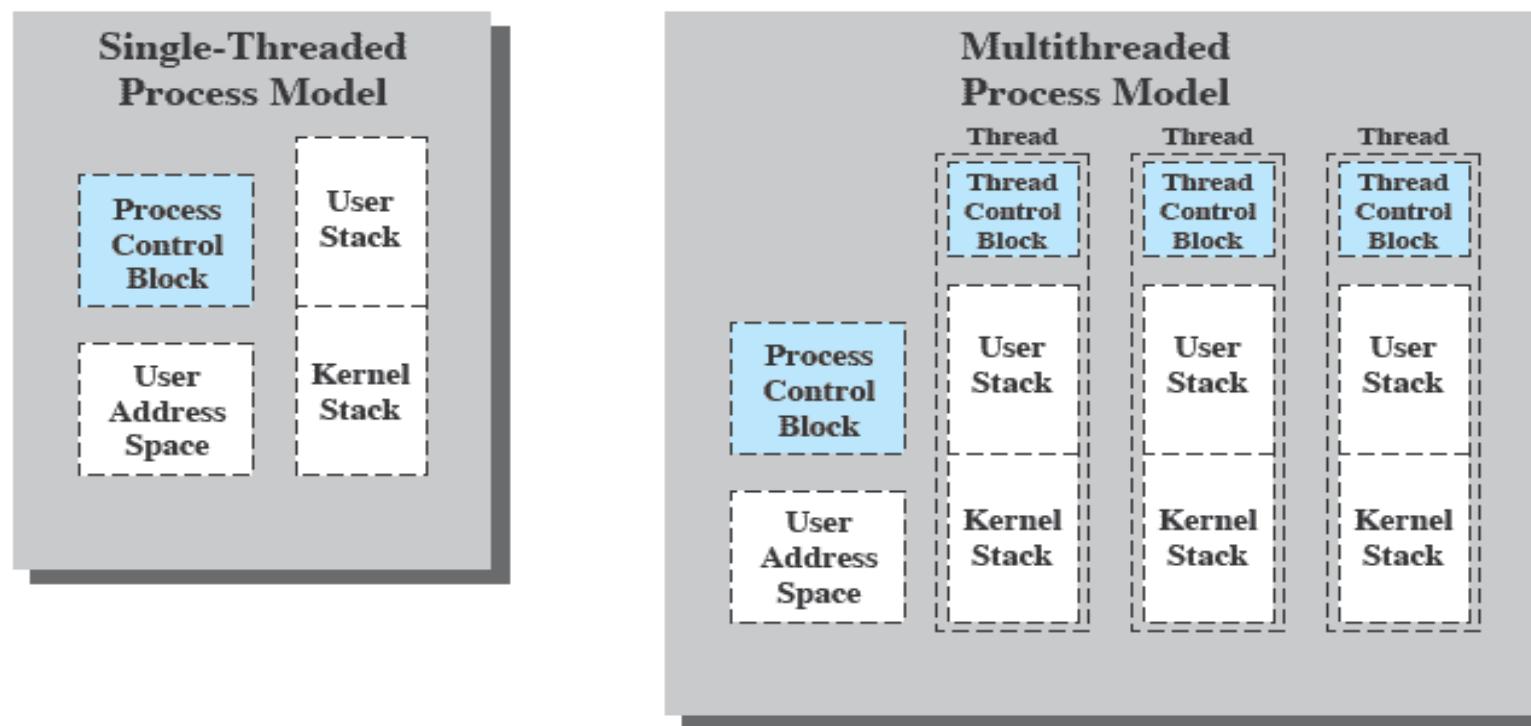


Multiple threads sharing a single CPU



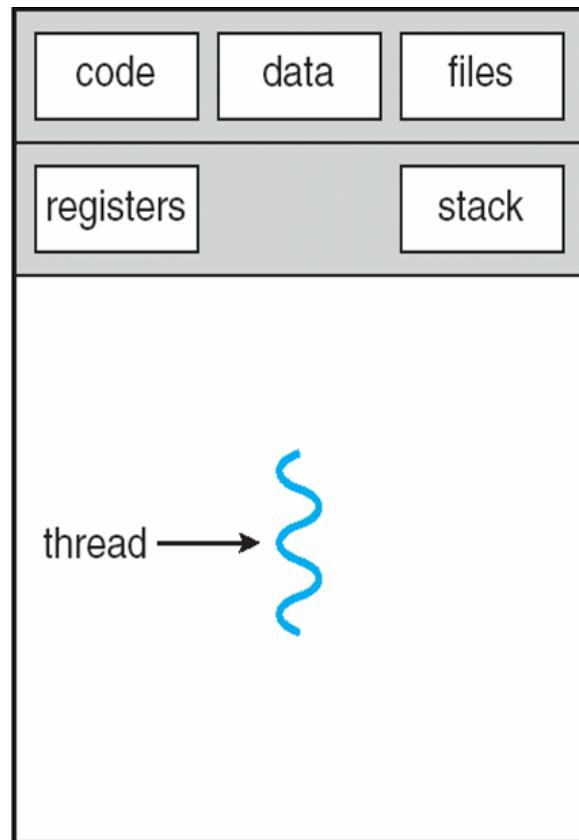
- A process is an abstraction for representing resource allocation.
- A *thread* is an abstraction for execution: a thread represents the execution of a particular sequence of instructions in a program's code, or equivalently a particular path through the program's flow of control.
- A process may have multiple threads, each sharing the resources allocated to the process.

# Threads vs Processes

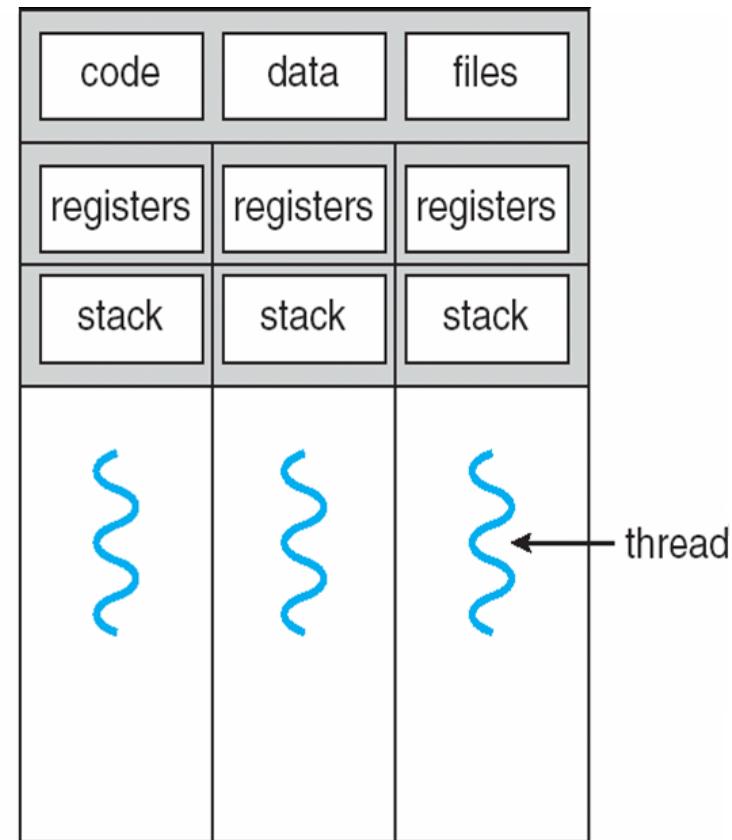


**Figure 4.2** Single Threaded and Multithreaded Process Models

# Single and Multithreaded Processes



single-threaded process



multithreaded process

- A single thread executes a portion of a program, cooperating with other threads concurrently executing within the same address space. Each thread makes use of a separate program counter and a stack of activation records and a thread control block.
- The control block contains the state information necessary for thread management, such as putting a thread into a ready list and for synchronizing with other threads.
- Threads share all resources (memory, open files, etc.) of their parent process *except* the CPU.

- The purpose of the thread abstraction is to simplify programming of logically parallel, cooperating activities; threads enable each activity to be implemented largely as a sequential program that shares resources with its peer threads.
- But, since all threads in the same process share the process's resources, communication (and hence cooperation) among threads is easier to achieve than if each thread was a separate process.
- Most of the information that is part of a process is common to all the threads executing within a single address space and hence maintenance is common to all threads.
- By sharing common information overhead incurred in creating and maintaining information and the amount of information that needs to be saved when switching between threads of the same program is reduced significantly.

Each thread requires its own context:

- program counter
- stack
- Registers

Each Thread has

- an execution state (Running, Ready, etc.)
- saved thread context when not running (TCB)
- an execution stack
- some per-thread static storage for local variables
- access to the shared memory and resources of its process (all threads of a process share this)

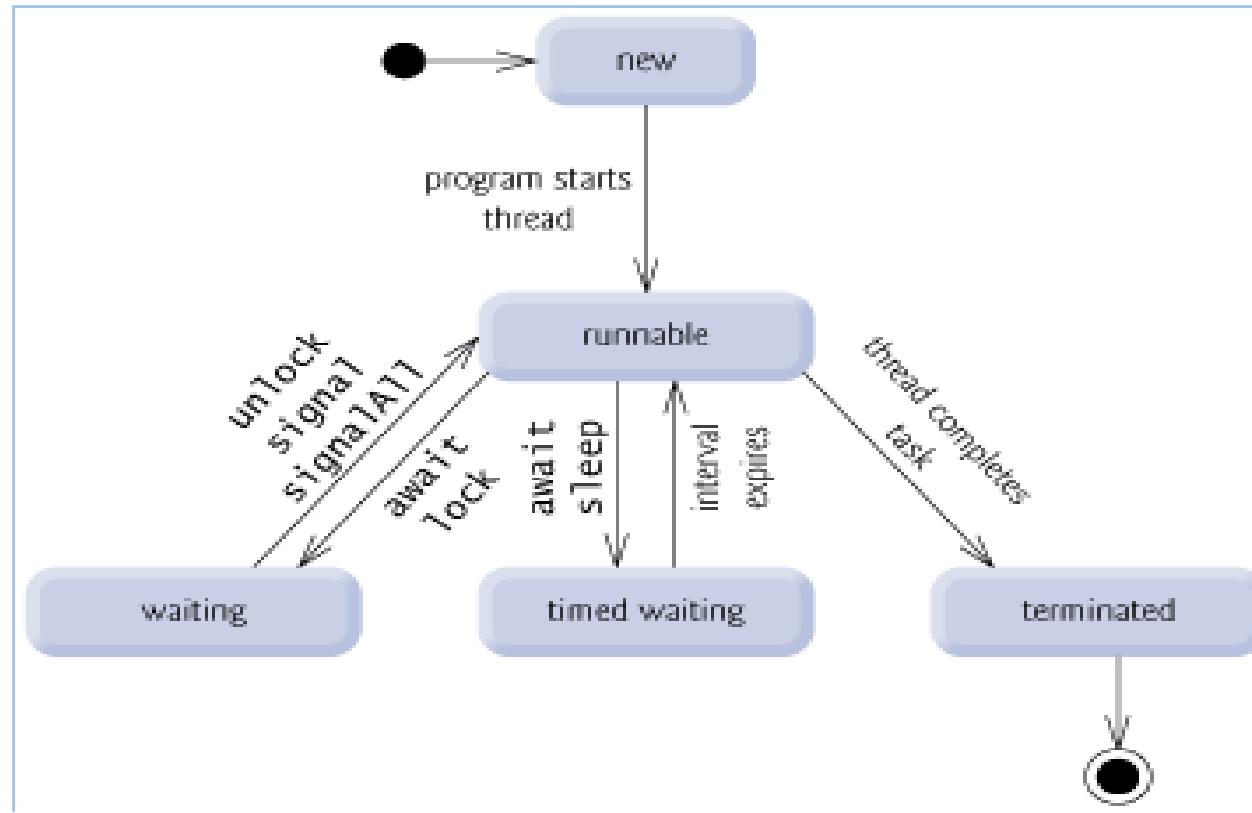
# Thread Execution States

Similar to a process a thread can be in any of the primary states:  
Running, Ready and Blocked.

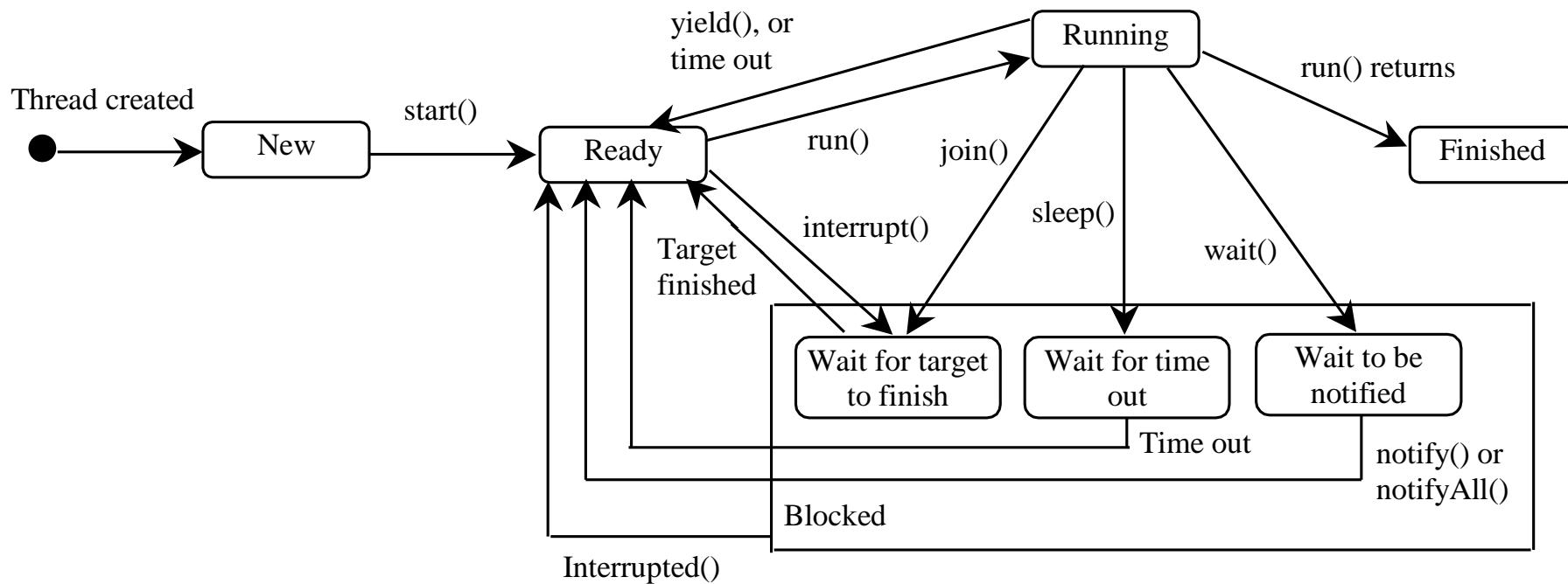
The operations needed to change state are:

- Spawn: new thread provided register context and stack pointer.
- Block: event wait, save user registers, PC and stack pointer
- Unblock: moved to ready state
- Finish: deallocate register context and stacks.

# Thread States



# Thread States



# Thread Not Runnable

A thread becomes Not Runnable when one of these events occurs:

- Its sleep method is invoked.
- The thread calls the wait method to wait for a specific condition to be satisfied.
- The thread is blocking on I/O.

# Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
  - suspending a process involves suspending all threads of the process
  - termination of a process terminates all threads within the process

# Thread Scheduling

- An operating system's thread scheduler determines which thread runs next.
- Most operating systems use *timeslicing* for threads of equal priority.
- *Preemptive scheduling*: when a thread of higher priority enters the running state, it preempts the current thread.
- *Starvation*: Higher-priority threads can postpone (possibly forever) the execution of lower-priority threads.

# Thread Synchronization

- It is necessary to synchronize the activities of the various threads
  - all threads of a process share the same address space and other resources
  - any alteration of a resource by one thread affects the other threads in the same process

# Common Thread Models

- User level threads

These threads implemented as user libraries. Thread library provides programmer with API for creating and managing threads. Benefits of this are no kernel modifications, flexible and low cost. The drawbacks are thread may block entire process and no parallelism.

- Kernel level threads

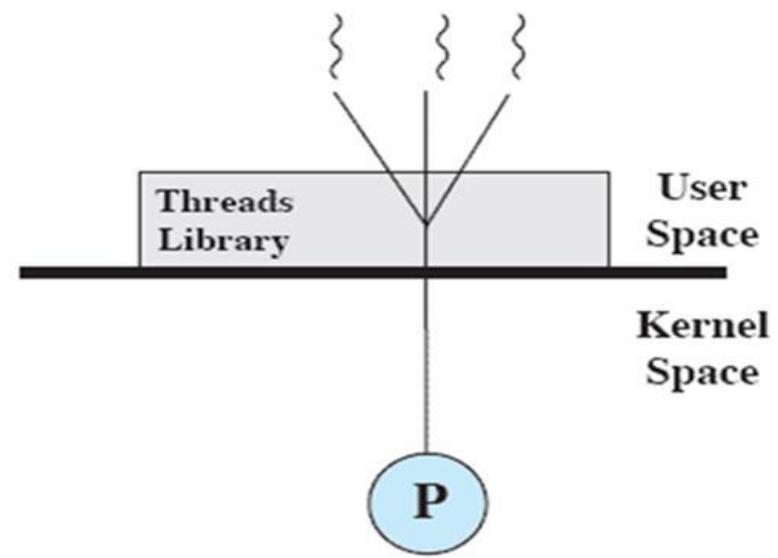
Kernel directly supports multiple threads of control in a process. Benefits are scheduling/synchronization coordination, less overhead than process, suitable for parallel application. The drawbacks are more expensive than user-level threads and more overhead.

- Light-Weight Processes (LWP)

It is a kernel supported user thread. LWP bound to kernel thread but a kernel thread may not be bound to an LWP. It is scheduled by kernel and user threads scheduled by library onto LWPs, so multiple LWPs per process.

# User-Level Threads (ULTs)

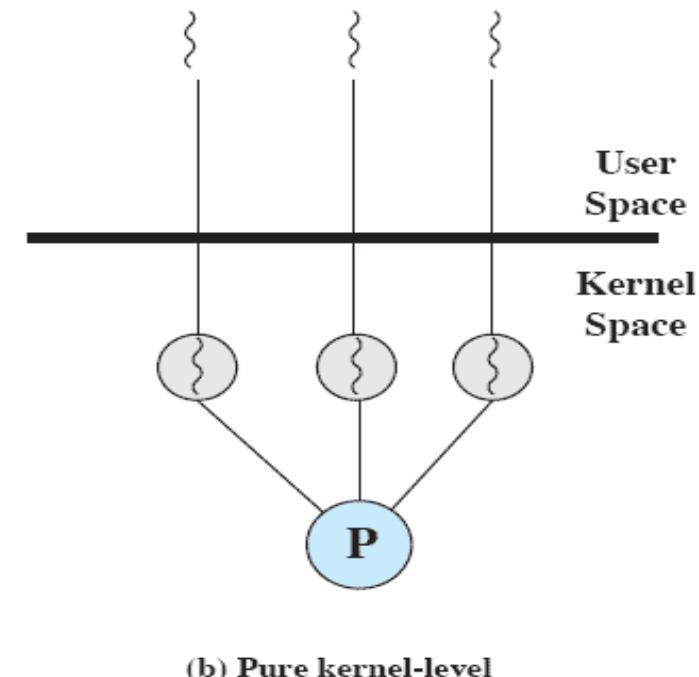
- Thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level

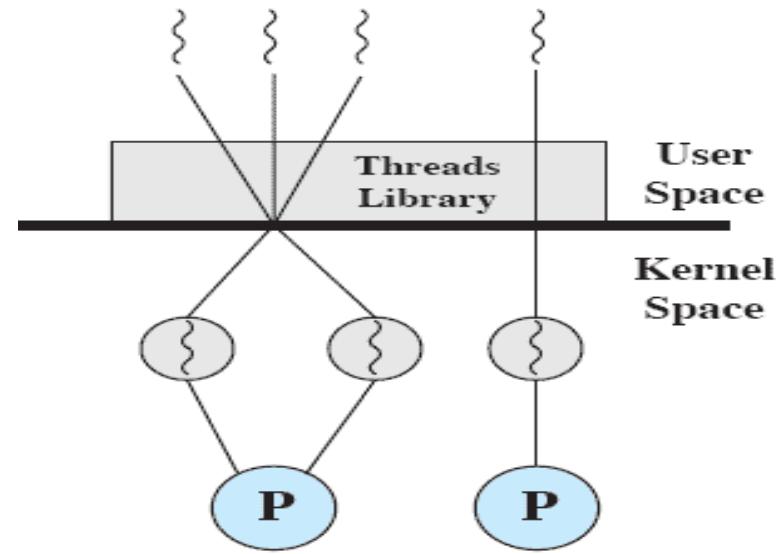
# Kernel-Level Threads (KLTs)

- Thread management is done by the kernel (could call them KMT)
- No thread management is done by the application; Windows is an example of this approach



# Combined Approaches

- Thread creation is done in the user space
- Bulk of scheduling and synchronization of threads is by the application
- Solaris is an example



(c) Combined

# Multithreading

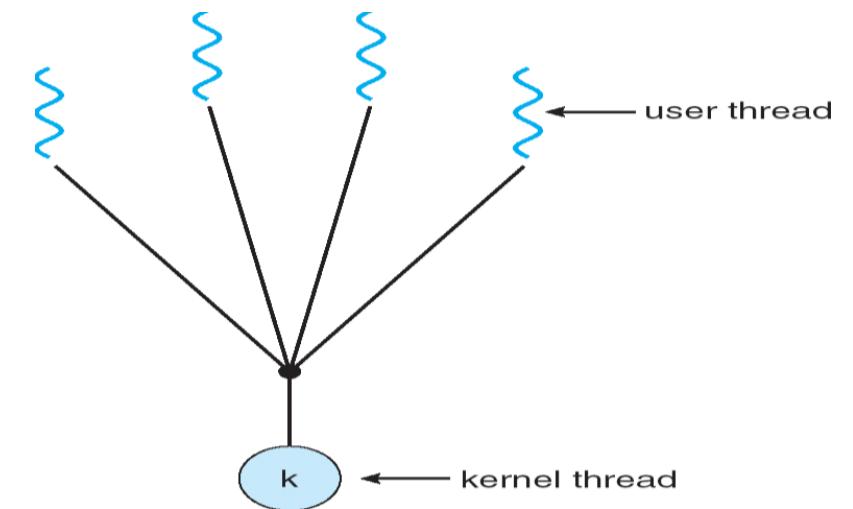
Kernels are generally multi-threaded.

Multi-threading models include

- Many-to-One: Many user-level threads mapped to single kernel thread
- One-to-One: Each user-level thread maps to kernel thread
- Many-to-Many: Many user-level threads mapped to many kernel threads.

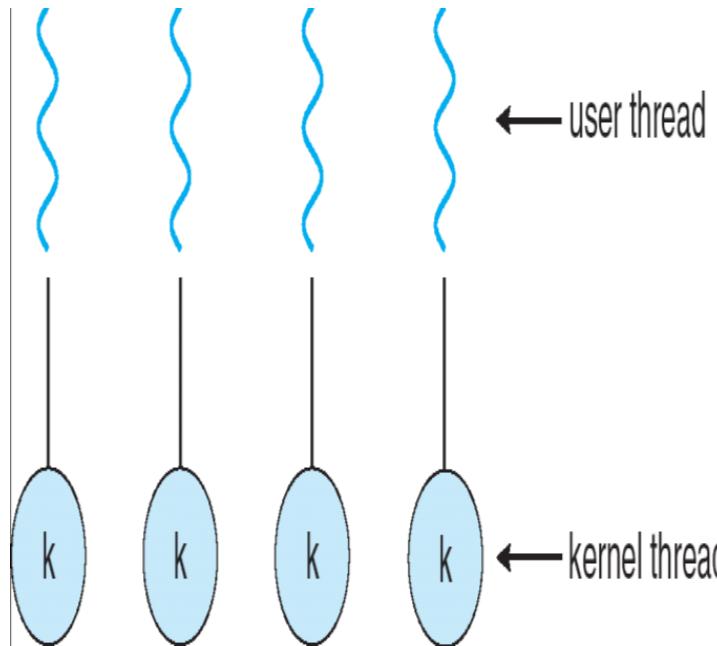
# Many-to-One

- Thread management is done by the thread library in user space
- The entire process will block if a thread makes a blocking system call
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Mainly used in language systems, portable libraries
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



- Advantages:
  - totally portable
  - easy to do with few systems dependencies
- Disadvantages:
  - cannot take advantage of parallelism
  - may have to block for synchronous I/O

# One-to-One

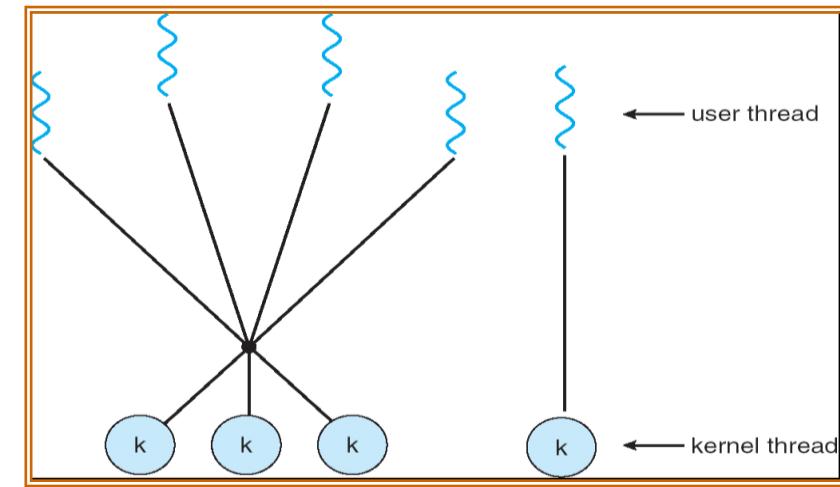
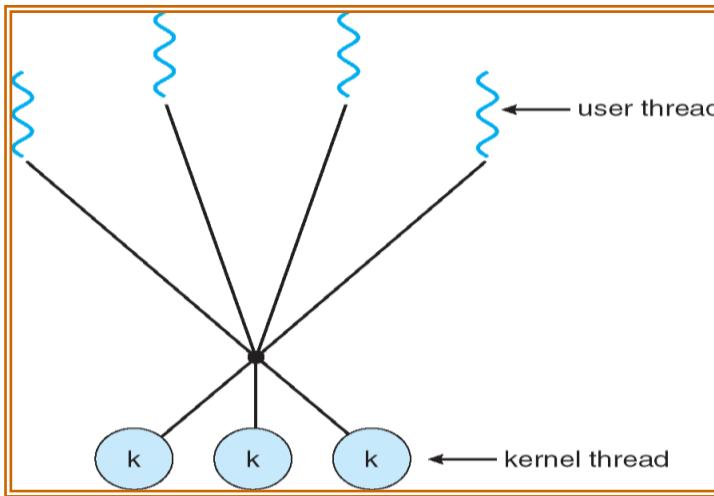


- Each **user-level thread** maps to **kernel thread**
  - Creating a **user-level thread** **creates a kernel thread**
  - *This is a drawback to this model; creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may **burden** the performance of a system.*
- More concurrency than many-to-one
  - Allows another thread to run when a thread makes a blocking system call
  - It also allows multiple threads to run in parallel on multiprocessors.
- Number of threads per process sometimes restricted due to overhead
- Used in LinuxThreads and other systems where LWP creation is not too expensive

- Advantages:
  - can exploit parallelism, blocking system calls
- Disadvantages:
  - thread creation involves LWP creation
  - each thread takes up kernel resources
  - limiting the number of total threads

# Many-to-many

- In this model, the library has two kinds of threads: *bound* and *unbound*
  - bound threads are mapped each to a single lightweight process
  - unbound threads *may* be mapped to the same LWP



- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
  - The number of kernel threads may be specific to either a particular application or a particular machine
  - *An application may be allocated more kernel threads on a system with eight processing cores than a system with four cores*
- Used in the Solaris implementation of Pthreads (and several other Unix implementations)
- Windows with the *ThreadFiber* package
- Otherwise not very common

- Issues in multithreading include
  - Thread Creation
  - Thread Cancellation
  - Signal Handling (synchronous / asynchronous),
  - Handling thread-specific data and scheduler activations.

## Thread pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

## Thread cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - Asynchronous cancellation terminates the target thread immediately
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled

## Signal handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  - Deliver the signal to the thread to which the signal applies
  - 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

## Thread specific data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

## Scheduler activations

- Many : Many models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

# Relationship Between Threads and Processes

Threads:Processes	Description	Example Systems
<b>1:1</b>	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
<b>M:1</b>	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
<b>1:M</b>	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
<b>M:N</b>	Combines attributes of M:1 and 1:M cases.	TRIX

## User Threads

- Thread management done by user-level threads library
- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*

## Supported by the Kernel

- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux

# Various Implementations

## PThreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

## Windows Threads

- Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads

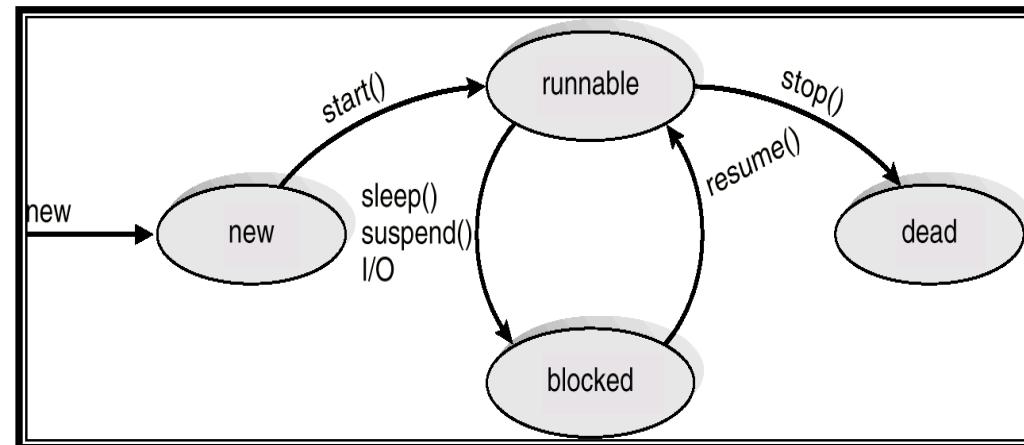
# Various Implementations

## Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

## Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java threads are managed by the JVM.



# Process Scheduling

- Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.
- Multiprogramming system's objective is to allow processes run all the time so that CPU utilization is maximized. With CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform.

- The scheduling mechanism is the part of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of particular strategy.
- Aim is to assign processes to be executed by the processor in a way that meets system objectives, such as response time, throughput, and processor efficiency
- For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

- Process may be in one of two states: Running and Not Running.
- When a new process is created by OS, that process enters into the system in the running state. Processes that are not running are kept in queue, waiting their turn to execute. Queue is implemented by using linked list.

Use of dispatcher is as follows:

- When a process is interrupted, that process is transferred to the waiting queue.
- If the process has completed or aborted, the process is discarded.
- In either case, the dispatcher then selects a process from the queue to execute.

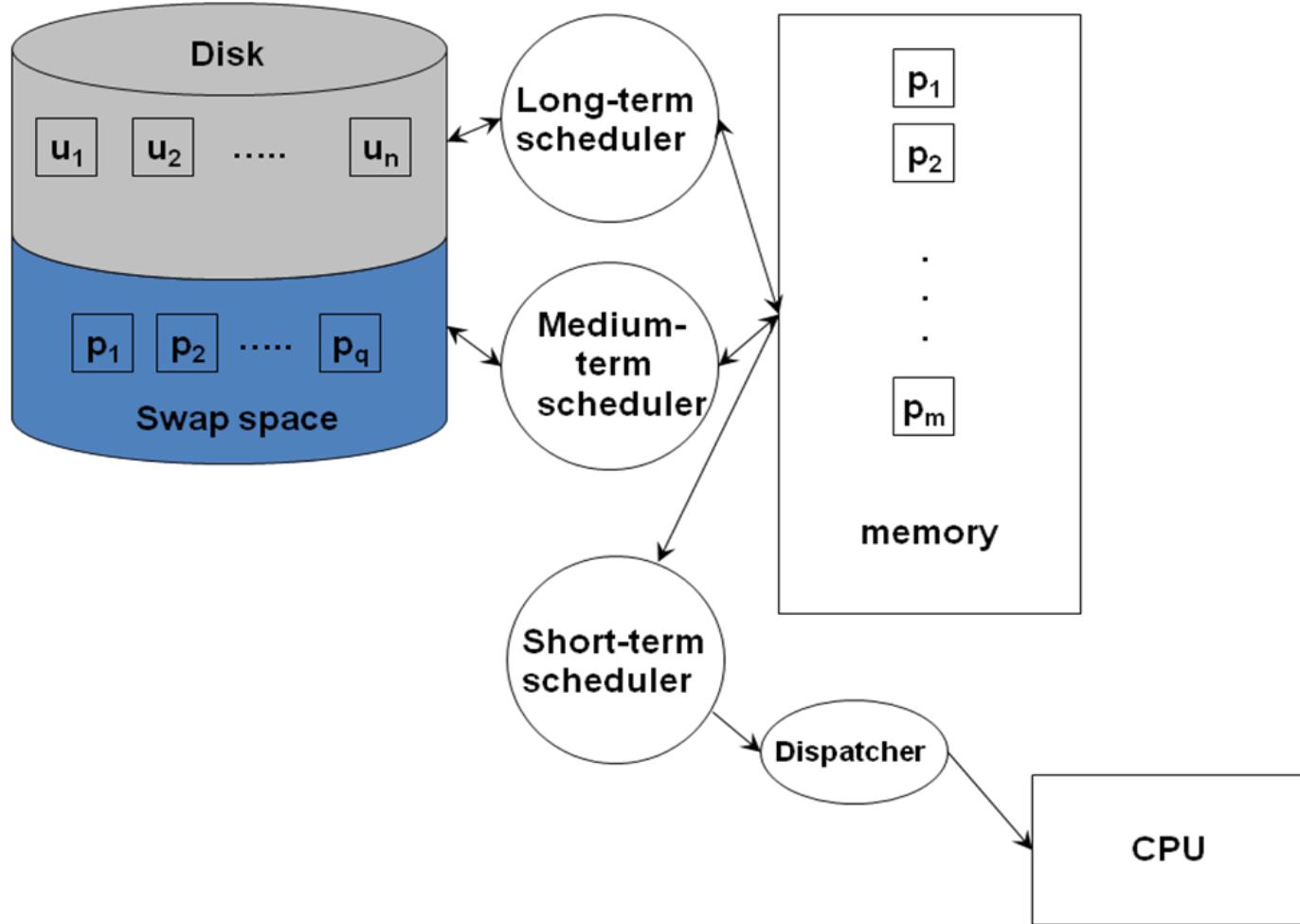
# When is Scheduling needed?

- When a new process is created
- When a process exits
- When a process is blocked and waiting (on an IO device, a semaphore)
- When an I/O interrupt occurs

# Schedulers

- Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run.
- Schedulers are of three types
  - Long-Term Scheduler
  - Short-Term Scheduler
  - Medium-Term Scheduler

# Scheduling Environments



# Types of Scheduling

**Long-term scheduling** The decision to add to the pool of processes to be executed

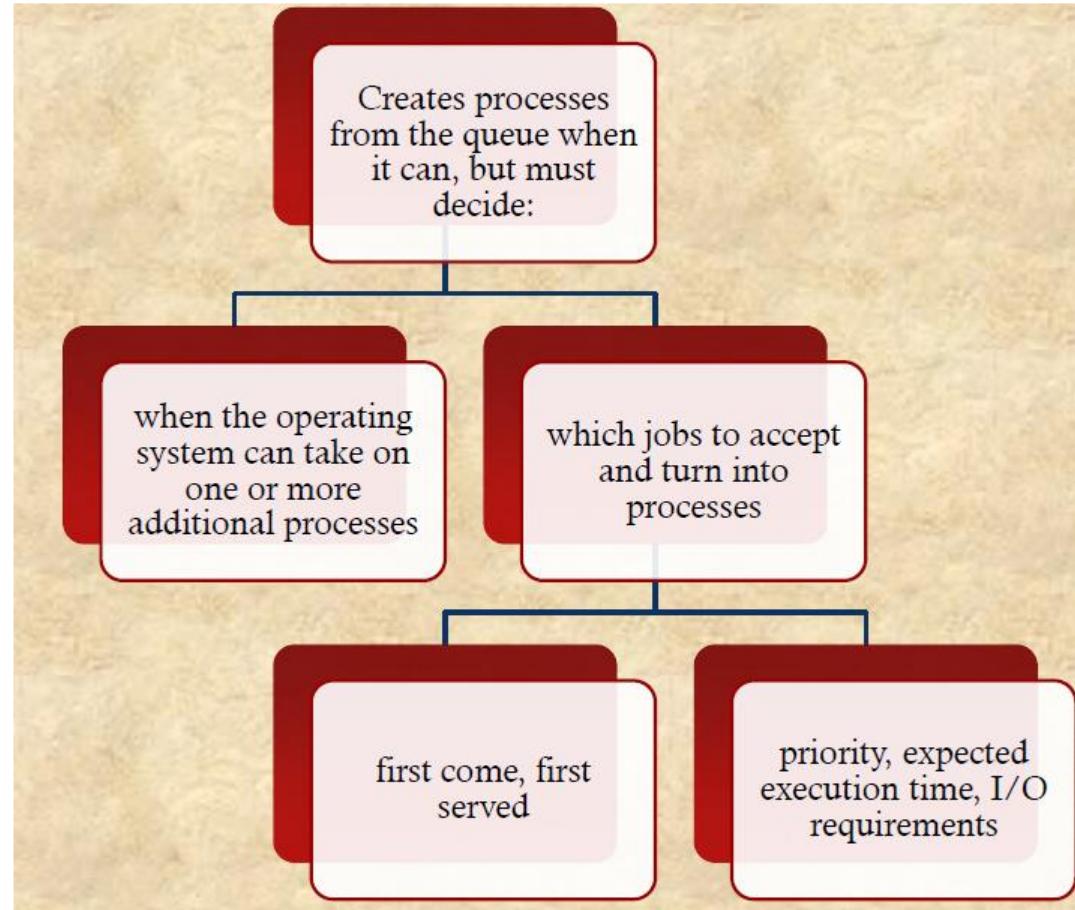
**Medium-term scheduling** The decision to add to the number of processes that are partially or fully in main memory

**Short-term scheduling** The decision as to which available process will be executed by the processor

**I/O scheduling** The decision as to which process's pending I/O request shall be handled by an available I/O device

# Long-Term Scheduler

- Determines which programs are admitted to the system for processing
- Controls the degree of multiprogramming
- the more processes that are created, the smaller the percentage of time that each process can be executed
- may limit to provide satisfactory service to the current set of processes



# Medium-Term Scheduling

- Part of the swapping function
- Swapping-in decisions are based on the need to manage the degree of multiprogramming
- considers the memory requirements of the swapped-out processes

# Short-Term Scheduling

- Known as the dispatcher
- Executes most frequently
- Makes the fine-grained decision of which process to execute next
- Invoked when an event occurs that may lead to the blocking of the current process or that may provide an opportunity to preempt a currently running process in favor of another
- Examples:
  - Clock interrupts
  - I/O interrupts
  - Operating system calls
  - Signals (e.g., semaphores)

# Short Term Scheduling Criteria

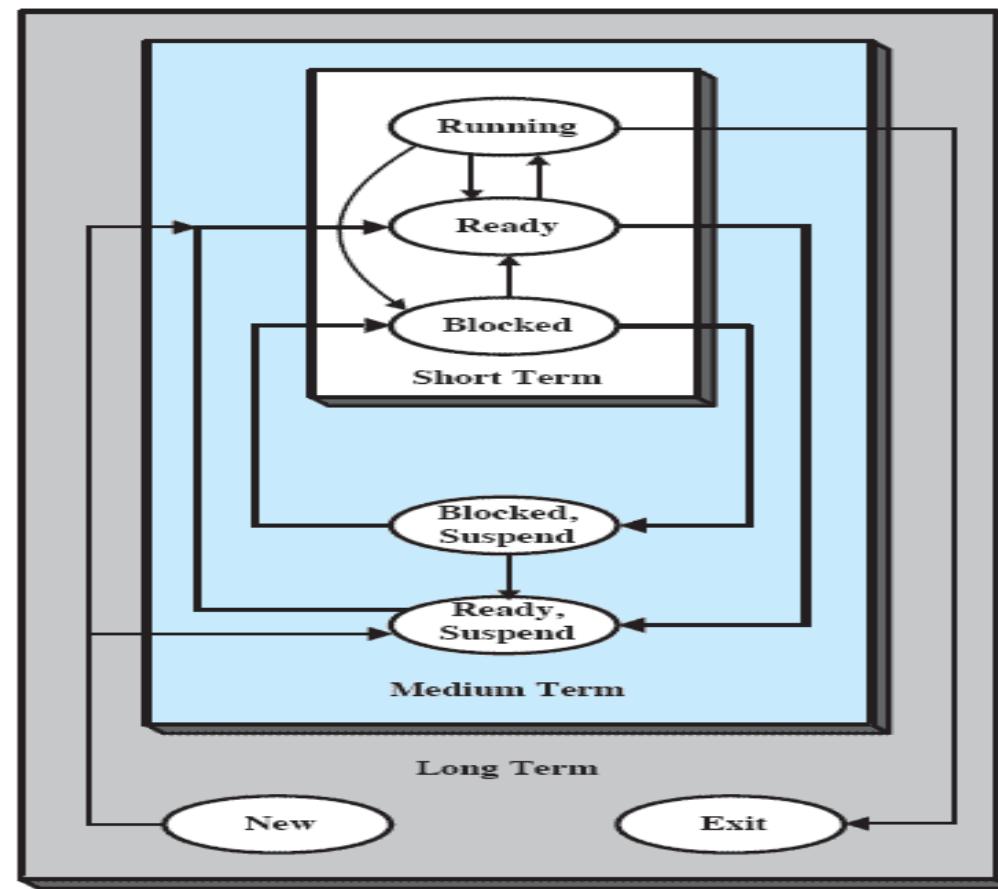
- Main objective is to allocate processor time to optimize certain aspects of system behavior
- A set of criteria is needed to evaluate the scheduling policy
- User-oriented criteria
  - relate to the behavior of the system as perceived by the individual user or process (such as response time in an interactive system)
  - important on virtually all systems
- System-oriented criteria
  - focus in on effective and efficient utilization of the processor (rate at which processes are completed)
  - generally of minor importance on single-user systems

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

# Scheduling and Process State Transitions



(b) With Two Suspend States

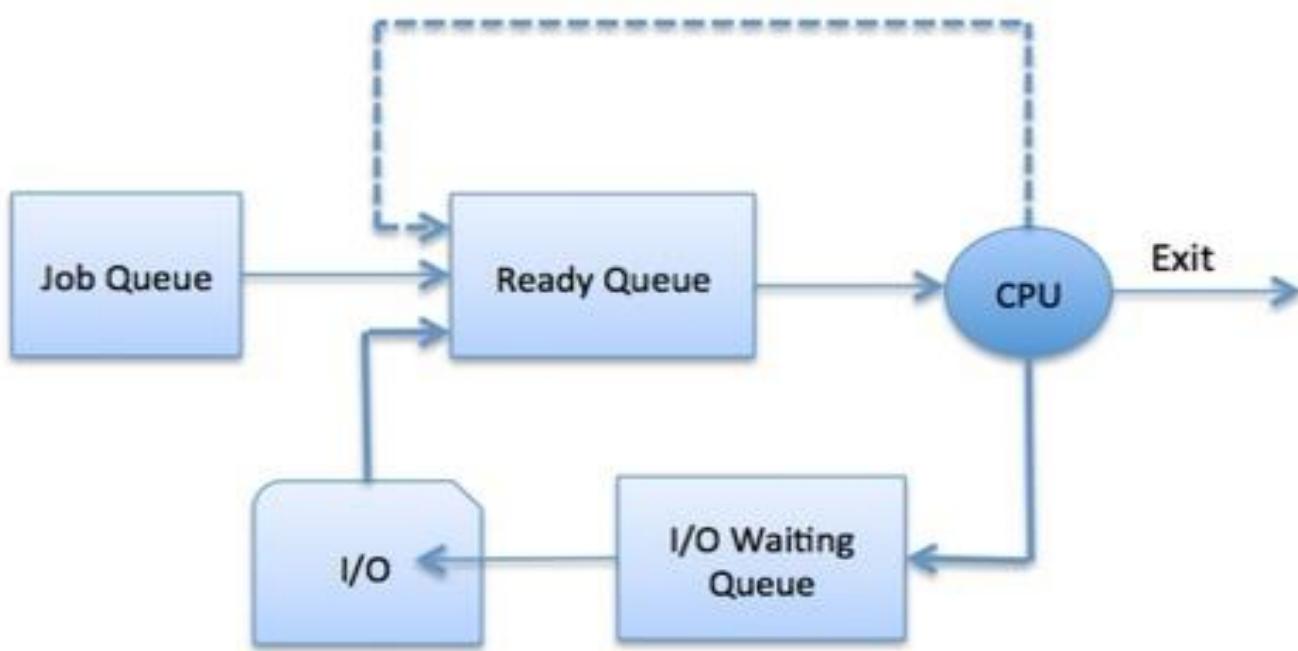


# Process Scheduling Queues

- The OS maintains all PCBs in Process Scheduling Queues.
- The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue.
- When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues

- **Job queue** – This queue keeps all the processes in the system.
- **Ready queue** – This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** – The processes which are blocked due to unavailability of an I/O device constitute this queue.



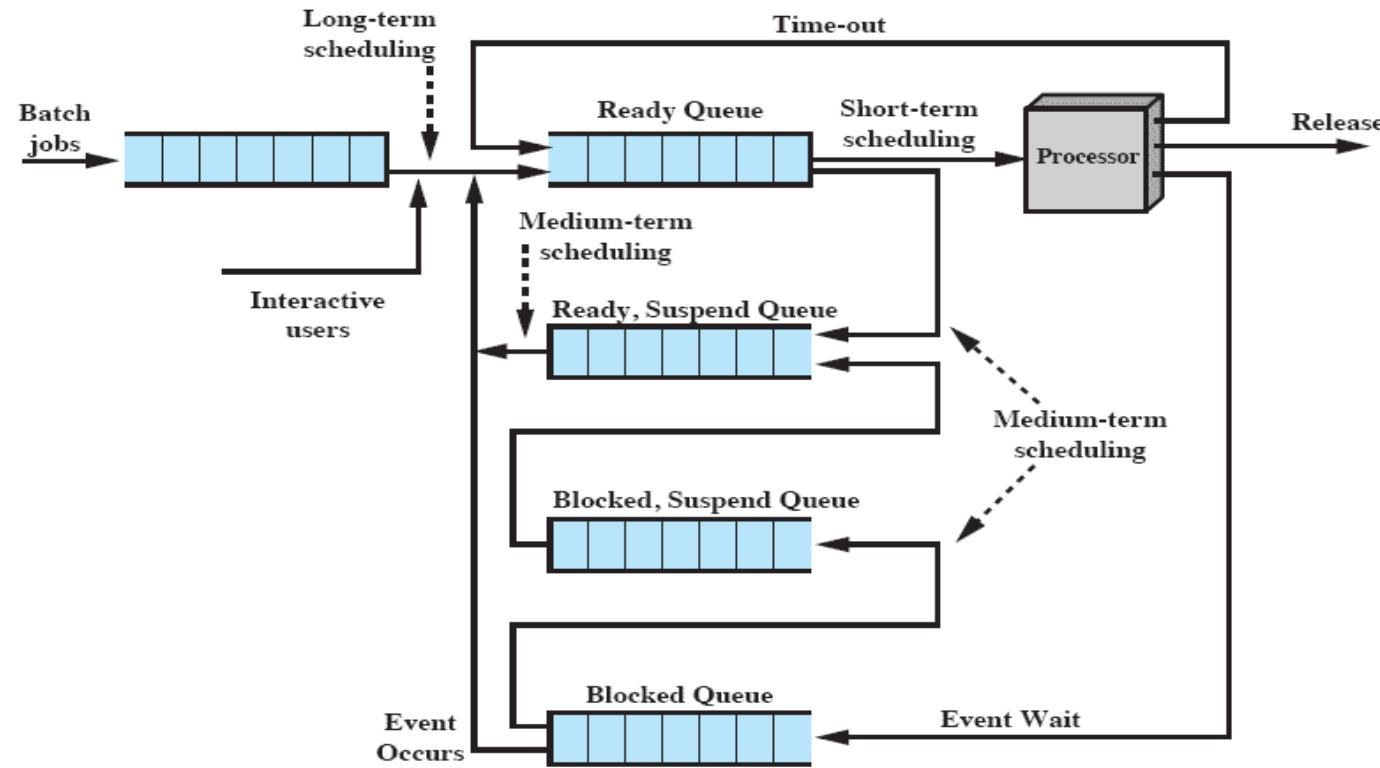


Figure 9.3 Queuing Diagram for Scheduling

# Scheduling policies

# Scheduling Criteria

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

# Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue and blocked queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

# Optimization Criteria

- Max throughput
  - Jobs per second
  - Throughput related to response time, but not identical
    - Minimizing response time will lead to more context switching than if you maximized only throughput
  - Minimize overhead (context switch time) as well as efficient use of resources (CPU, disk, memory, etc.)
- Min response time
  - Elapsed time to do an operation (job)
  - Response time is what the user sees
    - Time to echo keystroke in editor
    - Time to compile a program
    - Real-time Tasks: Must meet deadlines imposed by World

- Min turnaround time
- Min waiting time
- Max CPU utilization
- Fairness
  - Share CPU among users in some equitable way
  - Not just minimizing average response time

# Selection Function

- Determines which process is selected for execution.
- If it is based on execution characteristics then important quantities are:
  - $w$  = time spent in system so far, waiting
  - $e$  = time spent in execution so far
  - $s$  = total service time required by the process, including  $e$

# Decision Mode

- Specifies the instants in time at which the selection function is exercised.
- Two categories:
  - Non-preemptive
  - Preemptive

# Non-preemptive vs Preemptive

- Non-preemptive
  - Once a process is in the running state, it will continue until it terminates or blocks itself for I/O or OS service.
  - i.e. Run until completion or until they yield control of a processor
  - Unimportant processes can block important ones indefinitely
- Preemptive
  - Currently running process may be interrupted and moved to ready state by the OS.
  - Preemption may occur when new process arrives, on an interrupt, or periodically.
  - Can be removed from their current processor
  - Can lead to improved response times
  - Important for interactive environments
  - Preempted processes remain in memory

# Priorities

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

# Scheduling Objectives

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

# Scheduling Objectives

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

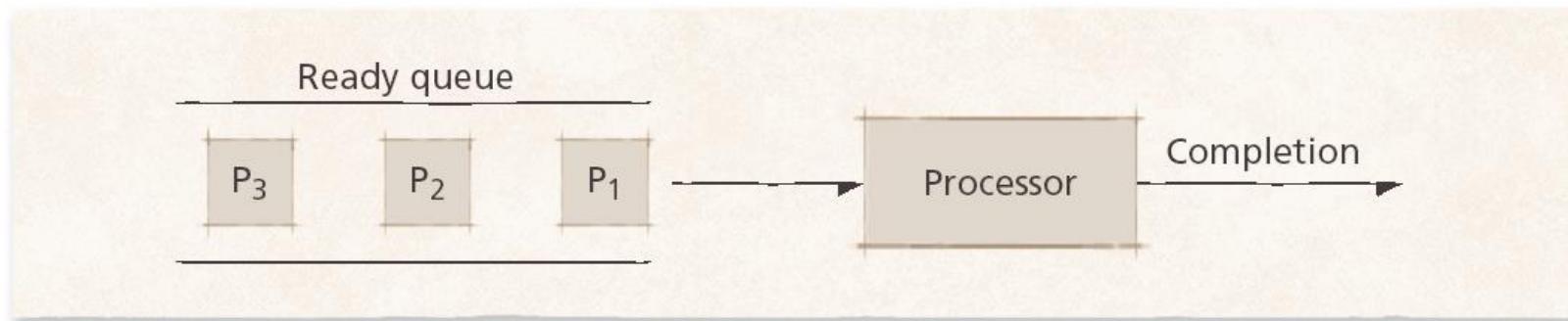
# Scheduling Algorithms

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

# First-In-First-Out (FIFO)/ First Come First Serve (FCFS) Scheduling

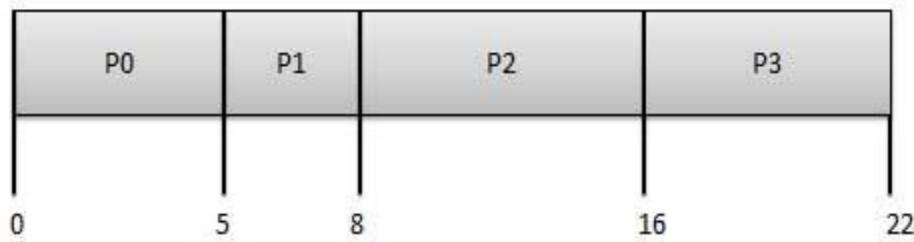
- Simplest scheme: Jobs are executed on first come, first serve basis
- Processes dispatched according to arrival time
- It is a non-preemptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue
- Poor in performance as average wait time is high
- Rarely used as primary scheduling algorithm
- Convoy effect

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



# First Come First Serve (FCFS)

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



**Turnaround time** of each process is as follows –

Process	Turnaround Time
P0	$5 - 0 = 5$
P1	$8 - 1 = 7$
P2	$16 - 2 = 14$
P3	$22 - 3 = 19$

Average Turnaround Time:  $(5+7+14+19) / 4 = 11.25$

**Wait time** of each process is as follows –

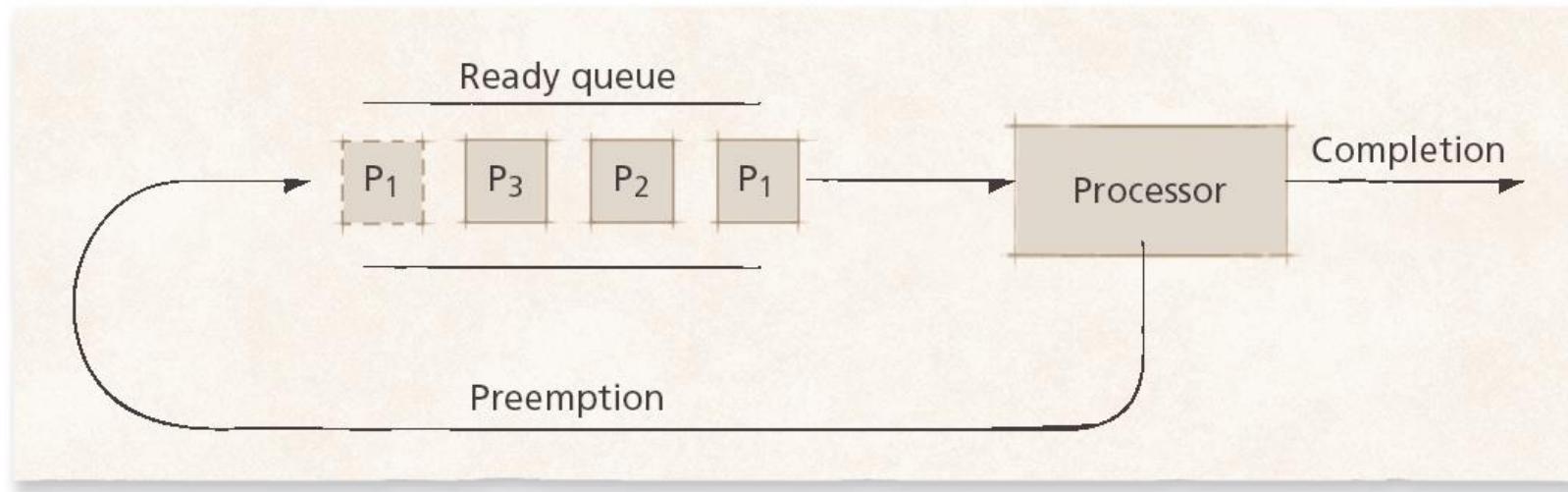
Process	Wait Time
P0	$0 - 0 = 0$
P1	$5 - 1 = 4$
P2	$8 - 2 = 6$
P3	$16 - 3 = 13$

$$\text{Average Wait Time: } (0+4+6+13) / 4 = 5.75$$

# Round-Robin (RR) Scheduling

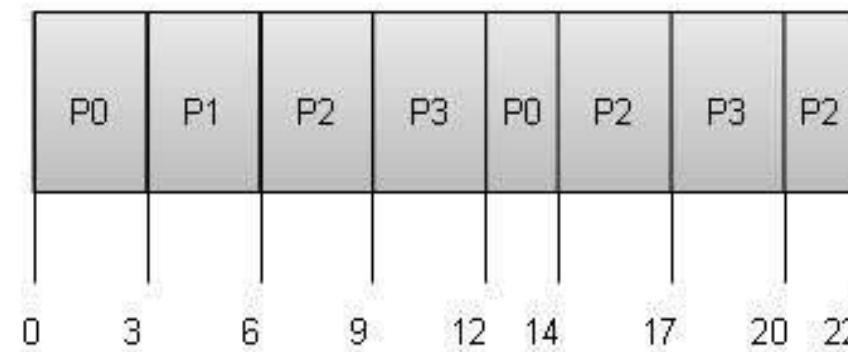
- Round-robin scheduling
  - Based on FIFO
  - Processes run only for a limited amount of time called a time slice or quantum
  - Preemptible
  - Requires the system to maintain several processes in memory to minimize overhead
  - Context switching is used to save states of preempted processes.
  - Often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

# Round-Robin (RR) Scheduling



Assumption at time = 3, P3 is first put in the ready queue then P0.

Quantum = 3



Turnaround time of each process is as follows :

Process	Turnaround Time
P0	$14 - 0 = 14$
P1	$6 - 1 = 5$
P2	$22 - 2 = 20$
P3	$20 - 3 = 17$

$$\text{Average Turnaround Time: } (14 + 5 + 20 + 17) / 4 = 14$$

**Wait time** of each process is as follows

Proces s	Wait Time
P0	$(0 - 0) + (12 - 3) = 9$
P1	$(3 - 1) = 2$
P2	$(6 - 2) + (14 - 9) + (20 - 17) = 12$
P3	$(9 - 3) + (17 - 12) = 11$

$$\text{Average Wait Time: } (9+2+12+11) / 4 = 8.5$$

# Round-Robin (RR) Scheduling

## Variants of RR:

- State dependent RR: same as RR but Q is varied dynamically depending on the state of the system.
- External priorities/ Weighted RR: RR but a user can pay more and get bigger Q.
- Selfish RR:
  - A new process starts at priority 0; its priority increases at rate  $a \geq 0$ . It becomes an accepted process when its priority reaches that of an accepted process (or until there are no accepted processes). At any time all accepted processes have same priority.
  - Increases priority as process ages
  - Two queues
    - Active
    - Holding
  - Favors older processes to avoids unreasonable delays

# Round-Robin (RR) Scheduling

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

# Shortest-Process-First/Next (SPF/N) / Shortest Job First (SJF) Scheduling

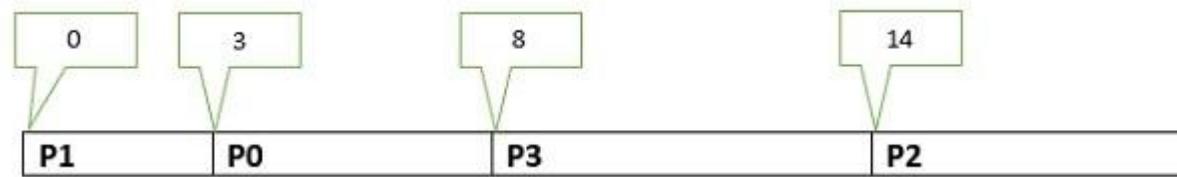
- Scheduler selects process with smallest time to finish
  - Lower average wait time than FIFO
    - Reduces the number of waiting processes
  - Potentially large variance in wait times
  - Non-preemptive
    - Results in slow response times to arriving interactive requests
  - Best approach to minimize waiting time.
  - Relies on estimates of time-to-completion
    - Can be inaccurate or falsified
  - Easy to implement in Batch systems where required CPU time is known in advance.
  - Impossible to implement in interactive systems where required CPU time is not known. The processor should know in advance how much time process will take.
  - Unsuitable for use in modern interactive systems

# Shortest Job First

Process	Arrival Time	Execute Time
P0	0	5
P1	0	3
P2	0	8
P3	0	6

# Gantt Chart

When arrival time of all processes is zero



Turnaround Time of each process is as follows:

Process	Turnaround Time
P0	8
P1	3
P2	22
P3	14

$$\text{Average Turnaround Time: } (8+3+22+14) / 4 = 11.75$$

Wait Time of each process is as follows:

Process	Wait Time
P0	3
P1	0
P2	14
P3	8

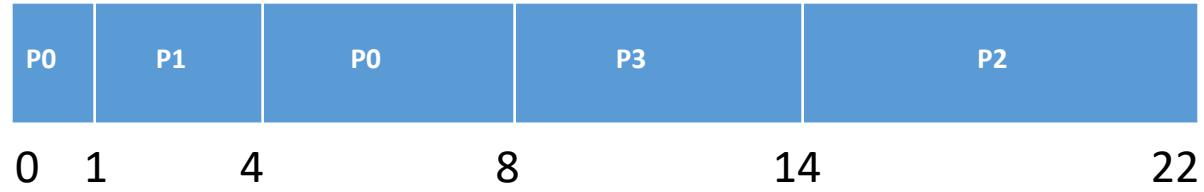
$$\text{Average Wait Time: } (3+14+8) / 4 = 6.25$$

# Shortest-Remaining-Time (SRT) Scheduling

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

# Shortest Remaining Time

Process	Arrival Time	Execute Time
P0	0	5
P1	1	3
P2	2	8
P3	3	6



Turnaround Time of each process is as follows:

Process	Turnaround Time
P0	$8-0 = 8$
P1	$4-1 = 3$
P2	$22-2 = 20$
P3	$14-3 = 11$

$$\text{Average Turnaround Time: } (8+3+20+11) / 4 = 10.5$$

Wait Time of each process is as follows:

Process	Wait Time
P0	$4-1 = 3$
P1	0
P2	$14-2 = 12$
P3	$8-3 = 5$

$$\text{Average Wait Time: } (3+12+5) / 4 = 5$$

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

- The discrimination towards long jobs in SJF is reduced by the strategy HRRN. Response ratio is the sum of wait time and Job time divided by the Job time. The job with the highest response time is chosen for scheduling. To start with long jobs suffer but as their wait time increases the response time ratio also increases.
- HRRN scheduling
  - Improves upon SPF scheduling
  - Still nonpreemptive
  - Considers how long process has been waiting
  - Prevents indefinite postponement

Process ID	Arrival Time	Burst Time
0	0	3
1	2	5
2	4	4
3	6	1
4	8	2

P0 is executed for 3 units

P1 is executed for 5 units

$$RR(P2) = ((8-4) + 4)/4 = 2$$

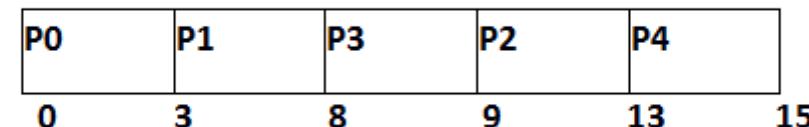
$$RR(P3) = (2+1)/1 = 3$$

$$RR(P4) = (0+2)/2 = 1$$

P3 is scheduled for 1 unit.

$$\begin{aligned}RR(P2) &= (5+4)/4 = 2.25 \\RR(P4) &= (1+2)/2 = 1.5\end{aligned}$$

P2 will be scheduled, P4 will be scheduled



# Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

# Priority Scheduling

- Ready queue is maintained in priority order, where priority of a process is:
  - determined by OS (e.g., run system processes before user processes use the CPU, average time of last CPU bursts, number of open files, memory size etc.)
  - purchased by user
  - based on corporate policy (e.g. give high priority to some project viewed as very important to company)

Process	Arrival Time	Execute Time	Priority
P0	0	5	1
P1	0	3	2
P2	0	8	1
P3	0	6	3



Turnaround Time of each process is as follows:

Process	Turnaround Time
P0	14
P1	9
P2	22
P3	6

$$\text{Average Turnaround Time: } (14+9+22+6) / 4 = 12.75$$

Wait Time of each process is as follows:

Process	Wait Time
P0	9
P1	6
P2	14
P3	0

$$\text{Average Wait Time: } (9+6+14+0) / 4 = 7.25$$

# Multilevel Queues (MLQ)

- The basic idea is to put different classes of processes in different queues.
- Processes do not move one queue to another. Different queues may follow different policies.
- There should be a policy among queues. ( e.g. one queue for the foreground processes, another for background processes etc. )

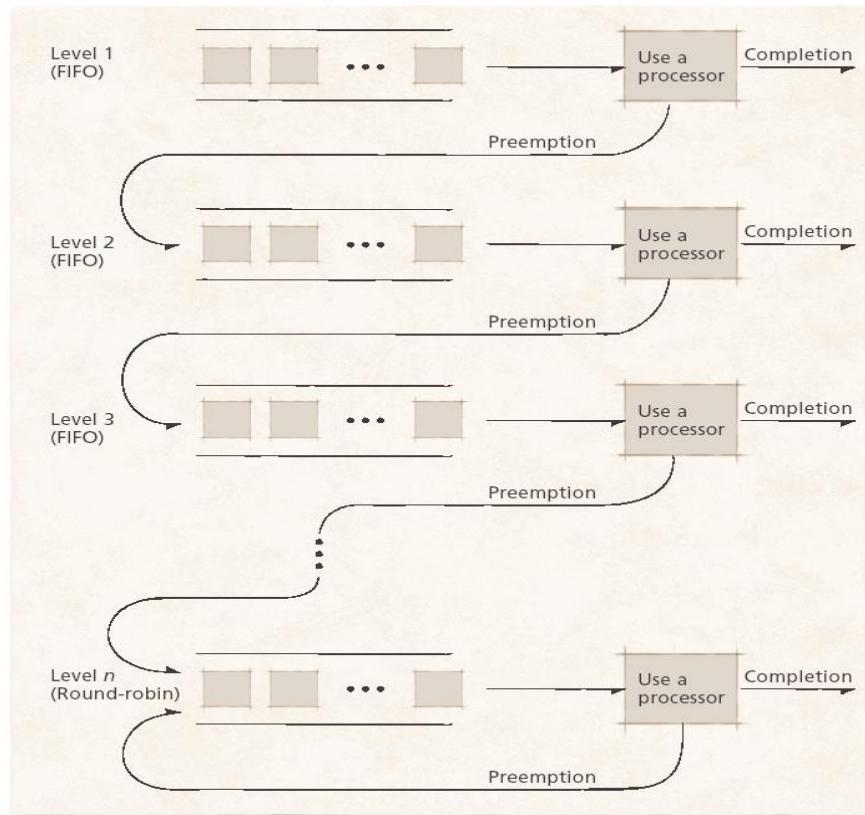
# Multilevel Feedback Queues

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

# Multilevel Feedback Queues

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

# Multilevel Feedback Queues



**Table 9.3** Characteristics of Various Scheduling Policies

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
<b>Selection function</b>	$\max[w]$	constant	$\min[s]$	$\min[s - e]$	$\max\left(\frac{w + s}{s}\right)$	(see text)
<b>Decision mode</b>	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
<b>Throughput</b>	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
<b>Response time</b>	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
<b>Overhead</b>	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
<b>Effect on processes</b>	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
<b>Starvation</b>	No	No	Possible	Possible	No	Possible

## Guaranteed Scheduling:

- System keeps track of how much CPU time each process has had since its creation. It computes the amount of CPU time each is entitled to (time since creation divided by n) and then computes the ratio of actual CPU time consumed to CPU time entitled and run the process with the lowest ratio until its ratio passes its closest competitor.

## Lottery Scheduling:

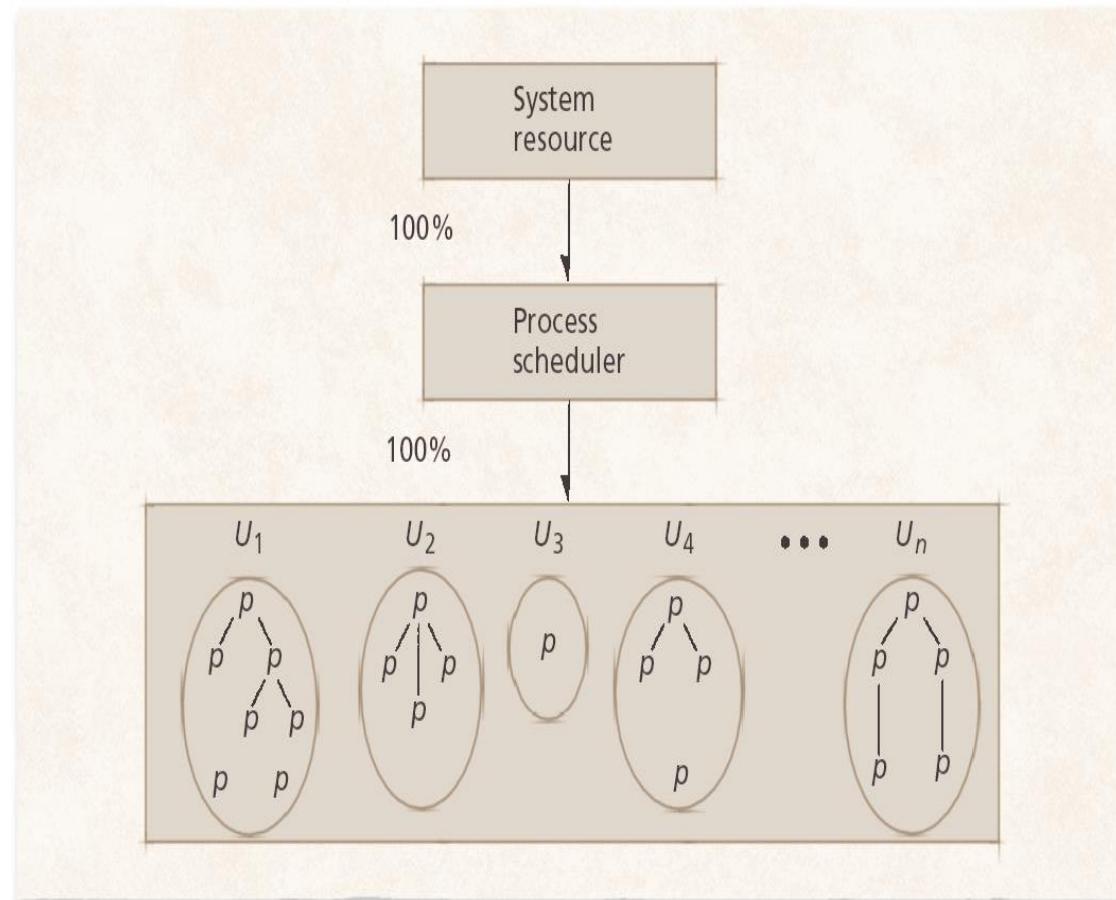
- Processes are given “lottery tickets” for system resources such as CPU time. When scheduling decision needs to be made, lottery ticket is chosen at *random* and the process holding that ticket gets the resource.

# Fair Share Scheduling

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

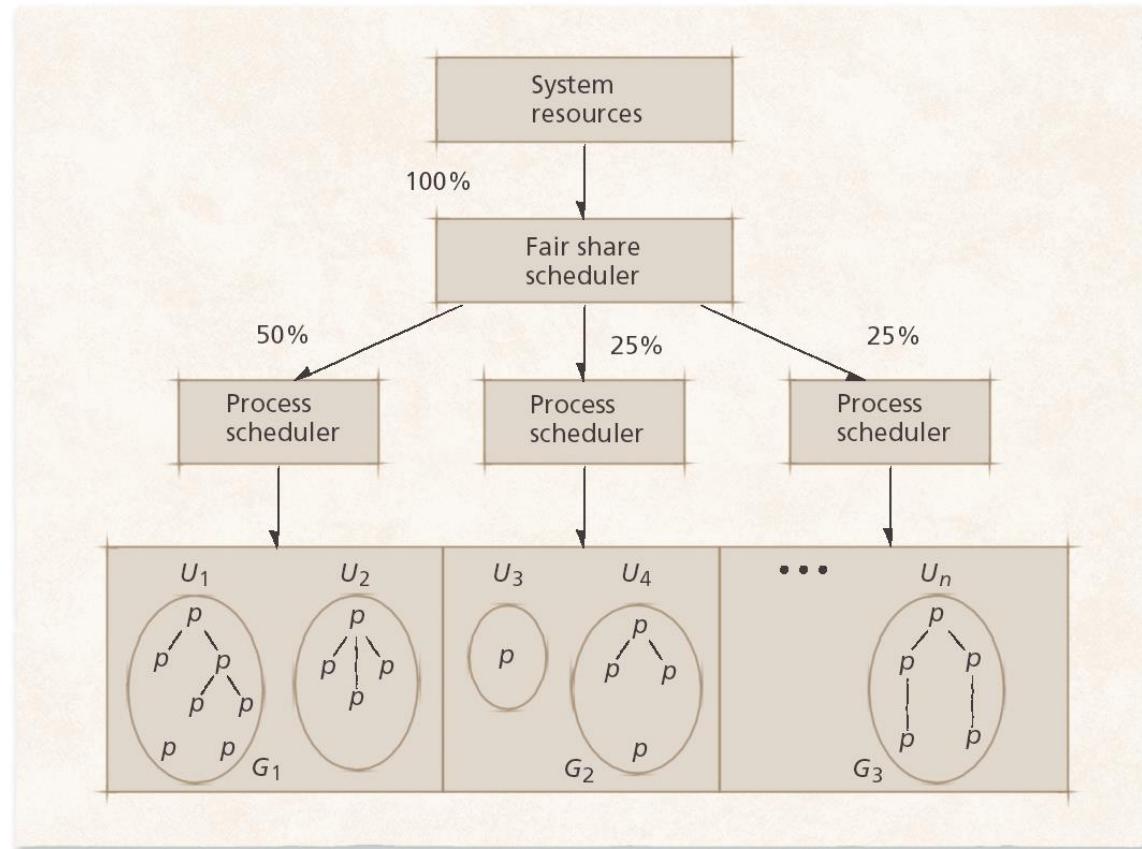
# Fair Share Scheduling

Standard UNIX process scheduler. The scheduler grants the processor to users, each of whom may have many processes.



# Fair Share Scheduling

Fair share scheduler. The fair share scheduler divides system resource capacity into portions, which are then allocated by process schedulers assigned to various fair share groups.



# Deadline Scheduling

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

# Real-Time Scheduling

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

# Real-Time Scheduling

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

# Real-Time Scheduling

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

# Process Synchronization

# Concurrent Processes

- Two processes are concurrent if their execution can overlap in time. In a single processor system, physical concurrency can be due to concurrent execution of the CPU and an I/O.
- Logical concurrency is obtained, if a CPU interleaves the execution of several processes.

# Process Relationship

- Since processes may be executing on the same physical computer, they will compete for processor time, hardware resources and slow down each other but other than that they operate independently.
- But sometimes processes communicate with each other many reasons and in various ways and the communication may be repeated many times. Processes have to compete and coordinate with other for resources.

Two fundamental relations among concurrent process:

- Competition: By virtue of sharing resources of a single system all concurrent processes compete with each other for allocation of system resources needed for their operation.
- Cooperation: A collection of related processes that represent a single logical application cooperate with each other by exchanging data and synchronization signals.
- Synchronization among cooperating concurrent processes is essential for preserving precedence relationships and for preventing concurrently related timing problems.

# Interprocess Interaction

There are three primary forms of explicit interprocess interaction:

- Interprocess Synchronization: A set of protocols and mechanisms used to preserve system integrity and consistency when concurrent processes share resources that are serially reusable.
- Interprocess Signaling: Exchange of timing signals among concurrent processes or threads used to coordinate their collective progress.
- Interprocess Communication: Concurrent cooperating processes must communicate for some purposes like exchanging of data, reporting progress and accumulating collective results.

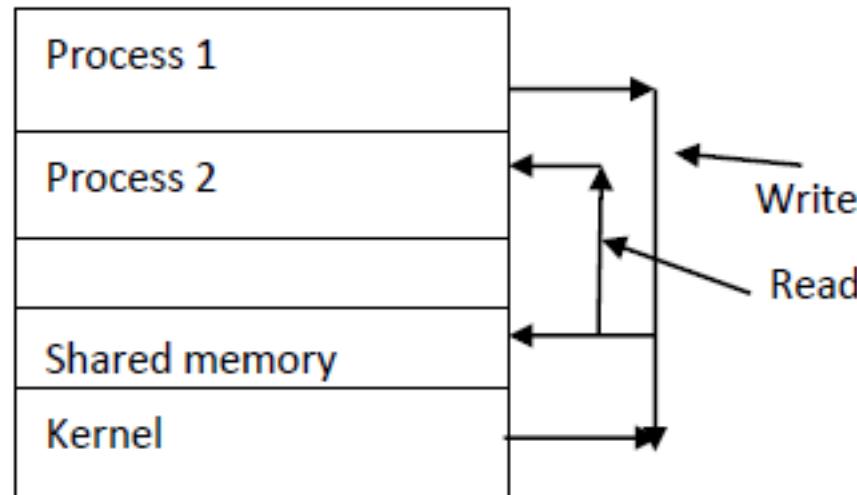
Concurrent processes generally interact through either of the following models:

- Shared memory
- Message Passing

# Shared Memory model

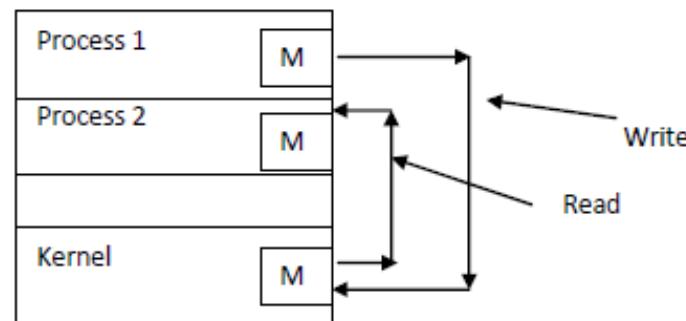
In the shared memory model, a region of memory that is shared by co-operating processes is established.

Processes can then exchange information by reading and writing a common variable or common data to shared region



# Message Passing

In the message passing model, communication takes place by means of messages exchanged between the co-operating processes using sending and receiving primitives



# Terms Related to Concurrency

- **Atomic operation** A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.
- **Race condition** A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
- **Critical section** A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
- **Mutual exclusion** The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
- **Starvation** A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.
- **Deadlock** A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
- **Livelock** A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.

# Atomic action

- In most machines, a single machine instruction is an atomic action, i.e. once the instruction has begun execution, the entire instruction executes before any interrupts are handled.
- If a time shared single processor executes processes, a timer interrupt can occur and the running process can change between any two instructions but not within a single instruction.
- So the sequence of instructions executed are to be atomic.

# Race Conditions

- Normally when two processes are running at the same time the results come out the same no matter which one finishes first.
- But it is not so when the results depend on the order of execution in Uniprocessor systems.
- In Multi core - Process 1 and process 2 are executing at the same time on different cores

Consider two processes, Producer and Consumer with following code,

```
process producer {
    while (true) {
        while (count == BUFFER_SIZE); // busy wait
        ++count;
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
    }
}
```

```
process consumer {
    while (true) {
        while (count == 0); // busy wait
        --count;
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
    }
}
```

# Race Condition

Assume *count* = 5 and both producer and consumer execute the statements *++count* and *--count*.

The results of *count* could be set to 4, 5, or 6 (but only 5 is correct).

- *++ count* could be implemented as

```
reg1 = count
```

```
reg1 = reg1 + 1
```

```
count = reg1
```

- *-- count* could be implemented as

```
reg2 = count
```

```
reg2 = reg2 - 1
```

```
count = reg2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer executes  $\text{reg1} = \text{count}$  { $\text{reg1} = 5$ }  
S1: producer executes  $\text{reg1} = \text{reg1} + 1$  { $\text{reg1} = 6$ }  
S2: consumer executes  $\text{reg2} = \text{count}$  { $\text{reg2} = 5$ }  
S3: consumer executes  $\text{reg2} = \text{reg2} - 1$  { $\text{reg2} = 4$ }  
S4: producer executes  $\text{count} = \text{reg1}$  { $\text{count} = 6$ }  
S5: consumer executes  $\text{count} = \text{reg2}$  { $\text{count} = 4$ }

- Variable count represents a shared resource

# Critical Section Problem

- Concurrent access to shared data results in data inconsistency.
- Examples are variables not recording all changes; a process may read inconsistent values ; final value of the variable may be inconsistent.
- Maintaining data consistency in multi-process environment requires mechanisms to ensure orderly execution of cooperating processes i.e. processes are to be synchronized such a way that one process can access the variable at any one time.
- This is referred as the mutual exclusion problem.

Critical section: A critical section is a code segment, common to n cooperating processes, in which the processes may be accessing common/shared variables.

A critical section environment consists of

- Entry Section : Core requesting entry into critical section
- Critical Section : Code in which only one process can execute at any one time
- Exit Section : The end of critical section, releasing or allowing others in
- Remainder section : Rest of the code after the critical section.

The solution to the mutual exclusion must satisfy the following requirements:

- Mutual Exclusion condition: only one process can execute its critical section at any one time.
- Progress: When no process is executing in its critical section, any process that requests entry to its critical section must be permitted to enter without delay.
- Bounded Waiting: When two or more processes compete to enter the critical section, a fair chance should be given all processes and there should not be any indefinite postponement.

Two general approach used to handle critical sections in OS

- *Preemptive kernels*, which allows a process to be pre-empted while it is running in a kernel mode,
- *Non-preemptive kernels*, which does not allow a process running in a kernel mode to pre-empt.

# Simplest Solution

## Disabling interrupts

- Each process disable all interrupts after entering CS and re-enable before leaving

## Problem

- if user process does not turn on -> end of the system
- Kernel has to frequently disable interrupts for few instructions while it is updating variables or lists.

# Software Solutions

- Only 2 processes,  $P_i$  and  $P_j$
- General structure of process  $P_i$  (other process  $P_j$ )

```
do {
    entry section
    critical section
    exit section
    remainder section
} while (1);
```
- Processes may share some common variables to synchronize their actions.

- Given below is a simple piece of code containing the components of a critical section used for solving CS problem.

```
do {  
    while ( turn != i );           /* Entry Section */  
    /* critical section */  
    turn = j;                     /* Exit Section */  
    /* remainder section */  
} while(TRUE);
```

# Algorithm 1

- Shared variables:

- int turn;

initially turn = 0

- turn - i  $\Rightarrow P_i$  can enter its critical section

- Process  $P_i$

```
do {  
    while (turn != i) ;  
    critical section  
    turn = j;  
    remainder section  
} while (1);
```

- Satisfies mutual exclusion, but not progress
  - Suppose that  $P_i$  finishes its critical section quickly and sets  $turn = j$ ; both processes are in their non-critical parts.  $P_i$  is quick also in its non-critical part and wants to enter the critical section. As  $turn == j$ , it will have to wait even though the critical section is free.
  - Moreover, the behaviour inadmissibly depends on the relative speed of the processes

# Algorithm 2

- Shared variables
  - boolean flag[2]; initially flag [0] = flag [1] = false.
  - flag [i] = true  $\Rightarrow P_i$  ready to enter its critical section

- Process  $P_i$ 

```
do {
    flag[i] := true;
    while (flag[j]) ;      critical section
    flag [i] = false;
    remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

# Algorithm 3 (Peterson's Solution)

- Combined shared variables of algorithms 1 and 2.

- Process  $P_i$

```
do {
    flag [i]:= true;
    turn = j;
    while (flag [j] and turn = j) ;
        critical section
        flag [i] = false;
        remainder section
    } while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.

# Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes  $P_i$  and  $P_j$  receive the same number, if  $i < j$ , then  $P_i$  is served first; else  $P_j$  is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,4,5...

# Bakery Algorithm

- Notation  $\leqslant$  lexicographical order (ticket #, process id #)
  - $(a,b) < (c,d)$  if  $a < c$  or if  $a = c$  and  $b < d$
  - $\max(a_0, \dots, a_{n-1})$  is a number,  $k$ , such that  $k \geq a_i$  for  $i = 0, \dots, n - 1$
- Shared data

```
boolean choosing[n];  
int number[n];
```

Data structures are initialized to false and 0 respectively

# Bakery Algorithm

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], ..., number [n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        while (choosing[j]) ;
        while ((number[j] != 0) && (number[ j] < number[i])) ;
    }
    critical section
    number[i] = 0;
    remainder section
} while (1);
```

H/W solutions

# Locks

- Critical sections must be protected with locks to guarantee the property of mutual exclusion.
- Using a simple lock variable and manipulating it will not work, because the race condition will now occur when updating the lock.
- Better alternative is to use atomic instructions
- Threads/Processes that want to access a critical section must try to acquire the lock, and proceed to the critical section only when the lock has been acquired.

# Synchronization via Locks

```
while (true) {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
}
```

- Lock(x)

```
    var x: shared integer;  
Lock(x) : begin  
    var y: integer;  
    y = x;  
    while y =1 do y = x; // wait until gate is open  
    x =1;  
    end;
```

- Unlock(x)

```
Unlock(x)  
x = 0;
```

If requested lock is held by other threads/processes

Two Options:

1. the thread/process could wait busily, constantly polling to check if the lock is available.
2. the thread/process could be made to give up its CPU, go to sleep (i.e., block), and be scheduled again when the lock is available.

The former way of locking is usually referred to as a spinlock, while the latter is called a regular lock or a mutex.

# HARDWARE SOLUTIONS

- To provide a generalized solution to the critical section problem, some sort of lock must be set to prevent other processes from entering their critical sections simultaneously, and must release the lock when exiting the critical section.
- The hardware required to support critical sections must have, indivisible instructions or atomic operations.
- These operations are guaranteed to operate as a single instruction without interruption .

# Synchronization Hardware

Test and modify the content of a word atomically

```
var x: shared integer;  
Test-and-set (x) :  
begin  
    var y: integer;  
    y = x;  
    if y = 0 then x = 1  
end;
```

**Usage:**

**Lock**

```
var x: shared integer
```

**lock(x):**

```
begin
```

```
    var y: integer;
```

```
    repeat y = test-and-set (x)
```

```
        until y = 0;
```

```
end;
```

**Unlock**

```
x = 0;
```

# Mutual Exclusion with Test-and-Set

- Shared data:

```
boolean lock = false;
```

- Process  $P_i$

```
do {
```

```
    while (TestAndSet(lock)) ;
```

```
        critical section
```

```
        lock = false;
```

```
        remainder section
```

```
}
```

# Synchronization Hardware

- Atomically swap two variables.

```
Swap (var x,y: boolean);
    var temp: boolean;
    begin
        temp = x;
        x =y;
        y = temp;
    end
```

## Usage

- Lock

`p = true;`

`repeat swap (S,P) until p = false;`

- Unlock

`s = false`

# Mutual Exclusion with Swap

- Shared data (initialized to **false**):  
boolean lock;

- Process  $P_i$

```
do {
    key = true;
    while (key == true)
        Swap(lock, key);
    critical section
    lock = false;
    remainder section
}
```

# Classical Problems of Synchronization

## Producer Consumer Bounded Buffer Problem

Two classes of processes

Producers, which produce times and insert them into a buffer.

Consumers, which remove items and consume them.

Issues

Overflow Condition: if the producer encounters a full buffer? Block it.

Underflow Condition : if the consumer encounters an empty buffer? Block it.

## **Reader/Writers Problem**

Two classes of processes.

Readers, which can work concurrently.

Writers, which need exclusive access.

### **Rules**

1. Must prevent 2 writers from being concurrent.
2. Must prevent a reader and a writer from being concurrent.
3. Must permit readers to be concurrent when no writer is active.
4. Perhaps want fairness (i.e., freedom from starvation).

### **Variants**

Writer-priority readers/writers.

Reader-priority readers/writers.

Reader Writer Alternates

## Dining Philosophers Problem

A classical problem. Some number of philosophers spend time thinking and eating. Being poor, they must share a total of five chopsticks.

### Rules:

One chopstick between each philosopher

Philosopher first picks up one (if it is available) then the second (if available)

Only puts down chopsticks after eating for a while

Deadlock can occur, e.g., if all philosophers simultaneously pick up chopstick to his/her left.

### Solutions:

Allow only  $n-1$  (if we have  $n$  chopsticks) at table

Philosopher only picks up one chopstick if he/she can pick up both.

Odd philosopher first picks up left, even philosopher first picks up right.

# Synchronization Primitives

# Semaphore

Software-based synchronization mechanism/tool ( originally proposed by Dijkstra)

It avoids busy waiting i.e. wasting of CPU time

A semaphore S is an integer variable

Two indivisible operations can modify **S**:

**P(S) & V(S) or acquire() & release() or wait() & signal()**

## Indivisible testing & modification of value

```
acquire() {  
    while (value <= 0)  
        ; // no-op  
    value--;  
}  
  
release() {  
    value++;  
}
```

CourseSmart

```
WAIT ( S ):  
    while ( S <= 0 );  
    S = S - 1;
```

```
SIGNAL ( S ):  
    S = S + 1;
```

FORMAT:

```
wait( mutex );           <-- Mutual exclusion: mutex init to 1.  
CRITICAL SECTION  
signal( mutex );  
REMAINDER SECTION
```

Instead of loop on busy, suspend can be used:

Block on semaphore == False,

Wakeup on signal ( semaphore becomes True),

There may be numerous processes waiting for the semaphore, so keep a list of blocked processes,

Wakeup one of the blocked processes upon getting a signal ( choice of who depends on strategy ).

To PREVENT looping, we redefine the semaphore structure as:

Semaphore  $S$  – system object

With each semaphore there is an associated waiting queue.

Each entry in the waiting queue has two data items (object properties):

**value** (of type integer)

pointer to next record in the queue

P(S) and V(S) operations

P(S) : If  $S \geq 1$  then  $S := S - 1$

else block the process on the semaphore queue;

V(S) : If some processes are blocked on the semaphore S

then unblock a process

else  $S = S + 1$ ;

```
typedef struct {  
    int          value;  
    struct process *list; /* linked list of process id waiting on S */  
} SEMAPHORE
```

```
SEMAPHORE s;  
wait(s) {  
    s.value = s.value - 1;  
    if ( s.value < 0 ) {  
        add this process to s.L;  
        block;  
    }  
}
```

```
SEMAPHORE s;  
signal(s) {  
    s.value = s.value + 1;  
    if ( s.value <= 0 ) {  
        remove a process P from s.L;  
        wakeup(P);  
    }  
}
```

# Counting semaphore (General semaphore)

- An integer value used for signalling among processes and the integer value can range over an unrestricted domain
- Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment.
- The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process.
- Spin-lock is a general (counting) semaphore using busy waiting instead of blocking
- Blocking and switching between threads and/or processes may be much more time demanding than the time waste caused by short-time busy waiting

# Binary Semaphore

- A semaphore that takes on only the values 0 and 1 on which processes can perform two indivisible operations i.e. changing the integer value from 0 to 1 or 1 to 0
- Often known as mutex lock. A key difference between mutex and binary semaphore is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).

# Binary Semaphore

```
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};

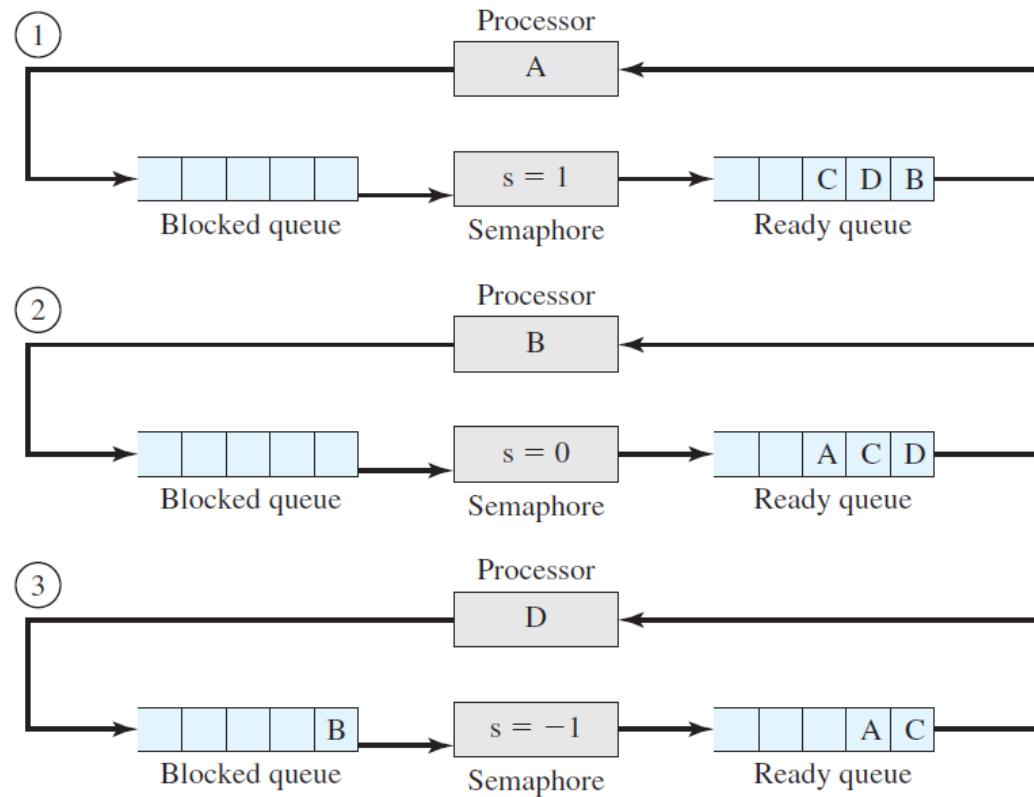
void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
```

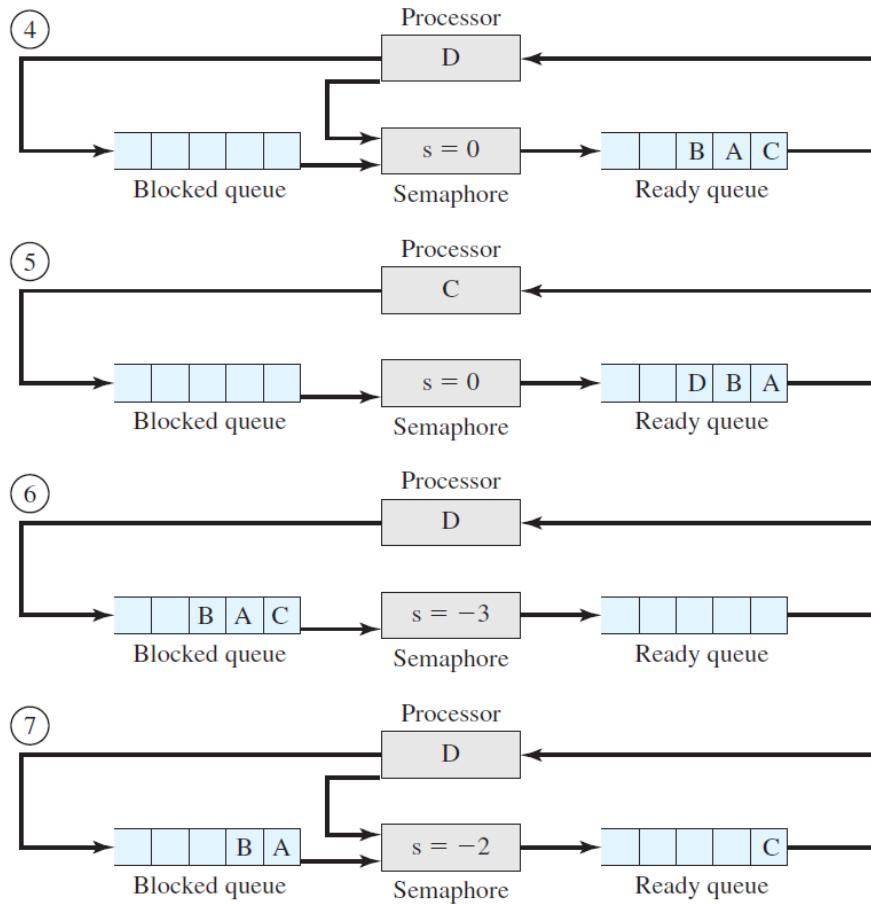
```
void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

- For both counting semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore.
- The fairest removal policy is first-in-first-out (FIFO):The process that has been blocked the longest is released from the queue first; a semaphore whose definition includes this policy is called a **strong semaphore**.
- A semaphore that does not specify the order in which processes are removed from the queue is a **weak semaphore**.

# Example

- Processes A, B, and C depend on a result from process D.
- Initially (1), A is running; B, C, and D are ready; and the semaphore count is 1, indicating that one of D's results is available.
- When A issues a semWait instruction on semaphore s, the semaphore decrements to 0, and A can continue to execute; subsequently it rejoins the ready queue.
- Then B runs (2), eventually issues a semWait instruction, and is blocked, allowing D to run (3).When D completes a new result, it issues a semSignal instruction, which allows B to move to the ready queue (4).
- D rejoins the ready queue and C begins to run (5) but is blocked when it issues a semWait instruction.
- Similarly, A and B run and are blocked on the semaphore, allowing D to resume execution (6).When D has a result, it issues a semSignal, which transfers C to the ready queue.
- Later cycles of D will release A and B from the Blocked state.





Semaphores can be used to force synchronization ( precedence ) if the preceding process does a signal at the end, and the follower does wait at beginning. For example, here we want P1 to execute before P2.

**P1:**

statement 1;  
signal ( synch );

**P2:**

wait ( synch );  
statement 2;

## **DEADLOCKS:**

May occur when two or more processes try to get the same multiple resources at the same time.

**P1:**

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

**P2:**

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

# Bounded-Buffer Problem using Semaphores

Three semaphores

- `mutex` – for mutually exclusive access to the buffer – initialized to 1
- `used` – counting semaphore indicating item count in buffer – initialized to 0
- `free` – number of free items – initialized to `BUF_SZ`

```
void producer() {  
    while (1) { /* Generate new item into nextProduced */  
        wait(free);  
        wait(mutex);  
        buffer[in] = nextProduced; in = (in + 1) % BUF_SZ;  
        signal(mutex);  
        signal(used);  
    }  
}
```

```
void consumer() {
    while (1) { wait(used);
                wait(mutex);
                nextConsumed = buffer[out]; out = (out + 1) % BUF_SZ;
                signal(mutex);
                signal(free);
                /* Process the item from nextConsumed */
            }
}
```

# Critical Regions and Monitors

# Critical Regions

- High-level synchronization construct
- A shared variable  $v$  of type  $T$ , is declared as:  
 **$v: shared T$**
- Variable  $v$  accessed only inside statement  
**region  $v$  when  $B$  do  $S$**   
where  $B$  is a boolean expression.
- While statement  $S$  is being executed, no other process can access variable  $v$ .

# Critical Regions

- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement, the Boolean expression  $B$  is evaluated. If  $B$  is true, statement  $S$  is executed. If it is false, the process is delayed until  $B$  becomes true and no other process is in the region associated with  $v$ .

# Example – Bounded Buffer Producer Consumer Problem

Shared data:

```
struct buffer {
    int pool[n];
    int count, in, out;
}
```

# Bounded Buffer Producer Process

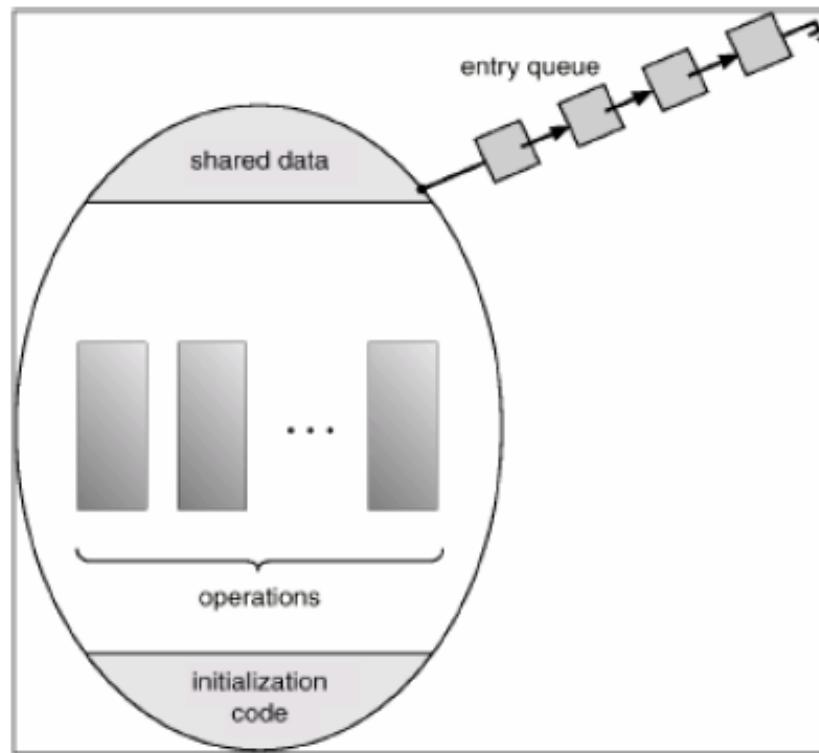
- Producer process inserts **nextp** into the shared buffer
  - region buffer when( count < n) {**
  - pool[in] = nextp;**
  - in:= (in+1) % n;**
  - count++;**
  - }**
- Consumer process removes an item from the shared buffer and puts it in **nextc**
  - region buffer when (count > 0) {**
  - nextc = pool[out];**
  - out = (out+1) % n;**
  - count--;**
  - }**

# Monitors

- High-level synchronization construct which allows the safe sharing of an abstract data type among concurrent processes.
- A monitor is a class, in which all data are private, and with the special restriction that only one method within any given monitor object may be active at the same time.
- Monitor methods can only access the shared data within the monitor, they cannot access an outside variable and any data passed to them only as parameters.
- The variables or data local to a monitor cannot be directly accessed from outside the monitor.

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        ...
    }
    procedure body P2 (...) {
        ...
    }
    procedure body Pn (...) {
        ...
    }
}
initialization code
}
```

# Schematic view of a Monitor



The monitor construct allows only one process at a time to be active within the monitor

To allow a process to wait within the monitor, a condition variable must be declared as

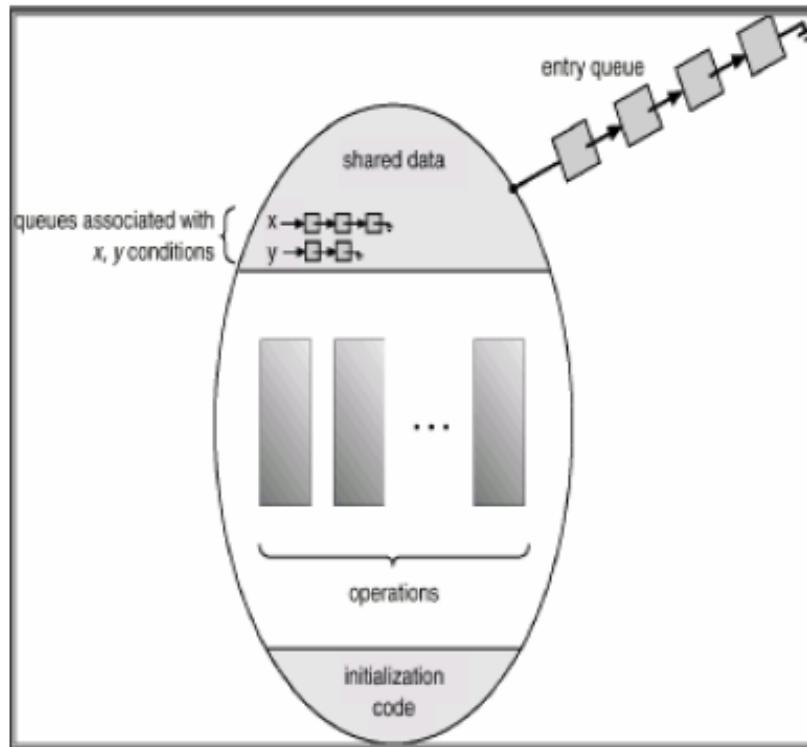
```
condition x, y;
```

Condition variable can only be used with the operations `wait()` and `signal()`.

The operation `x.wait()` means that the process invoking this operation is suspended until another process invokes `x.signal()`.

The `x.signal()` operation resumes exactly one suspended process. If no process is suspended, then the *signal* operation has no effect.

# Monitor with condition variables



Suppose that when the `x.signal()` operation is invoked by a process P, there is a suspended process Q associated with condition x.

If the suspended process Q is allowed to resume its execution, the signaling process P must wait.

If not, both P and Q would be active simultaneously within the monitor.

Now two possibilities exist :

i) Signal and wait : P either waits until Q leaves the monitor or waits for another condition

ii) Signal and continue : Q either waits until P leaves the monitor or waits for another condition

The advantage of monitors is the flexibility they allow in scheduling the processes waiting in queues.

Drawbacks:

The major drawback of monitors is the absence of concurrency if a monitor encapsulates the resource, since only one process can be active within a monitor at a time.

Another drawback is possibility of deadlocks in the case of nested monitor calls.

# Interprocess Communication

# Message passing systems

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment where the communicating processes may reside on different computers connected by a network.
- A message passing facility provides at least two operations: send (message) and receive (message).

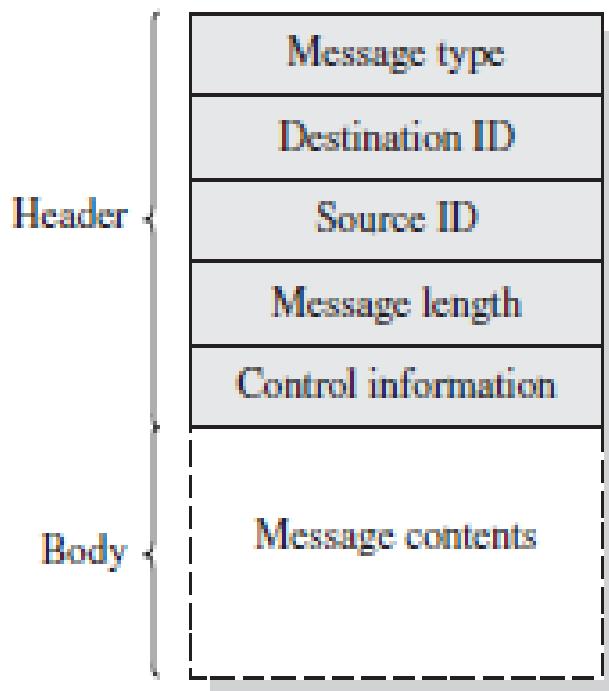
# Issues related to message passing systems

- Message size:

Messages sent by a process can be of either fixed or variable size. If only fixed - sized messages can be sent, the system – level implementation is straight forward.

Variable sized messages require a more complex system level implementation.

# Message Format



- Communication link:

If two processes want to communicate, they must send messages to and receive messages from each other, a communication link must exist between them.

A communication link 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.

# Direct or indirect communication

Naming: Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication.

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

This scheme exhibits symmetry in addressing; that is both the sender process and receiver process must name the other to communicate.

A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender.

Here, the send () and receive () primitives are defined as:

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

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

With indirect communication, the messages are sent to and received from mail boxes or ports.

A mail box can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

Each mail box has a unique identification.

A process can communicate with some other process via a number of different mail boxes.

Two processes can communicate only if the processes have a shared mail box.

The send () and receive() primitives are defined as:

Send (A, message) – send a message to mail box A

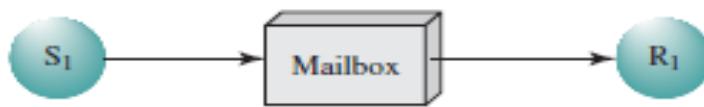
Receive (A, message) – receive a message from mail box A.

In this scheme, a communication link has the following properties –

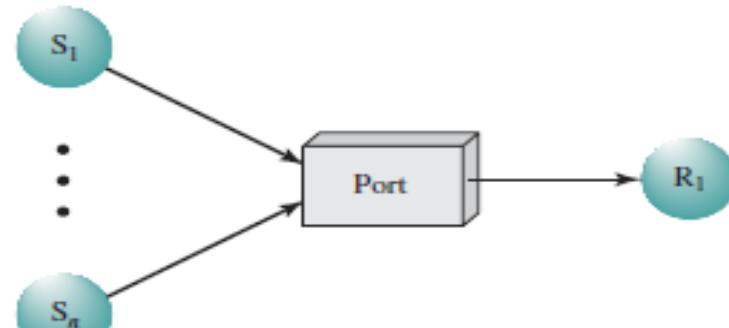
A link is established between a pair of processes only if both members of the pair have a shared mail box.

A link may be associated with more than two processes.

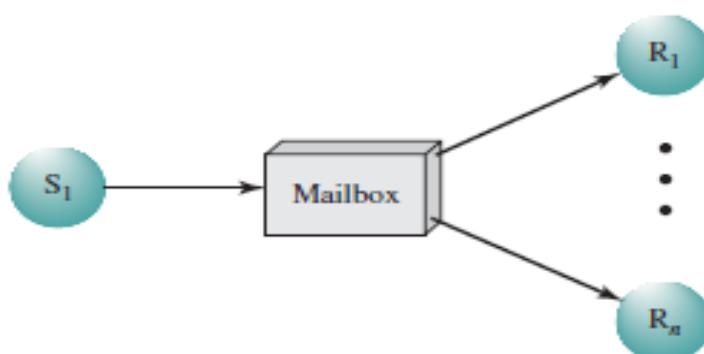
Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mail box.



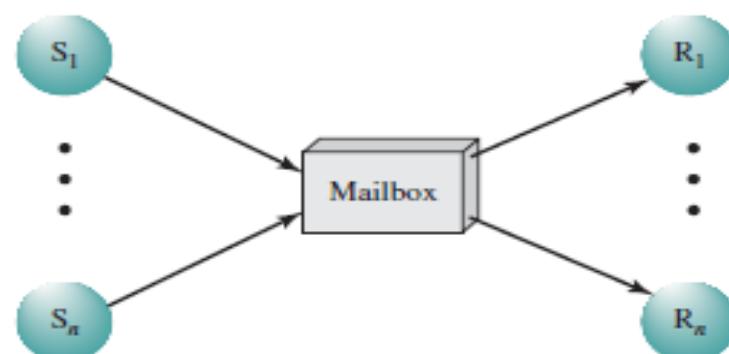
(a) One to one



(b) Many to one



(c) One to many



(d) Many to many

A mail box may be owned either by a process or by the OS.

- If the mail box is owned by a process, then we distinguish between the owner and the user. When a process that owns the mail box terminates, the mail box disappears. Any process that subsequently sends a message to this mail box must be notified that the mail box no longer exists.
- A mail box owned by the OS is independent and is not attached to any particular process. The OS then must provide a mechanism that allows a process to do the following:
  - Create a new mail box
  - Send and receive messages through the mail box
  - Delete a mail box

# Synchronous or asynchronous communication

Communication between processes takes place through calls to send () and receive () primitives. Message passing may be either blocking or non blocking – also known as synchronous and asynchronous.

- Blocking send: The sending process is blocked until the message is received by the receiving process or the mail box.
- Non blocking send: The sending process sends the message and resumes operation.
- Blocking receive: The receiver blocks until a message is available.
- Non blocking receive: The receiver retrieves either a valid message or a null.
- When both send () and receive () are blocking, we have a rendezvous between the sender and the receiver.

# Automatic or explicit buffering

- Buffering: Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- Such queues can be implemented in three ways –
  - Zero capacity: The queue has the maximum length of zero; thus the link cannot have any messages waiting in it.
  - Bounded capacity: The queue has finite length  $n$ ; thus, at most  $n$  messages can reside in it.
  - Unbounded capacity: The queue's length is infinite; thus any number of messages can wait in it. The sender never blocks.
- The zero capacity buffer is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

```
const int
    capacity = /* buffering capacity */ ;
    null = /* empty message */ ;
int i;
void producer()
{
    message pmsg;
    while (true) {
        receive (mayproduce,pmsg);
        pmsg = produce();
        send (mayconsume,pmsg);
    }
}
void consumer()
{
    message cmsg;
    while (true) {
        receive (mayconsume,cmsg);
        consume (cmsg);
        send (mayproduce,null);
    }
}
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1;i<= capacity;i++) send (mayproduce,null);
    parbegin (producer,consumer);
}
```

# Barrier Synchronization:

This synchronization mechanism is for group of processes.

Some applications are divided into phases and the set of processes can move on to next phase only when all the processes in that group complete their execution in the current phase.

A process wait on the barrier until all other processes reach the barrier.

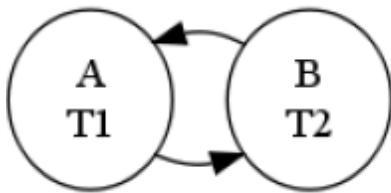
# Deadlocks

# Process Deadlocks

- *Definition : A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause*
- i.e. A deadlock is a situation where a process or a set of processes is blocked, waiting on an event which may be the release of computing resource or other hardware resources, that can only be caused by a member of the set and that will never occur.
- The formation and existence of deadlocks in a system lowers system efficiency.
- Therefore to avoid performance degradation, a system should be deadlock free or deadlocks should be quickly detected and recovered.

# Examples

- System has 2 tape drives. P1 and P2 each hold one tape drive and each needs the other one



- Semaphores A and B each initialized to 1

$P_0$

$wait(A)$

$wait(B)$

$P_1$

$wait(B)$

$wait(A)$

# Deadlock vs Starvation

- A process is *deadlocked* if it is waiting for an event that will never occur. Typically, more than one process will be involved in a deadlock and the processes are involved in a circular wait.
- Starvation occurs when a process waits for a resource that continually becomes available but is never assigned to that process because of priority or a flaw in the design of a scheduler e.g. the process is ready to proceed but never gets the CPU.

- Two major differences between deadlock and starvation:
  1. In starvation, it is not certain that a process will ever get the requested resource, whereas a deadlocked process is permanently blocked because the required resource never becomes available unless external actions are taken.
  2. In starvation, the resource under contention is in continuous use whereas in deadlock resources are not used as the processes are blocked.

# Resources:

Resource is a commodity required by a process to execute.

Under normal operation, a resource allocation proceed like this::

1. Request a resource (suspend until available if necessary ).
2. Use the resource.
3. Release the resource.

Resources can be of several types:

- Serially Reusable Resources

e.g. CPU cycles, memory space, I/O devices, files

A process acquires, uses and then releases the resource.

- Consumable Resources

Produced by a process, needed by a process

e.g. Messages, buffers of information, interrupts

A process creates, acquires and then use the resource. Resource ceases to exist after it has been used.

Another classification is

- Preemptable - the resource can be taken away from its current owner (and given back later). An example is memory.
- Non-preemptable- the resource cannot be taken away. An example is a printer.

# Fundamental causes of deadlocks

The following four conditions are *necessary* for a deadlock to occur.

- Mutual exclusion: A resource can be assigned to at most one process at a time (no sharing).
- Hold and wait: A process holding a resource is permitted to request another. i.e. a process holding at least one resource is waiting to acquire additional resources held by other processes
- No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task i.e. A process must release its resources; they cannot be taken away.
- Circular wait: There must be a chain of processes such that each member of the chain is waiting for a resource held by the next member of the chain.

There exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ , ...,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

# System model

System can be modelled using Resource allocation graph. Resource types are denoted by,  $R_1, R_2, \dots, R_n$ , and each resource  $R$  has  $W_i$  instances.

Resource Allocation Graph (RAG)

A set of vertices  $V$  and a set of edges  $E$

$V$  is partitioned into 2 types

$P = \{P_1, P_2, \dots, P_n\}$  - the set of processes in the system

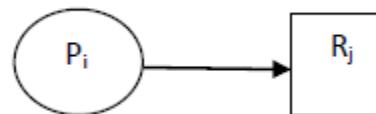
$R = \{R_1, R_2, \dots, R_n\}$  - the set of resource types in the system

## Two kinds of edges

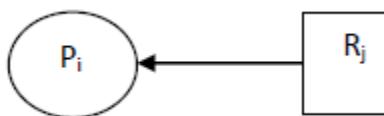
Request edge - Directed edge  $P_i \rightarrow R_j$

Assignment edge - Directed edge  $R_j \rightarrow P_i$

$P_i$  requests an instance of  $R_j$

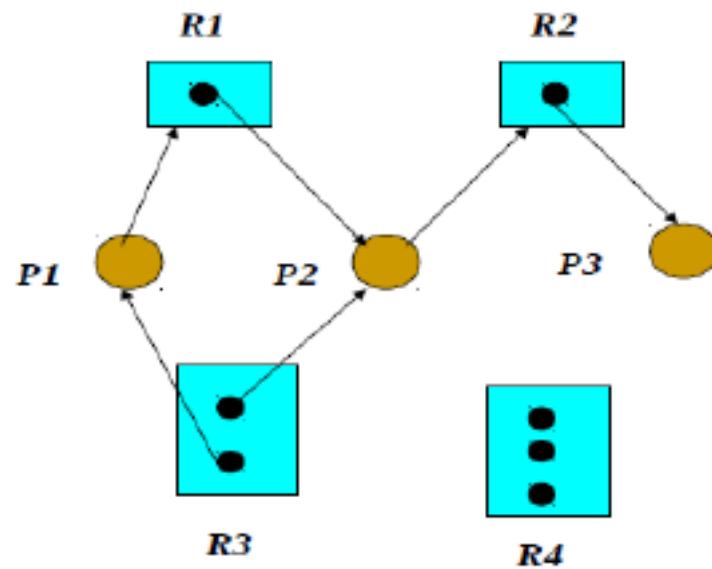


$P_i$  is holding an instance of  $R_j$

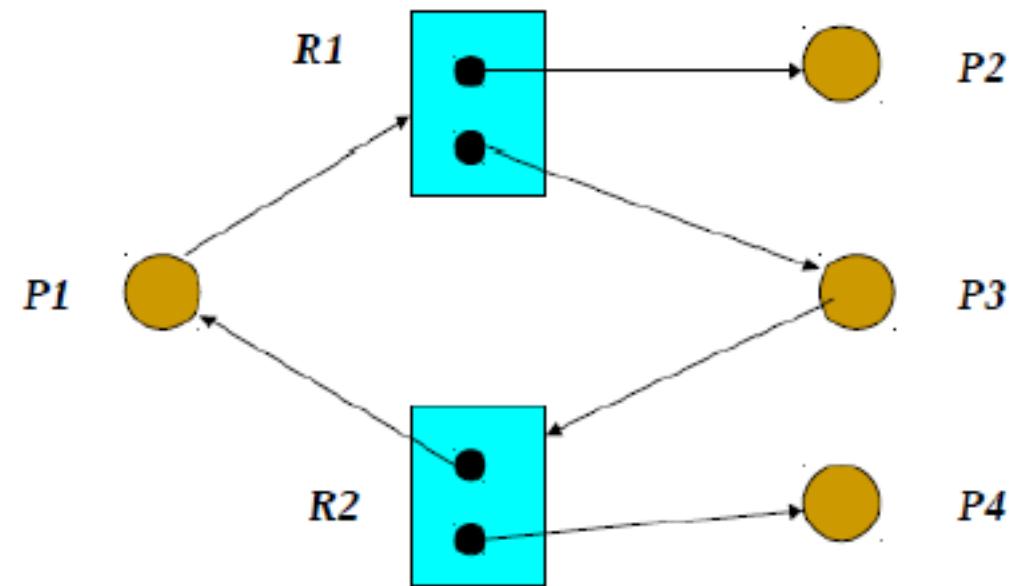


- The request for resources by processes can be modelled using RAG.
- If a resource allocation graph contains no cycles, then no process is deadlocked.
- If a RAG contains a cycle, then a deadlock may exist. Therefore, a cycle means deadlock is *possible*, but not necessarily *present*.
- A cycle is not sufficient proof of the presence of deadlock. A cycle is a *necessary* condition for deadlock, but not a *sufficient* condition for deadlock.

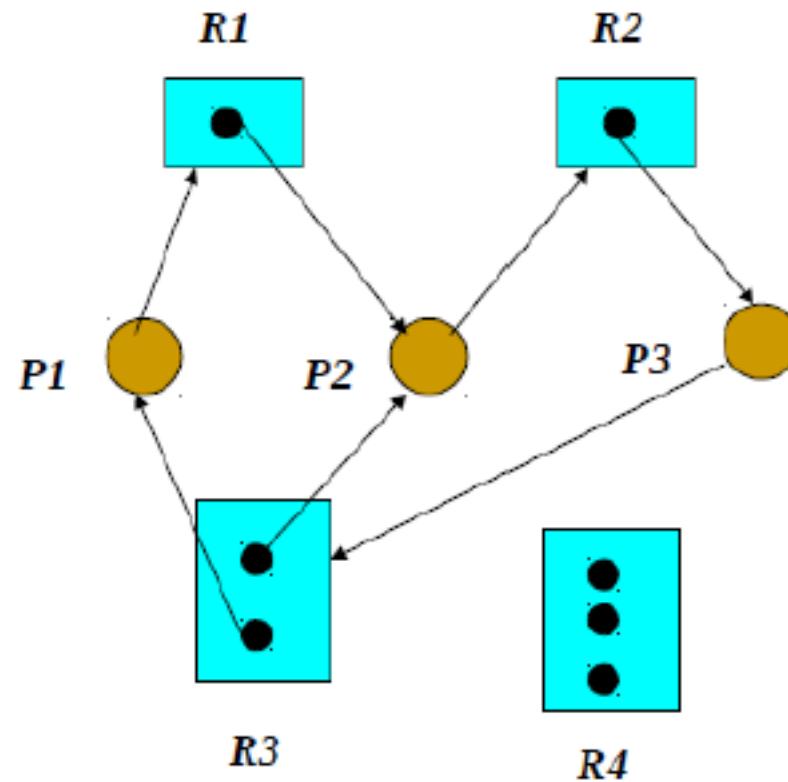
# RAG with no cycles



# RAG with cycles



# RAG with cycles and deadlocks



If RAG contains no cycles then NO DEADLOCK.

If RAG contains a cycle and

if only one instance per resource type, then deadlock

if several instances per resource type, possibility of deadlock.

# *Four strategies for handling with deadlocks*

- Ignore the problem - Ostrich Algorithm Popular in Distributed Systems.
  - Reasonable if
    - deadlocks occur very rarely
    - cost of prevention is high
  - UNIX and Windows takes this approach
- Deadlock Prevention - Prevent deadlocks by violating one of the 4 necessary conditions.
- Deadlock Avoidance - Avoid deadlocks by carefully deciding when to allocate resources.
- Deadlock Detection and recovery - Detect deadlocks and recover from them

# Deadlock Prevention

Restrain the ways request can be made

- Prevent Mutual Exclusion
  - Not required for sharable resources (e.g., read-only files);
  - Must hold for non-sharable resources
  - Prevention not possible, since some devices are intrinsically non-sharable

- Prevent Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

## 1. Pre-allocate

- do not pick up one chopstick if you cannot pick up the other
- for a process that copies data from DVD drive to a file on disk and then prints it from there:
  - request DVD drive
  - request disk file
  - request printer

2. A process can request resources only when it has none

- request DVD drive and disk file
- release DVD drive and disk file
- request disk file and printer (no guarantee data will still be there)
- release disk file and printer
- Disadvantages: inefficient, possibility of starvation.

- Prevent No Preemption
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
  - some resources cannot be feasibly preempted (e.g., printers, tape drives)
- Prevent Circular Wait
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

# Deadlock Avoidance

- Requires that the system has some additional *a priori* information available
- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

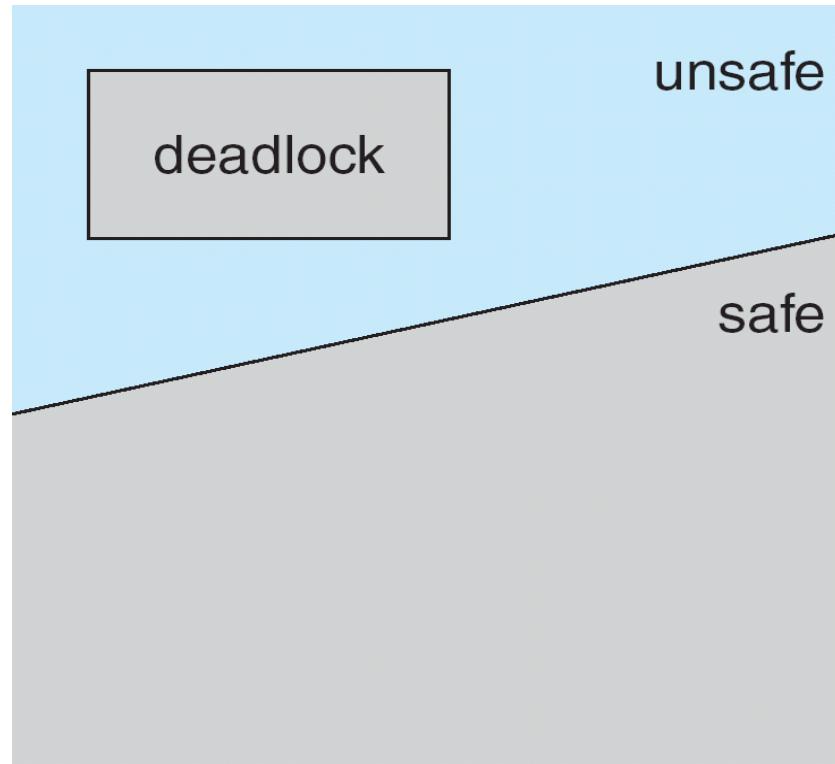
# Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence  $\langle P_1, P_2, \dots, P_n \rangle$  of ALL the processes in the systems such that for each  $P_i$ , the resources that  $P_i$  can still request can be satisfied by currently available resources + resources held by all the  $P_j$ , with  $j < i$
- That is:
  - If  $P_i$  resource needs are not immediately available, then  $P_i$  can wait until all  $P_j$  have finished
  - When  $P_j$  is finished,  $P_i$  can obtain needed resources, execute, return allocated resources, and terminate
  - When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on

# Basic Facts

- If a system is in safe state  $\Rightarrow$  no deadlocks
- If a system is in unsafe state  $\Rightarrow$  possibility of deadlock
- Avoidance  $\Rightarrow$  ensure that a system will never enter an unsafe state.

# Safe, Unsafe, Deadlock State



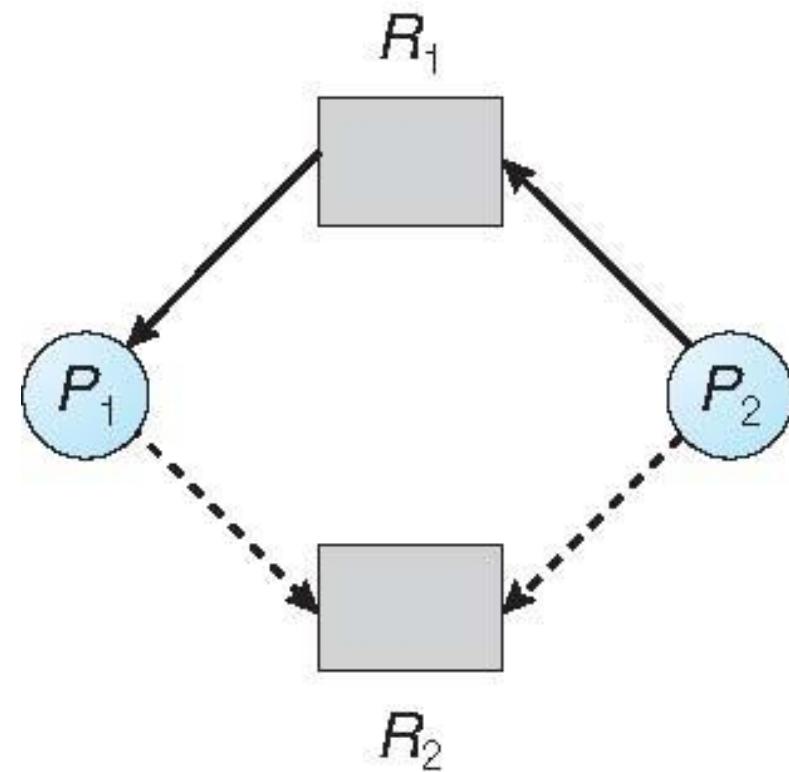
# Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

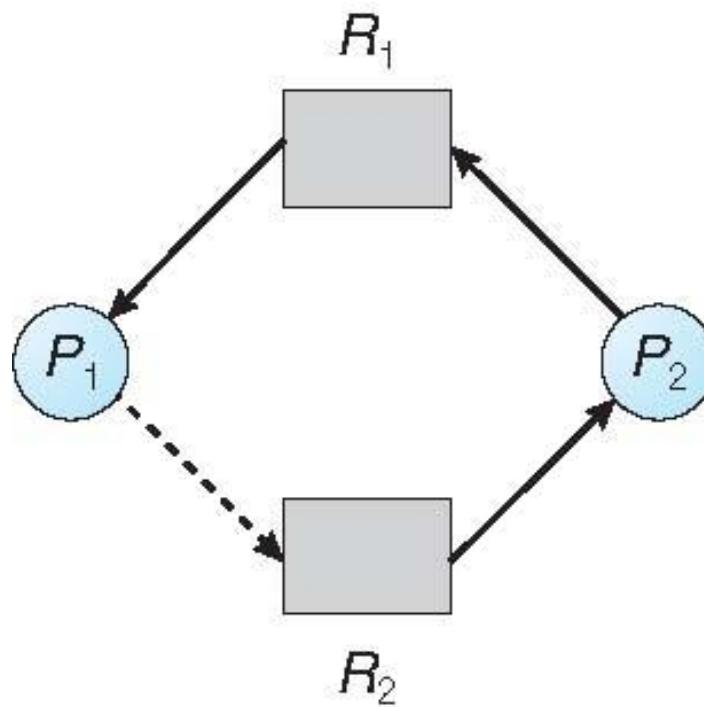
# Resource-Allocation Graph Scheme

- **Claim edge**  $P_i \rightarrow R_j$  indicated that process  $P_j$  may request resource  $R_j$ ; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

# Resource-Allocation Graph



# Unsafe State In Resource-Allocation Graph



# Resource-Allocation Graph Algorithm

- Suppose that process  $P_i$  requests a resource  $R_j$
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

# Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

# Data Structures for the Banker's Algorithm

- Let  $n$  = number of processes, and  $m$  = number of resources types.
- **Available:** Vector of length  $m$ . If available  $[j] = k$ , there are  $k$  instances of resource type  $R_j$  available
- **Max:**  $n \times m$  matrix. If  $\text{Max}[i,j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$
- **Allocation:**  $n \times m$  matrix. If  $\text{Allocation}[i,j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$
- **Need:**  $n \times m$  matrix. If  $\text{Need}[i,j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to complete its task

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

# Safety Algorithm

1. Let  $Work$  and  $Finish$  be vectors of length  $m$  and  $n$ , respectively. Initialize:

$Work = Available$

$Finish[i] = false$  for  $i = 0, 1, \dots, n-1$

2. Find an  $i$  such that both:

(a)  $Finish[i] = false$

(b)  $Need_i \leq Work$

If no such  $i$  exists, go to step 4

3.  $Work = Work + Allocation_i$ ,

$Finish[i] = true$

go to step 2

4. If  $Finish[i] == true$  for all  $i$ , then the system is in a safe state

## Resource-Request Algorithm for Process $P_i$

$\text{Request}_i$  = request vector for process  $P_i$ . If  $\text{Request}_i[j] = k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$

1. If  $\text{Request}_i \leq \text{Need}_i$ , go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If  $\text{Request}_i \leq \text{Available}$ , go to step 3. Otherwise  $P_i$  must wait, since resources are not available
3. Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:

$$\text{Available} = \text{Available} - \text{Request}_i;$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i;$$

- If safe  $\Rightarrow$  the resources are allocated to  $P_i$ ,
- If unsafe  $\Rightarrow P_i$  must wait, and the old resource-allocation state is restored

# Example of Banker's Algorithm

- 5 processes  $P_0$  through  $P_4$ ;  
3 resource types:  
 $A$  (10 instances),  $B$  (5 instances), and  $C$  (7 instances)
- Snapshot at time  $T_0$ :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
$P_0$	0 1 0	7 5 3	3 3 2
$P_1$	2 0 0	3 2 2	
$P_2$	3 0 2	9 0 2	
$P_3$	2 1 1	2 2 2	
$P_4$	0 0 2	4 3 3	

# Example

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	<u>Need</u>		
	A	B	C
$P_0$	7	4	3
$P_1$	1	2	2
$P_2$	6	0	0
$P_3$	0	1	1
$P_4$	4	3	1

- The system is in a safe state since the sequence  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  satisfies safety criteria

# Example: $P_1$ Request (1,0,2)

- Check that Request  $\leq$  Available (that is,  $(1,0,2) \leq (3,3,2) \Rightarrow$  true

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
$P_0$	0 1 0	7 4 3	2 3 0
$P_1$	3 0 2	0 2 0	
$P_2$	3 0 2	6 0 0	
$P_3$	2 1 1	0 1 1	
$P_4$	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  satisfies safety requirement
- Can request for (3,3,0) by  $P_4$  be granted?
- Can request for (0,2,0) by  $P_0$  be granted?

# Deadlock Detection

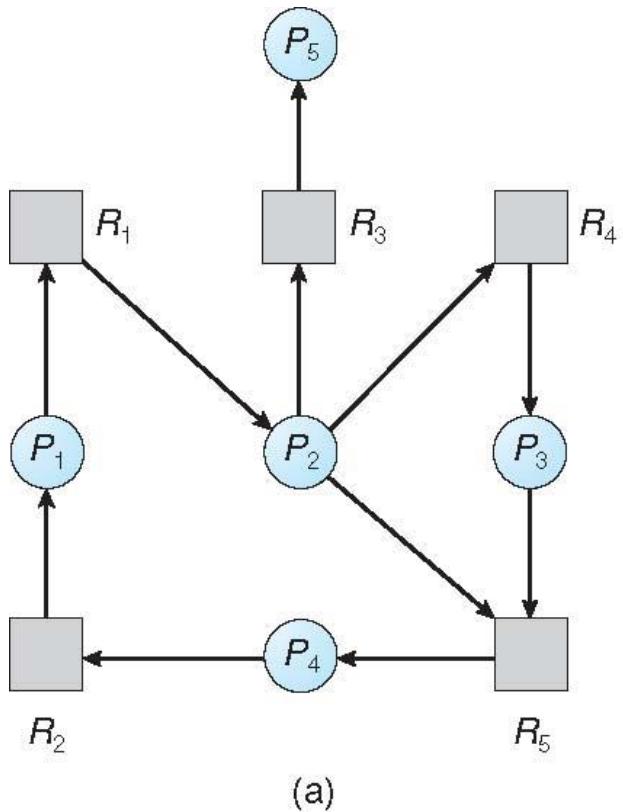
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

- It requires an algorithm which examines the state of the system to determine whether a deadlock has occurred
- It has overhead of run-time cost for maintaining necessary information and executing the detection algorithm and potential losses inherent in recovering from deadlock.

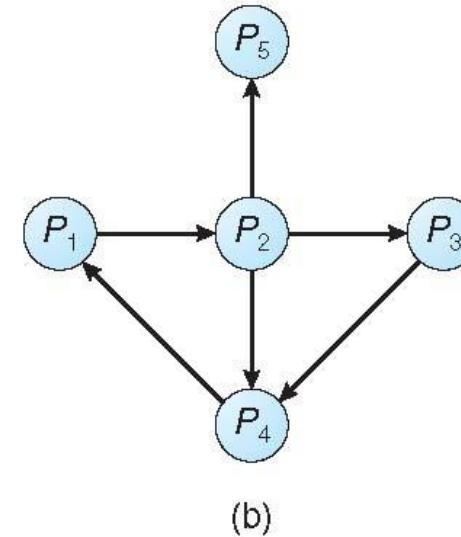
# Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- 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

# Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

# Several Instances of a Resource Type

- **Available:** A vector of length  $m$  indicates the number of available resources of each type
- **Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process
- **Request:** An  $n \times m$  matrix indicates the current request of each process. If  $\text{Request}[i][j] = k$ , then process  $P_i$  is requesting  $k$  more instances of resource type  $R_j$ .

# Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively  
Initialize:

- (a) ***Work = Available***
- (b) For  $i = 1, 2, \dots, n$ , if ***Allocation<sub>i</sub>***  $\neq 0$ , then  
***Finish[i] = false***; otherwise, ***Finish[i] = true***

2. Find an index ***i*** such that both:

- (a) ***Finish[i] == false***
- (b) ***Request<sub>i</sub> ≤ Work***

If no such ***i*** exists, go to step 4

# Detection Algorithm

3.  $Work = Work + Allocation;$

$Finish[i] = true$

go to step 2

4. If  $Finish[i] == false$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $Finish[i] == false$ , then  $P_i$  is deadlocked

Algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state

# Example of Detection Algorithm

- Five processes  $P_0$  through  $P_4$ ; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	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			

- Sequence  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  will result in  $Finish[i] = true$  for all  $i$

# Example

- $P_2$  requests an additional instance of type **C**

	<u>Request</u>		
	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

- State of system?
  - Can reclaim resources held by process  $P_0$ , but insufficient resources to fulfill other processes requests
  - Deadlock exists, consisting of processes  $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$

# Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

## Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

## Recovery from Deadlock: Resource Preemption

- Selecting a victim – criteria for selection to be made - minimize cost
- Rollback – return to some safe state, restart process for that state - some process may lose resources and cannot continue.
- Starvation – same process may always be picked as victim - include number of rollback in cost factor

- Not one solution is best for deadlock handling.
- Better solution can be provided by combining the three approaches namely prevention, avoidance and recovery, allowing the use of optimal approach for each class of resources in the system.
- Partition resources into hierarchically ordered classes and use most appropriate technique for handling deadlocks within each class.

# Memory Management

# Main Memory

- Main memory is the most critical resource as the speed of the programs depend on memory.
- Consists of large array of bytes or words each having their own address
- Limited size
- Stores programs and data required by CPU and I/O devices.

- Program must be brought into memory and placed within a process for it to be run.
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.
- User programs go through several steps before being run.

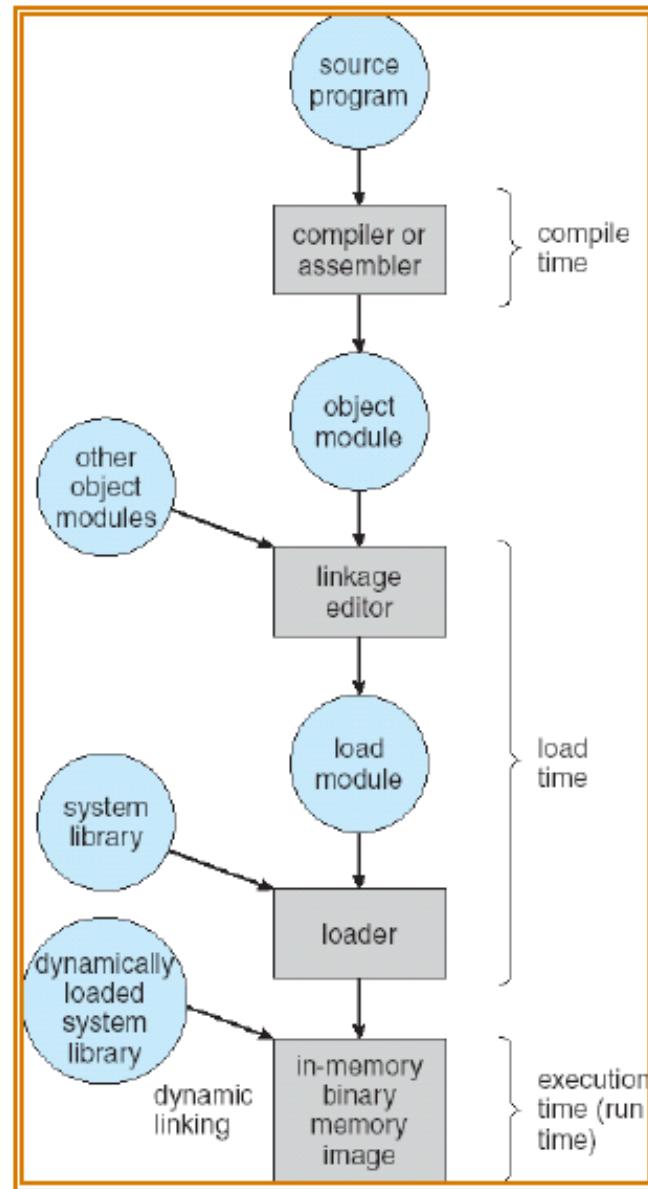
# Memory Management

- Ideally programmers want memory that is
  - large
  - fast
  - non volatile
- Memory hierarchy
  - small amount of fast, expensive memory – cache
  - some medium-speed, medium price main memory
  - gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy

## Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time:** If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
- **Load time:** Must generate *relocatable* code if memory location is not known at compile time.
- **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit registers*).



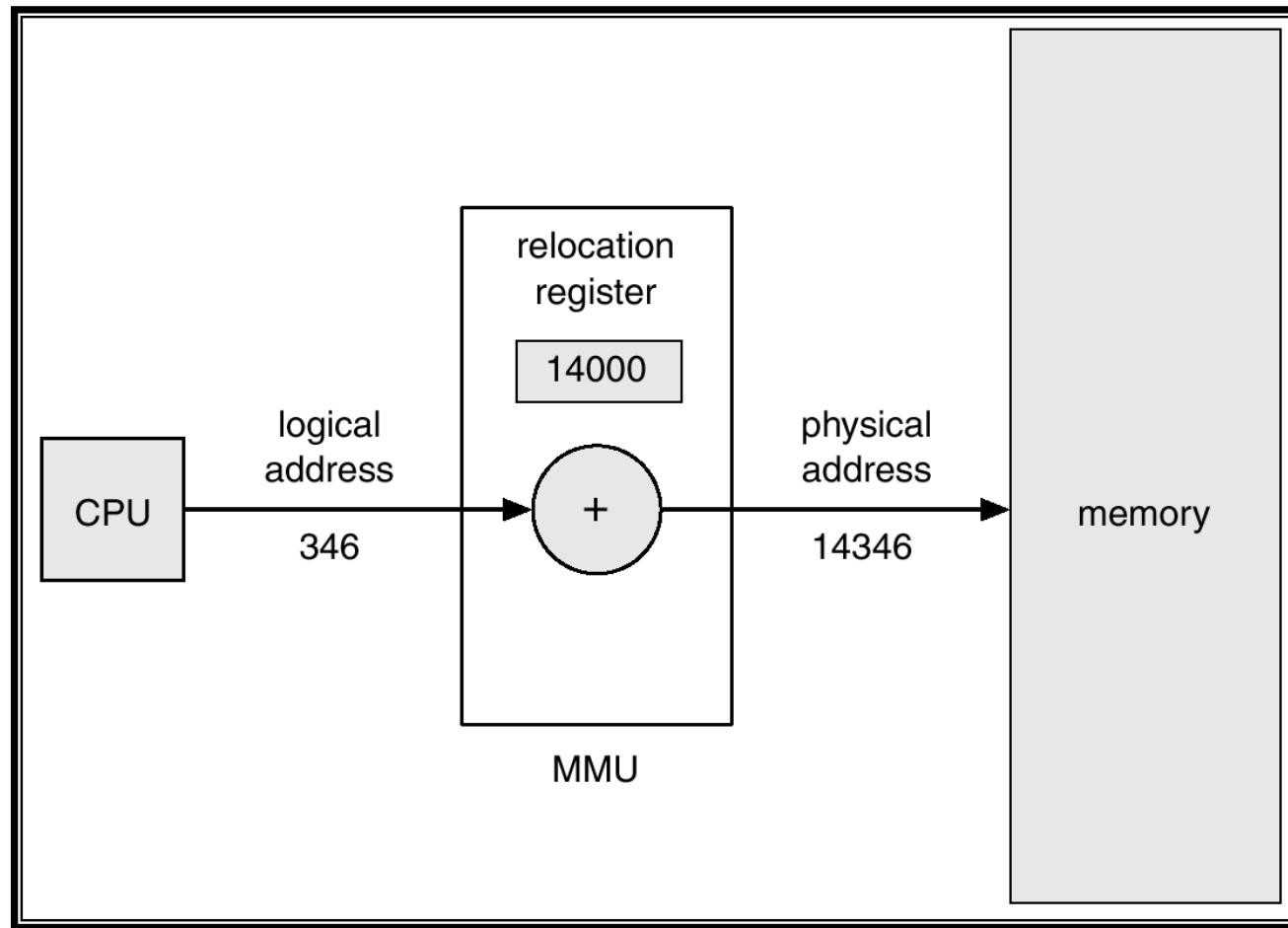
# Logical vs. Physical Address Space

- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management.
  - *Logical address* – generated by the CPU; also referred to as *virtual address*.
  - *Physical address* – address seen by the memory unit.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

# Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.

# Dynamic relocation using a relocation register



- The OS itself permanently occupies a portion of the memory for its programs and their static data.
- The remaining portion of the memory stores application programs and their static data and the dynamic data of processes and of OS.
- Memory space is recycled to store applications programs, process data and dynamic kernel programs and data. The subsystem that manages the allocation and de-allocation of memory space is called the *memory manager*.

# Memory Manager

- It is an OS component concerned with the system's memory organization scheme and memory management strategies.
- It determines how available memory space is allocated to processes and how to respond to changes in a process's memory usage.
- It also interacts with special purpose memory management hardware (if any is available) to improve performance.
- An ideal memory manager should thus minimize wasted memory and have minimal time complexity and memory access overhead, while providing good protection and flexible sharing.

Allocation of memory can be classified as contiguous allocation and noncontiguous allocation.

- Contiguous allocation scheme allocates a single block of memory for the requesting process
- Noncontiguous scheme allocates chunks or pieces of memory to a process.

# Contiguous Memory Management

## **Single Process Monitor/ Single Partition Monoprogramming :**

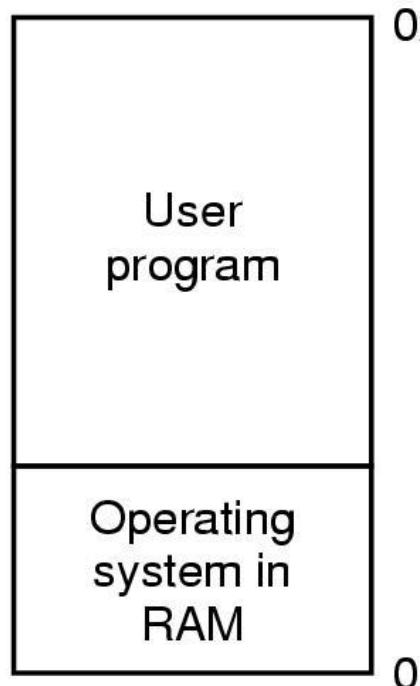
- This approach is one of the simplest ways of managing memory used in single process microcomputer.
- Memory is divided into two contiguous areas and one portion is permanently allocated to the resident portion of OS (monitor) and the remaining portion is allocated to the transient processes which are loaded and executed one at a time in response to user commands.

# Basic Memory Management

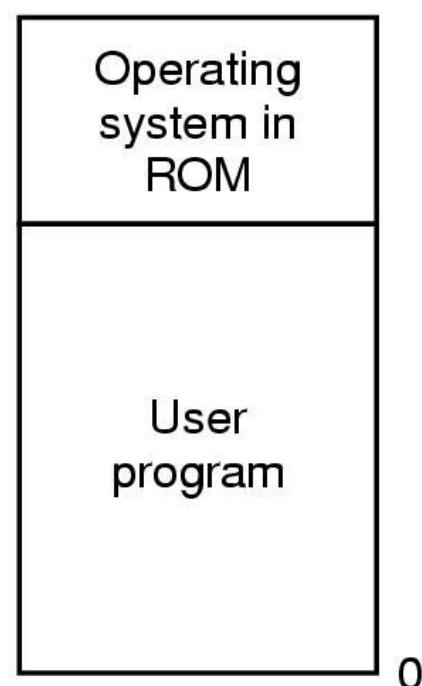
## Monoprogramming without Swapping or Paging

Three simple ways of organizing memory

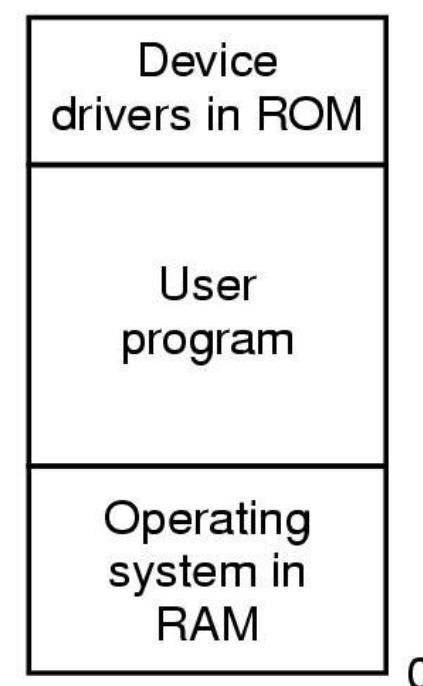
- an operating system with one user process



(a)

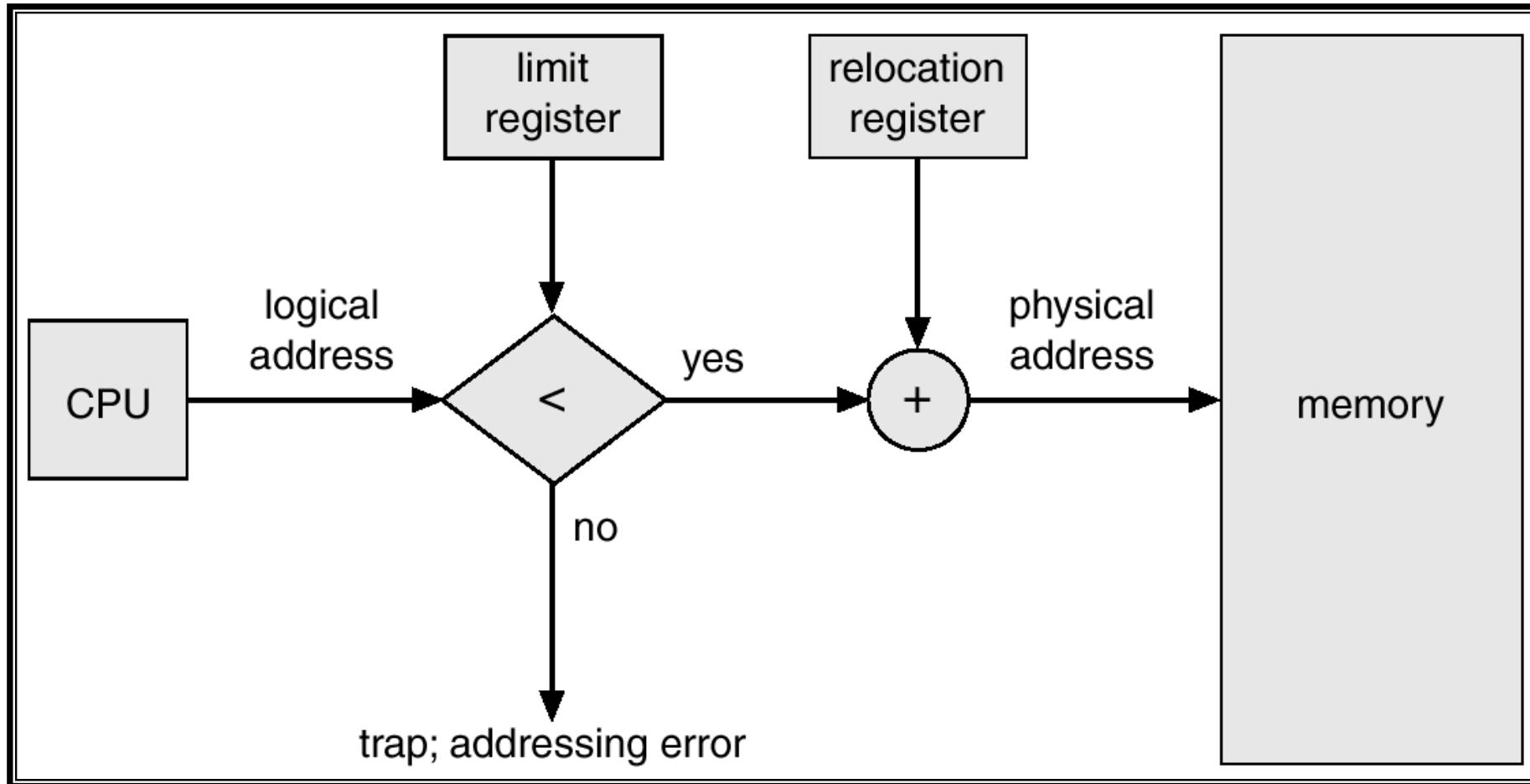


(b)



(c)

# Hardware Support for Relocation and Limit Registers



## Advantages:

- Straightforward memory allocation; Absence of multiprogramming complexities; Relatively simple to design and to comprehend; Used in systems with little hardware support for more advance forms of memory management.

## Disadvantages:

- Lack of support for multiprogramming reduces utilization of both processor and memory.

# Partitioned memory allocation

- One way to support multiprogramming is to divide the available physical memory into several partitions each of which may be allocated to a different process.
- Depending on when and how partitions are created and modified, memory partitioning may be static or dynamic.

# Static Partitioned memory allocation

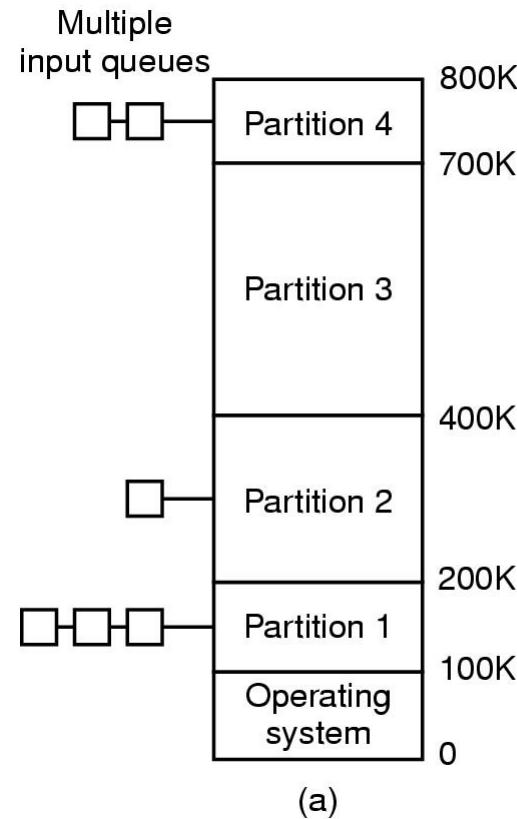
- Static partitioning implies that the division of memory is made at some time prior to the execution of user programs and that partitions remain fixed thereafter.

Allocation of partition:

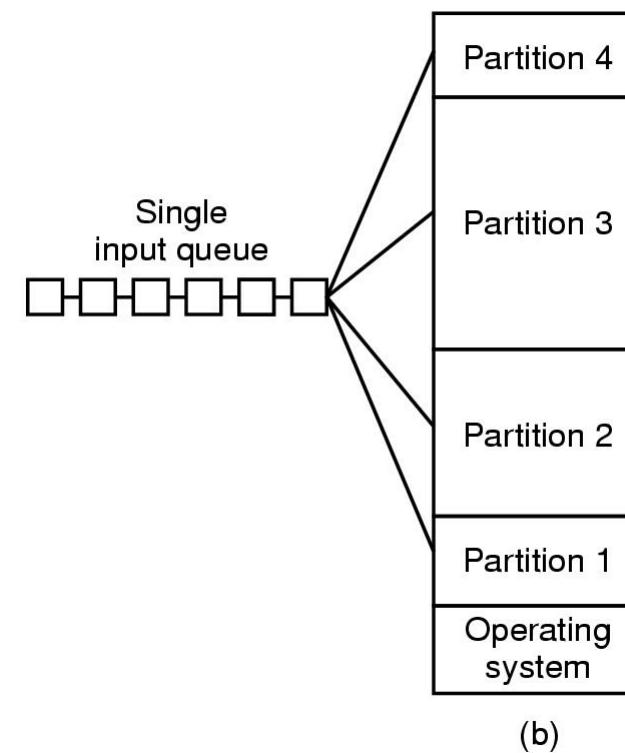
- The OS must first allocate a memory region large enough to hold the process image which is a file that contains a program in executable form and the related data and may also contain process attributes such as priority and memory requirements.

# Multiprogramming with Fixed Partitions

- Fixed memory partitions
  - separate input queues for each partition
  - single input queue



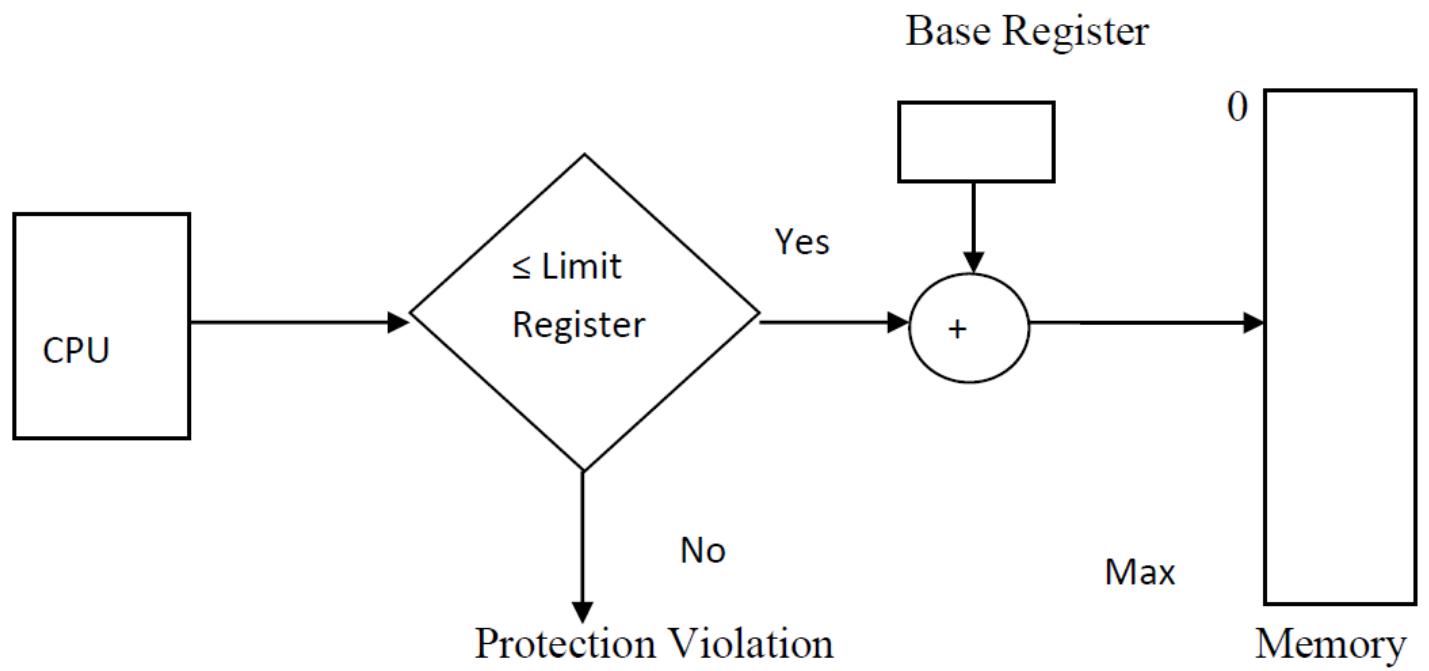
(a)



(b)

- Partition Descriptor Table (PDT) is a data structure used by OS to collect current Partition Status and attributes such as Partition Number, Partition Base and Partition Size.
- In static partitioning, partition number, base and size are fixed, only the partition status varies depending on whether the partition is allocated or not.
- When a non-resident process is to be created or activated, OS attempts to allocate a free memory partition of sufficient size by searching the PDT.

Partition Number	Partition Base	Partition Size	Partition Status
0	0K	100K	Allocated
1	100K	200K	Free
2	300K	100K	Allocated
3	400K	200K	Allocated
4	600K	100K	Free
5	700K	150K	Allocated
6	850K	150K	Free



The partition allocation strategy can be

- First Fit: Allocating the first free partition large enough to accommodate the process being created.
- Best Fit: OS allocates the smallest free partition that meets the requirements of the process under consideration.

# Problems associated with allocation

1. No partition is large enough to accommodate the incoming process.

Solution is to redefine the partitions accordingly or reduce the memory requirements by recoding or by using overlays.

2. All partitions are allocated.

Solution is by deferring the loading of the incoming process until a suitable partition can be allocated to it or forcing a memory resident process to vacate a sufficiently large partition.

Additional overhead of selecting a suitable victim and rolling it out to disk will be incurred. This operation is called swapping.

3. Some partitions are free, but none of them is large enough to accommodate the incoming process.

Solution to this problem is by both deferring and swapping.

# Swapping

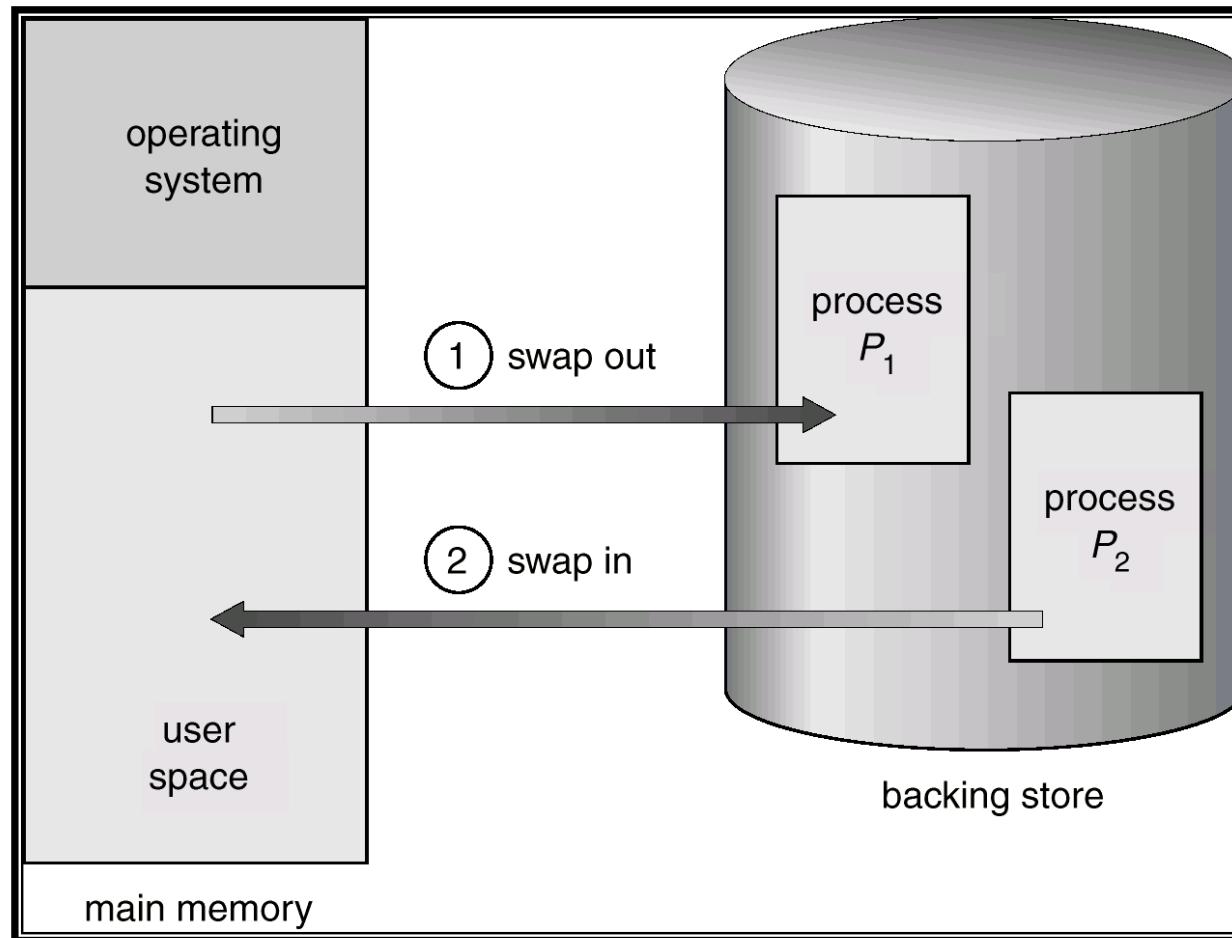
Removing suspended or pre-empted processes from memory and their subsequent bringing back is called swapping.

The major responsibilities of swapper are

- Selection of processes to swap out,
- Selection of processes to swap in and
- Allocation and management of swap space.

- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

# Schematic View of Swapping

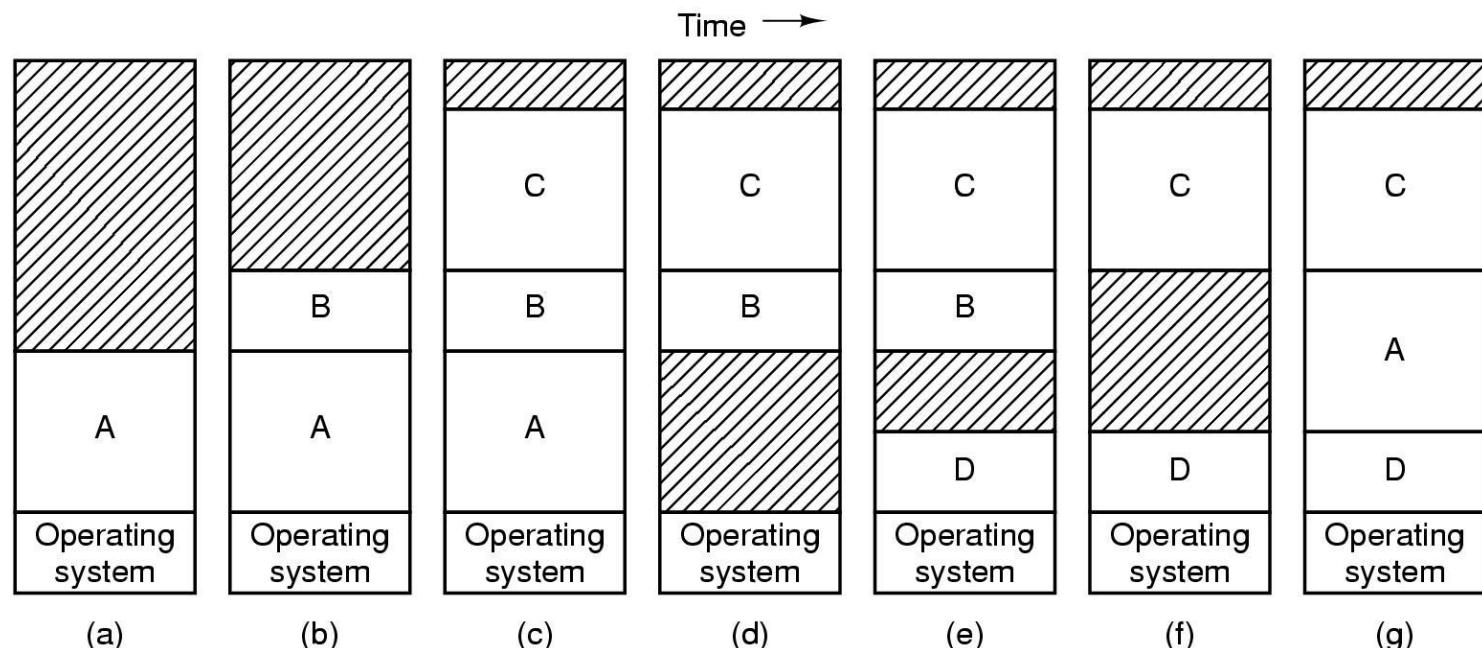


# Swapping

Memory allocation changes as

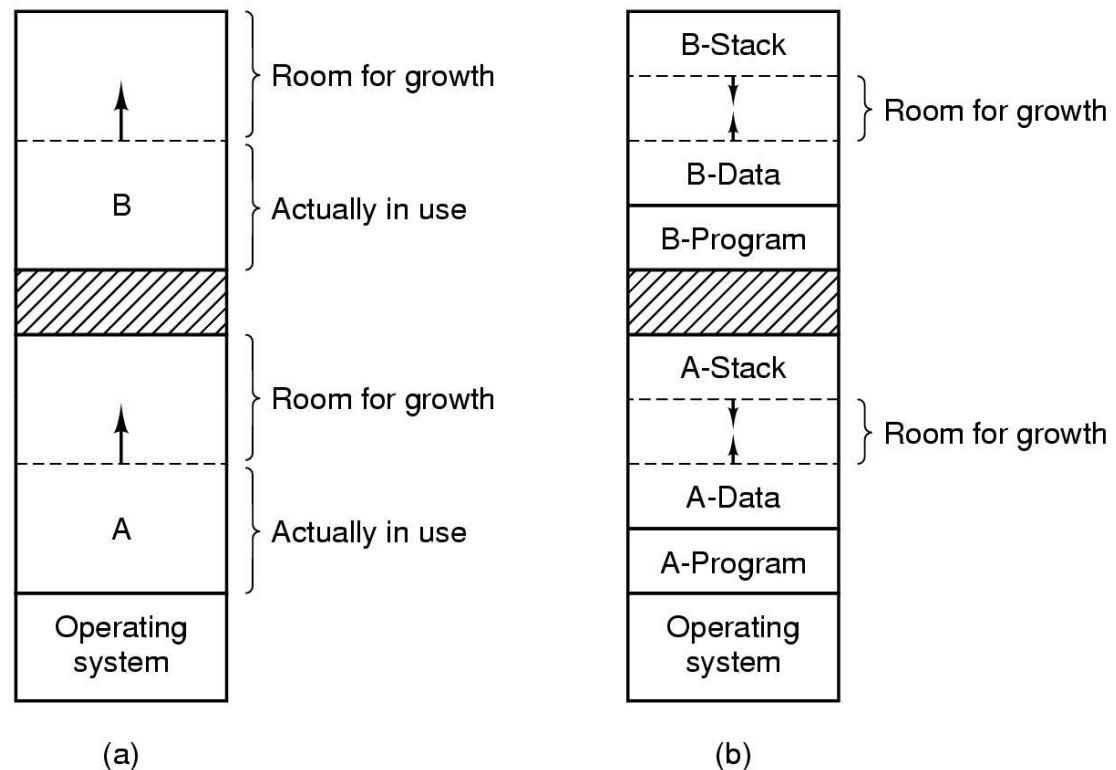
- processes come into memory
- leave memory

Shaded regions are unused memory



# Swapping

- Allocating space for growing data segment
- Allocating space for growing stack & data segment



A separate swap file must be available for storing the dynamic image of a rolled out process.

Two options for placing a swap file

i) System wide swap file: Stored in a fast secondary storage device so as to reduce the latency of swapping. The disadvantage is the size of the file. If small it may lead to run time errors and system stoppage due to inability to swap a designated process.

ii) Dedicated per process swap file. File overflow errors are avoided but it consumes more disk space.

## Advantage:

- Suitable for static environments where the workload is predictable and its characteristics are known.

## Disadvantages:

- Inflexible and inability to adapt to changing system needs.
- Internal fragmentation of memory resulting from the difference between the size of a partition and the actual requirements of the process that resides in it.
- Fixed size of the partitions does not support dynamically growing data structures such as heaps or stacks.

# Dynamic/ Variable Partitioned memory allocation

- Partition size may vary dynamically to overcome the problem of determining the number and sizes of partitions and to minimize internal fragmentation.

## Allocation of Partition:

- In this method, in addition to PDT, a free list of partitions is maintained. Initially, the whole memory is free and it is considered as one large block.
- When a new process arrives, the OS searches for a block of free memory large enough for that process. The rest of the available free space are kept for the future usage.

- If a block/partition becomes free, then the OS tries to merge it with its neighbours if they are also free otherwise append it to the free list. This appending may lead to a free list of holes of very small size.
- External fragmentation is possible i.e. a suitable single partition meeting the process requirement may not be available, but if the sizes of the holes are added, it may be greater than or equal to the requesting process's size.

The procedure for creating a partition of size P for a requesting process T is as follows:

1. Search for a partition of size  $F \geq P$ . If no match is found then error.
2. Calculate  $D = F - P$ . If  $D \leq c$  where  $c$  is a small constant value, allocate the entire free area by setting  $P = F$  and base address of P as F's base address and adjust the links in free list.

If  $D > c$ , then allocate space by setting base address of P as F's base address. Modify F's base address as P's base address + P's size and F's size as F's original size - P's size.

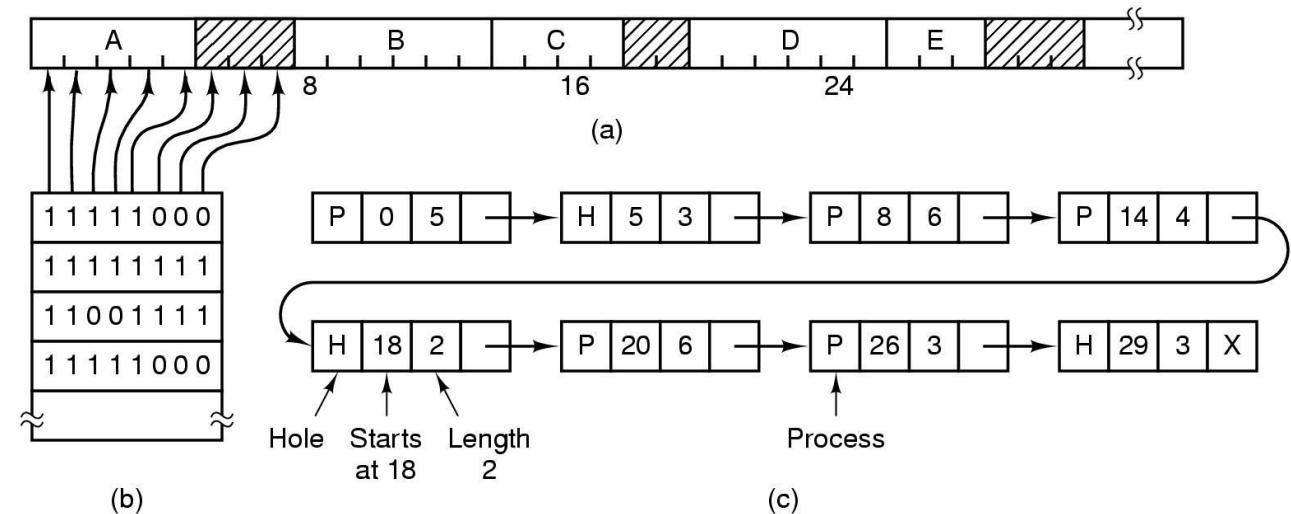
3. Find an unused entry in the PDT and record the base address, size of the partition and set the status as allocated.
4. Record the PDT's entry number in the process control block of the process T.

# Memory Allocation – Mechanism

- MM system maintains data about free and allocated memory alternatives
  - Bit maps – 1 bit per “allocation unit”
  - Linked Lists – free list updated and coalesced when not allocated to a process
- At swap-in or process create
  - Find free memory that is large enough to hold the process
  - Allocate part (or all) of memory to process and mark remainder as free
- Compaction
  - Moving things around so that holes can be consolidated
  - Expensive in OS time

# Memory Management with Bit Maps

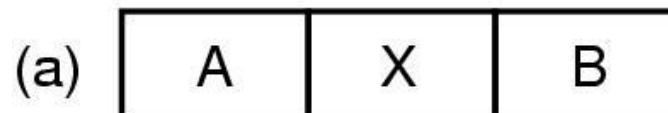
- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- Corresponding bit map
- Same information as a list



# Memory Management with Linked Lists

Four neighbor combinations for the terminating process X

Before X terminates



becomes

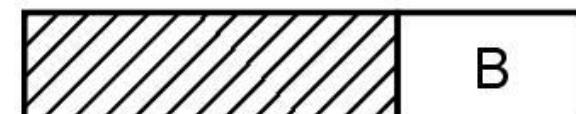
After X terminates



becomes



becomes



becomes



# Selection of partitions

- First Fit : Allocate the first free block that is large enough for the new process. This is a fast algorithm.
- Next Fit : A variant of first fit in which the pointer to the free list is saved following an allocation and used to begin the search for the subsequent allocation.
- Best Fit : Allocate the smallest block among those that are large enough for the new process.
- Worst Fit : Allocate the largest block among those that are large enough for the new process. Again a search of the entire list or sorting is needed.

# Buddy System

- An allocation-deallocation strategy, called Buddy system, facilitates merging of free space by allocating free areas with an affinity to recombine.
- The size of the partitions are powers of base 2 and the requests for free areas are rounded to the next power of base 2.
- If there is no partition of size of requesting process then a partition of next larger size is split into two buddies to satisfy the request.
- When the block is freed then the block may be recombined with its buddy if it is free.

# Compaction

- It is a method to overcome the external fragmentation problem.
- All small free blocks are brought together as one large block of free space. Compaction requires dynamic relocation.
- Selection of an optimal compaction strategy is difficult.
- One method for compaction is swapping out those processes that are to be moved within the memory, and swapping them into different memory locations.

# Protection and Sharing

- The mechanisms used are same as that of static partitioning except that dynamic partitioning allows adjacent partitions in physical memory to overlap. Sharing of code must be reentrant or executed in a mutually exclusive fashion with no preemptions.

## Advantages:

- All available memory except for the resident portion of the OS may be allocated to a single program.
- It can accommodate processes whose memory requirements are increasing during execution.

## Disadvantages:

- It needs complex bookkeeping and memory management algorithms.
- External fragmentation may lead to time penalty for compaction.

# Paging Memory management

# Noncontiguous Allocation

This means that memory is allocated in such a way that parts of a single logical object may be placed in noncontiguous areas of physical memory.

Address translation performed during the execution of instructions establishes the necessary correspondence between a contiguous virtual address space and the noncontiguous physical addresses of locations where object items reside in physical memory at run time.

# Paging

- It is a memory management scheme that removes the requirement of contiguous allocation of physical memory.
- The physical memory is conceptually divided into a number of fixed size slots called *page frames*.
- The virtual address space of a process is also split into fixed size blocks of the same size called *pages*.
- The page sizes are usually an integer power of base 2. Allocation of memory consists of finding a sufficient number of unused page frames for loading the requesting process's page.
- An address translation mechanism is used to map virtual pages to their counterparts. Since each page is mapped separately, different page frames allocated to a single process need not occupy contiguous areas of physical memory.
- Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory.

# Allocation

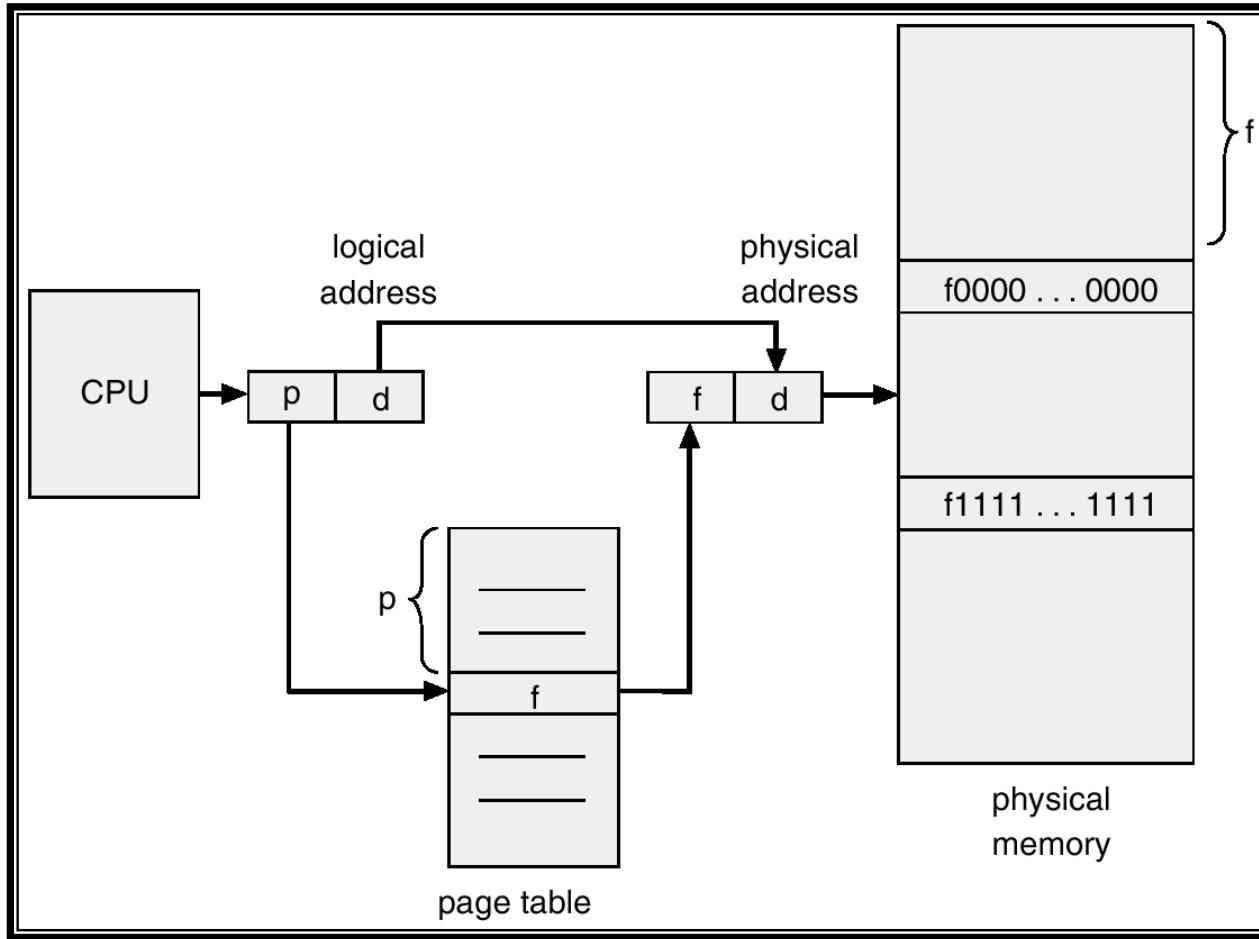
- When a process is to be executed, its corresponding pages are loaded into any available memory frames.
- The mapping of virtual address to physical addresses in paging systems is performed at the page level.
- Each virtual address is divided into two parts: Page Number (Virtual) and the offset within that page.



where p is the index of PT and d is the displacement within the page.

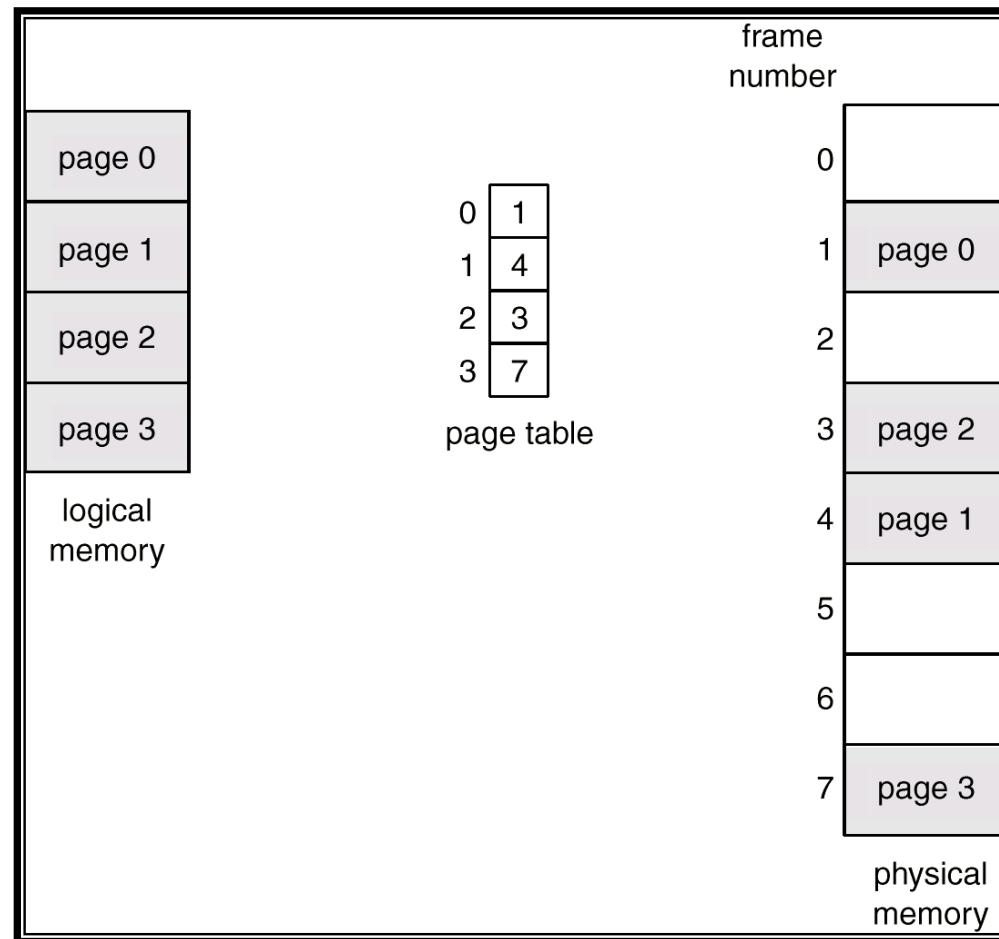
- In paging systems, address translation is performed with the aid of a mapping table called the Page Table (PT).
- There is one PT entry for each virtual page of a process and the higher order bits of the physical base address ( Page frame number) is stored in a PT entry.
- The logic of the address translation process in paged system is as follows:
  - The virtual address is split by hardware into the page number and the offset within that page.
  - The page number is used to index the PT and to obtain the corresponding physical frame number.
  - This value is then concatenated with the offset to produce the physical address, which is used to refer the target item in memory.

- OS keeps track of the status of each page frame by means of a physical memory map that may be structured as a static table referred as Memory Map Table (MMT). MMT has a fixed number of entries that is identical to the number of page frames in a given system.
- $f = m/p$  where  $f$  is the number of page frames,  $m$  is the capacity of the installed physical memory and  $p$  is the page size.
- When a process of size  $s$  requests the OS for allocation, OS must allocate  $n$  free page frames so that  $n = r(s/p) \uparrow$  where  $p$  is the page size.
- If the size of a given process is not a multiple of the page size, then the last page frame may be partly unused which is called page fragmentation or page breakage.
- All frames fit all pages and any fit is as good as any other.

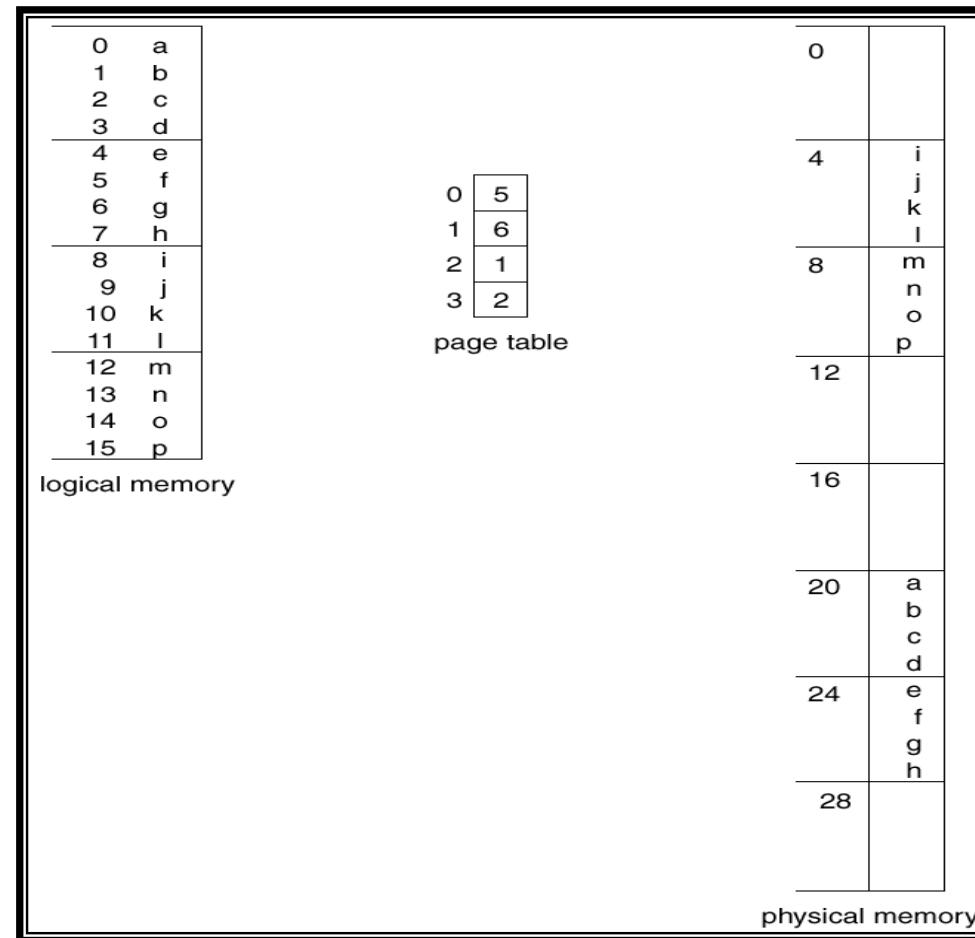


- The efficiency of the memory allocation algorithm depends on the speed with which it can locate free page frames.
- Assuming free frames are randomly distributed in memory, average number of MMT entries  $x$ , that needs to be examined in order to find  $n$  free frames may be expressed as
$$x = n/q \text{ where } q \text{ is the probability that a given frame is available for allocation.}$$
- The number of MMT entries searched  $x$  is directly proportional to  $kn$ , where  $k = 1/q$  and  $k \geq 1$ .
- If static table form is used then inverse proportion of  $x$  to  $q$  implies that the number of MMT entries that must be examined in order to locate  $x$  frames, which increases with the amount of memory in use.
- An alternative solution is to link the page frames into a free list.

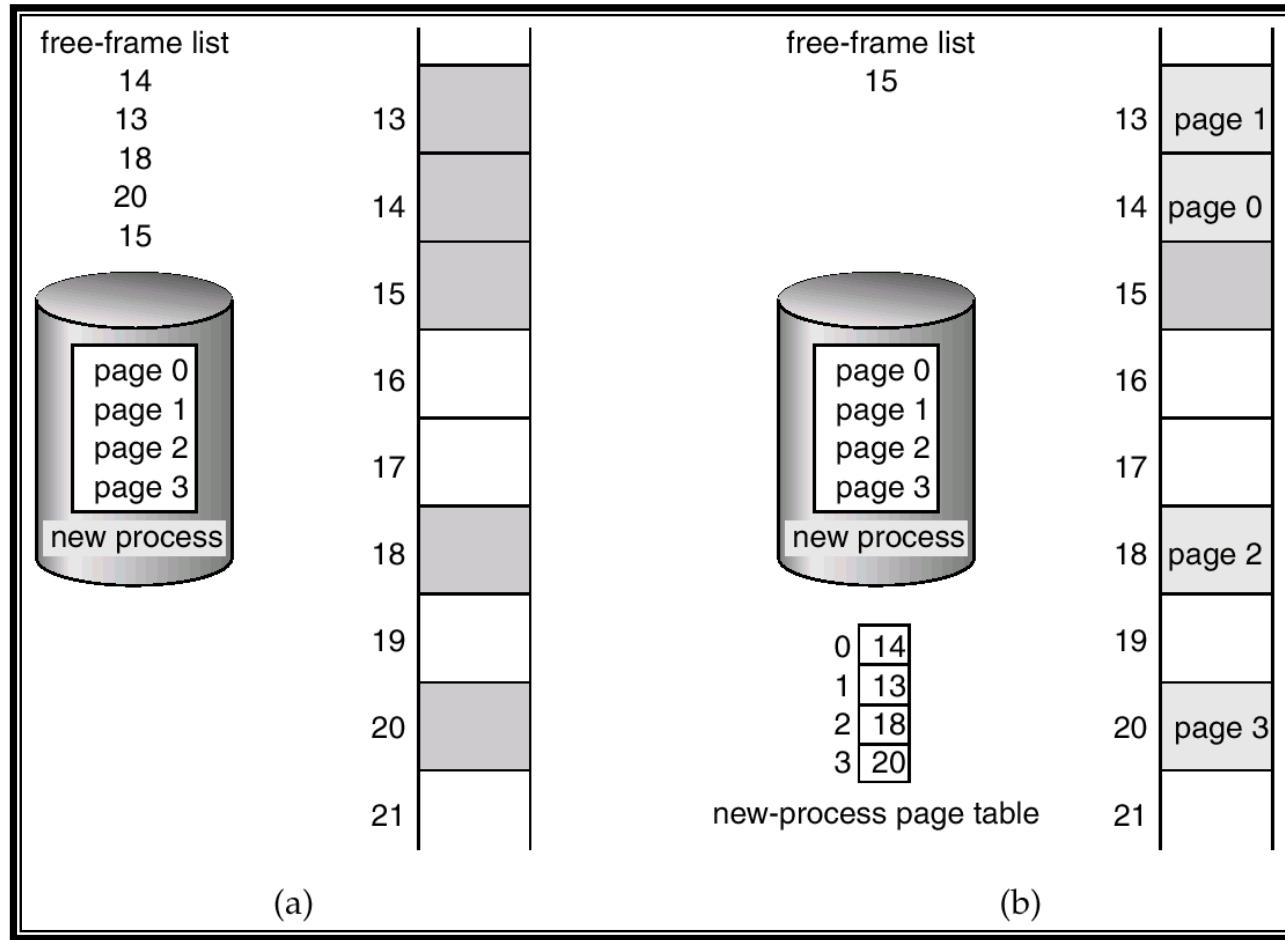
# Example



# Example



# Free Frames



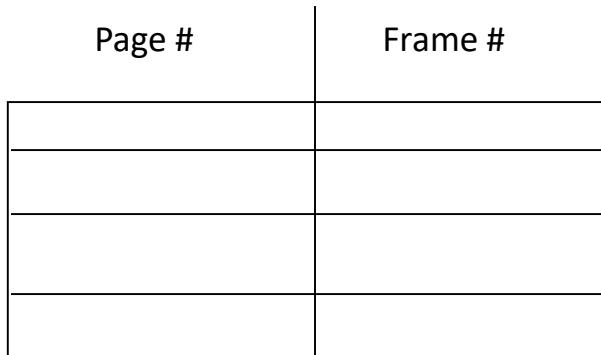
# Hardware support for paging

- It concentrates on conserving the memory necessary for storing of the mapping tables and on speeding up the mapping of virtual to physical address.
- Each PT is constructed with only as many entries as its related process has pages.
- Page Table Limit Register (PTLR) is set to the highest virtual page number defined in the PT of the running process.
- Accessing of the PT of the running process may be facilitated by means of the PT base register (PTBR) which points to the base address of the PT of the running process.

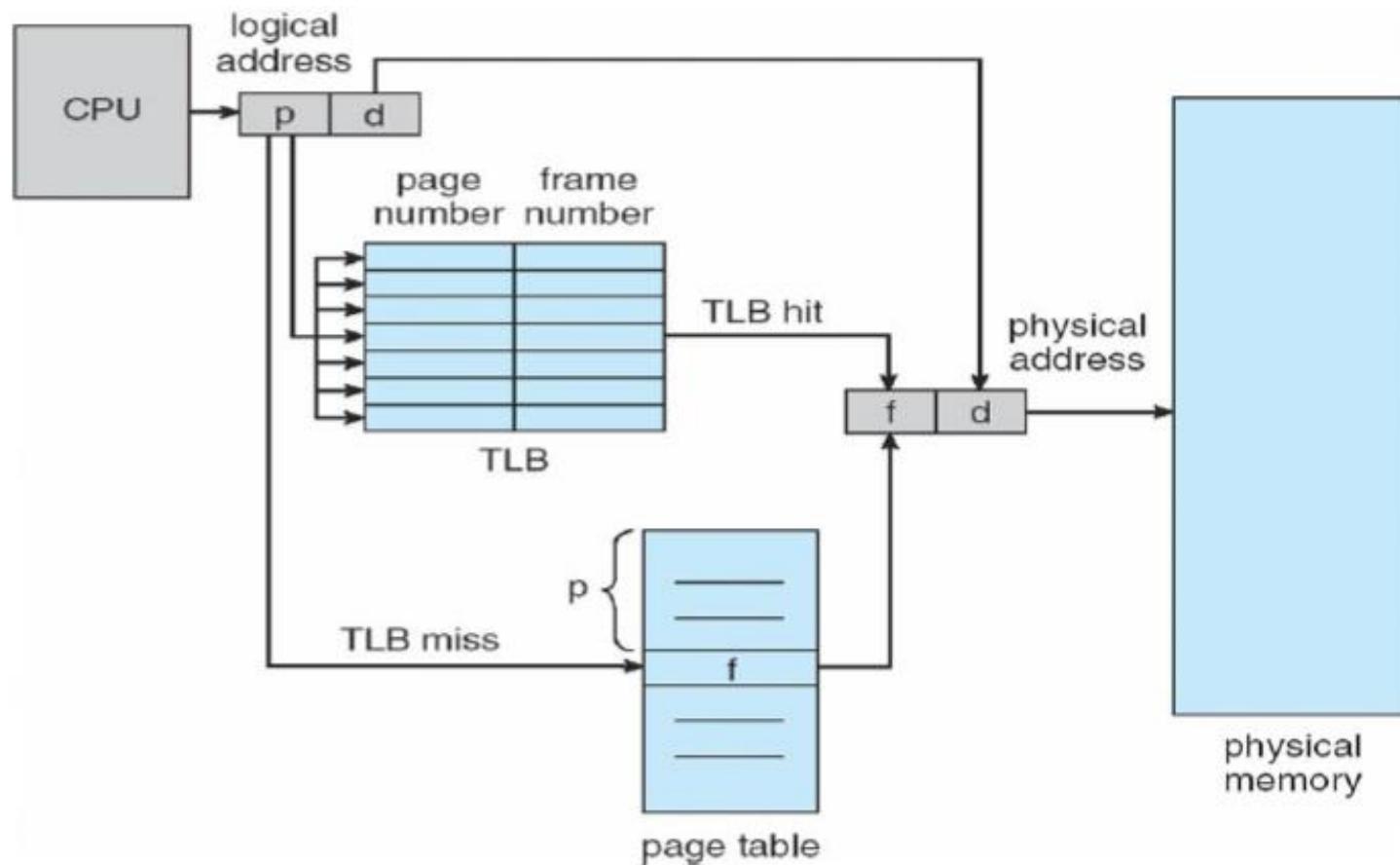
- The values loaded in the register are defined at process loading time and stored in the related PCBs.
- Address translation in paging systems also requires two memory references: One to access the PT for mapping and the other to refer the target item in physical memory.

- One popular approach is to use a high speed associative memory for storing a subset of often used PT entries called as Translation Lookaside Buffer (TLB).
- Address translation begins by presenting the page number portion of the virtual address to the TLB.
- If it is present then the page frame is combined with offset to produce the physical address. The target entry is searched in PT if it is not in TLB.

# Associative Memory



- Associative memory – parallel search
- Address translation ( $A'$ ,  $A''$ )
  - If  $A'$  is in associative register, get frame # out.
  - Otherwise get frame # from page table in memory



# Effective Access Time

- Associative Lookup =  $\varepsilon$  time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers.
- Hit ratio =  $\alpha$
- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

# Structure of PTs

## Hierarchical Paging:

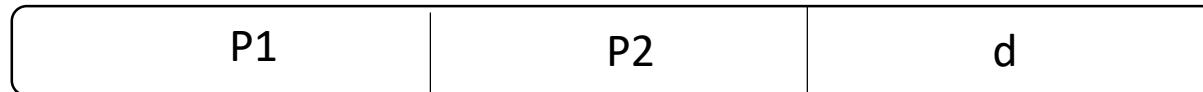
For systems with large logical/virtual address space, the size of PT needed will be in large size.

A PT of large size can be divided to smaller pieces to avoid allocating a large contiguous space.

This can be accomplished in several ways.

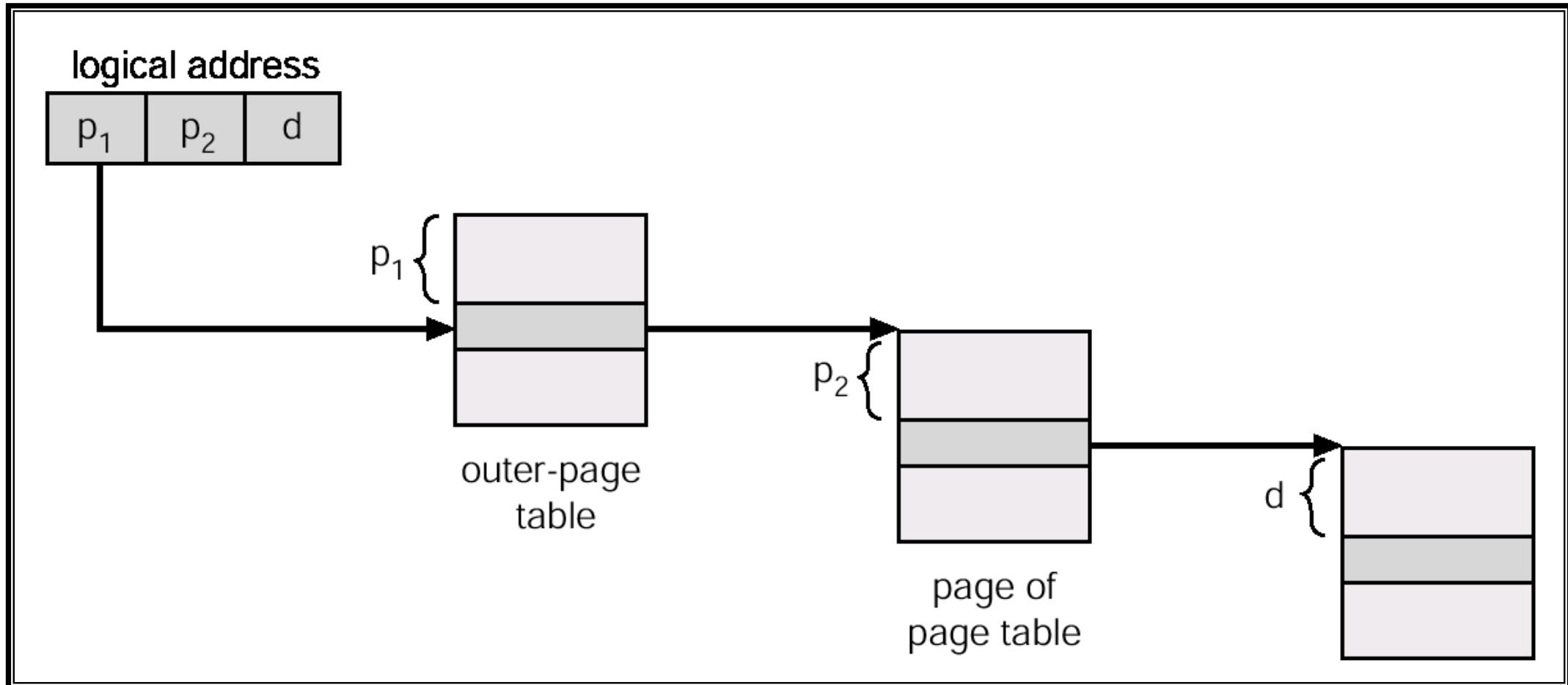
# Two level paging

PT itself is treated as pages. The page number component of virtual address is split into two portions one is the index of the outer table and second is the offset for PT.

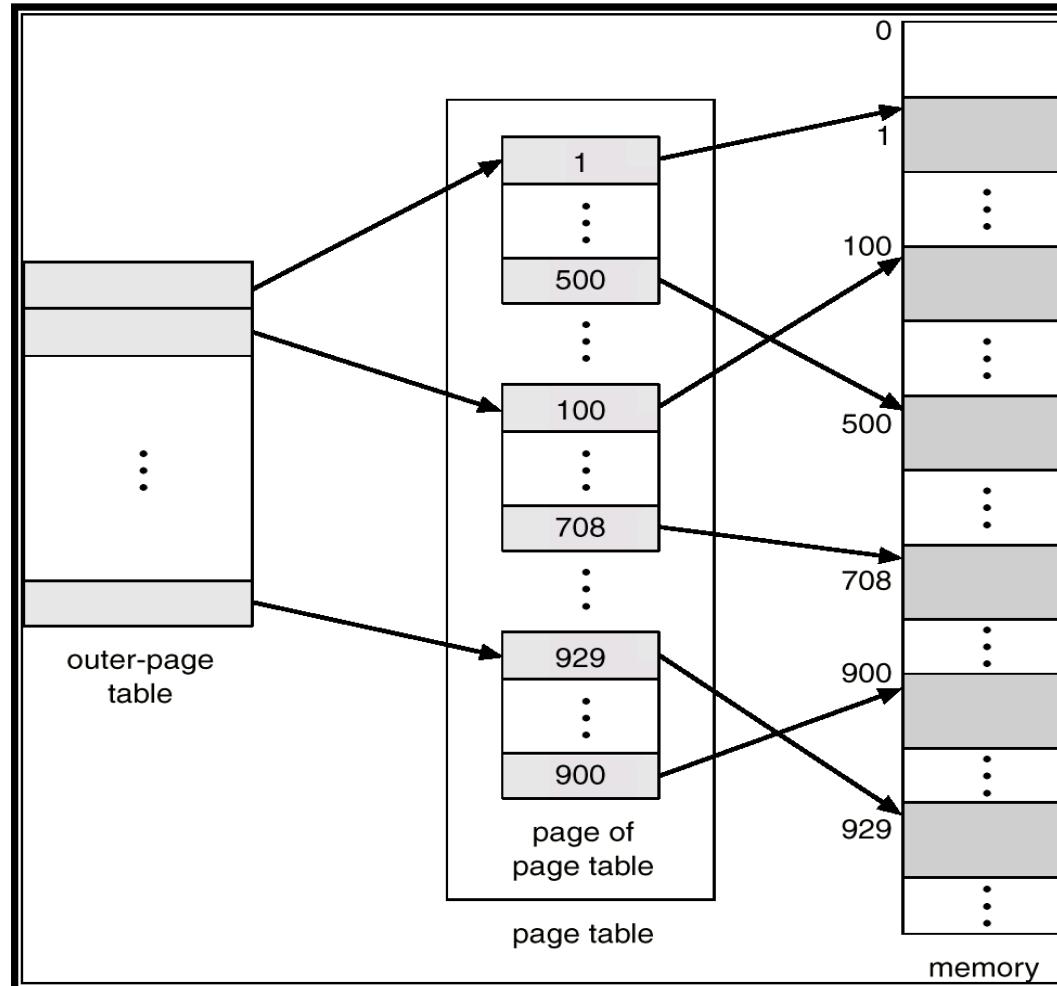


where P1 is the index of the outer PT, P2 is the displacement within the page of the outer page table and d is the displacement within the appropriate page.

# Address-Translation Scheme



# Two-Level Page-Table Scheme



## Variation of Two level Paging:

- The virtual address space of a process is divided into equal sections. The virtual address consists of section number, index to the PT and offset.

For systems with very large virtual address space three level or four level paging can be used.

# Hashed Page Tables

For handling address space larger than 32 bits, a hash page map table is used where the hash value is the virtual page number.

Each entry in the hash table contains a linked list of elements that hash to the same locations.

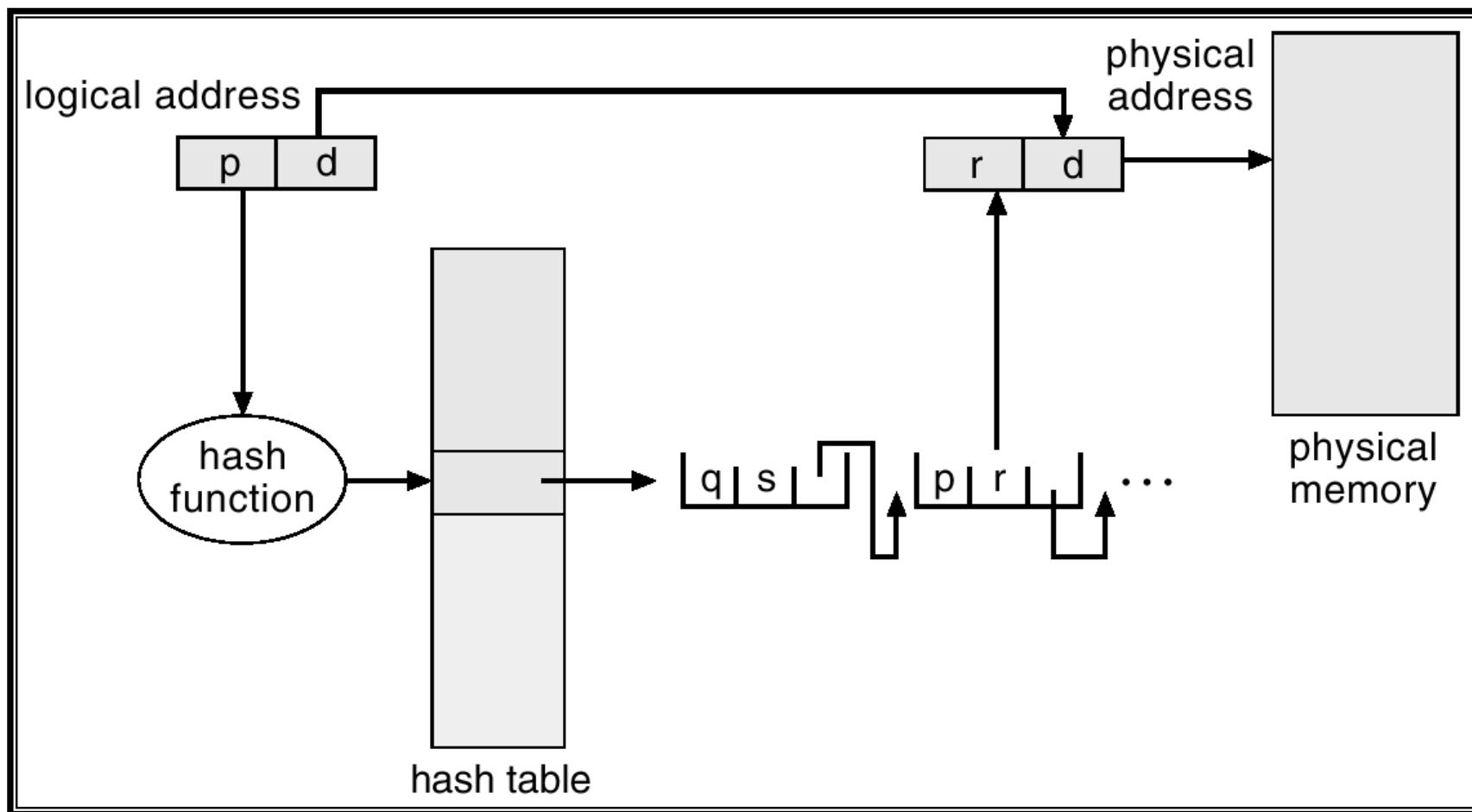
Each element consists of three fields:

- Virtual Page number

- Value of the mapped page frame

- A pointer to the next element in the linked list.

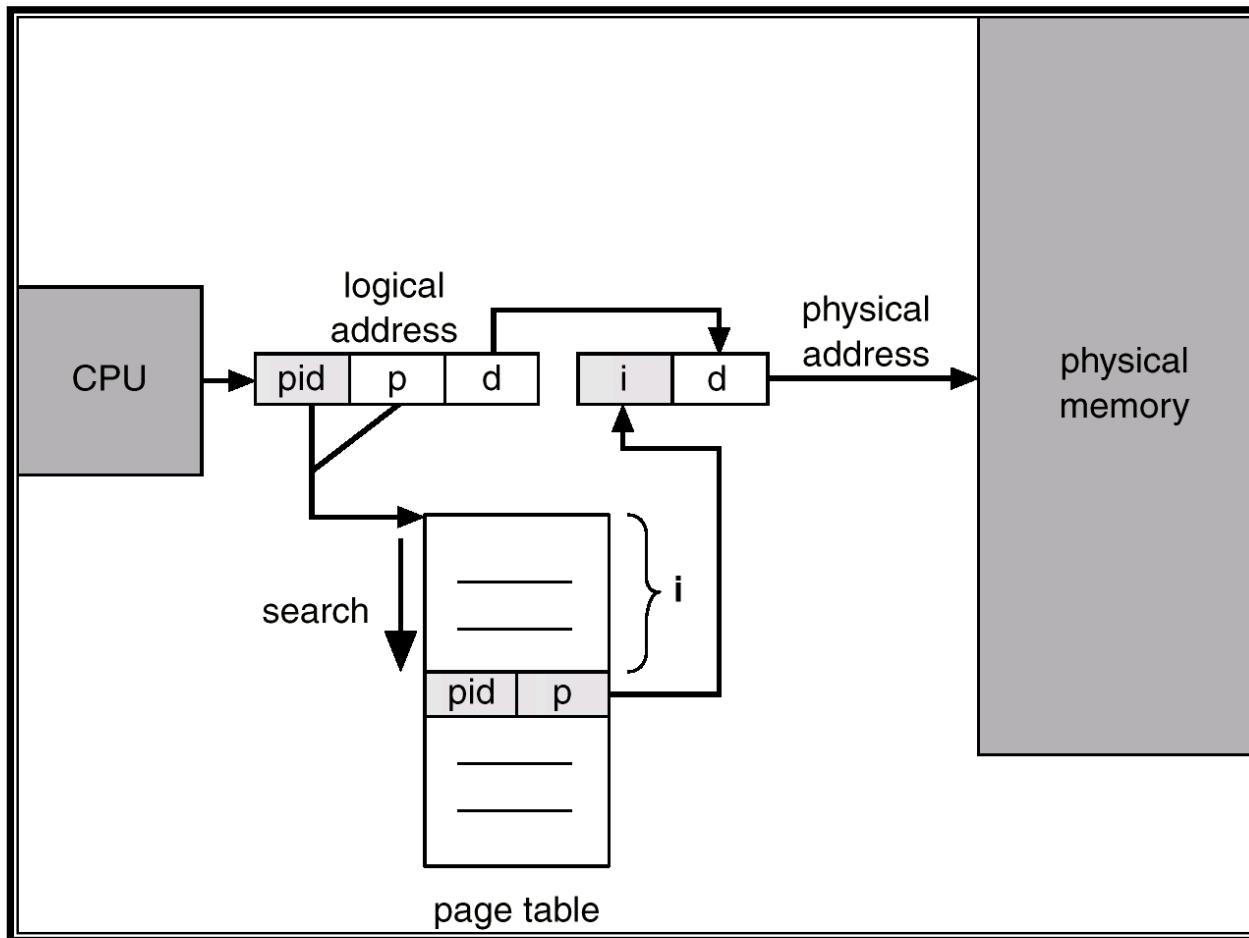
# Hashed Page Table



# Inverted Page table

- An inverted PT has one entry for each page frame of memory.
- Each entry consists of the virtual address of the page stored in that physical location with information about the process that owns the page.
- Virtual address consists of process id, page number and offset.

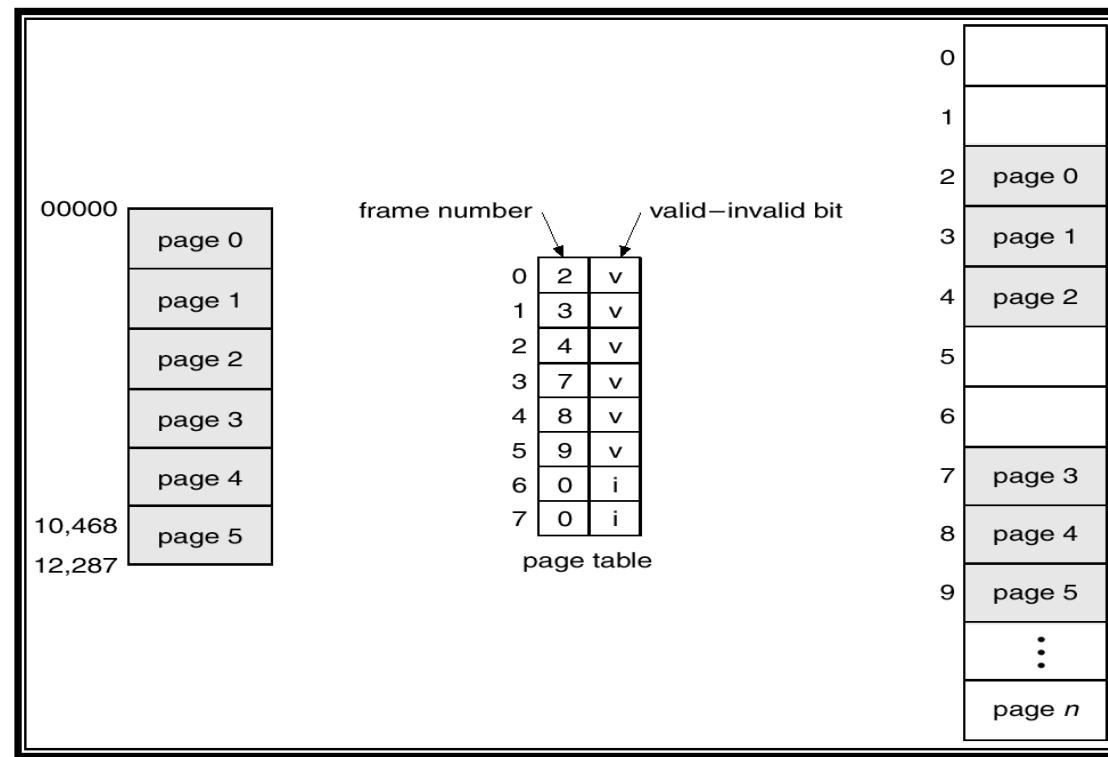
# Inverted Page Table Architecture



# Protection

- PTLR is used to detect and to abort attempts to access any memory beyond the legal boundaries of a process.
- Memory protection implemented by associating protection bit with each frame. Access bits can be added to PT entries and they can be made transparent to programmers.
- *Valid-invalid* bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
  - “invalid” indicates that the page is not in the process’ logical address space.

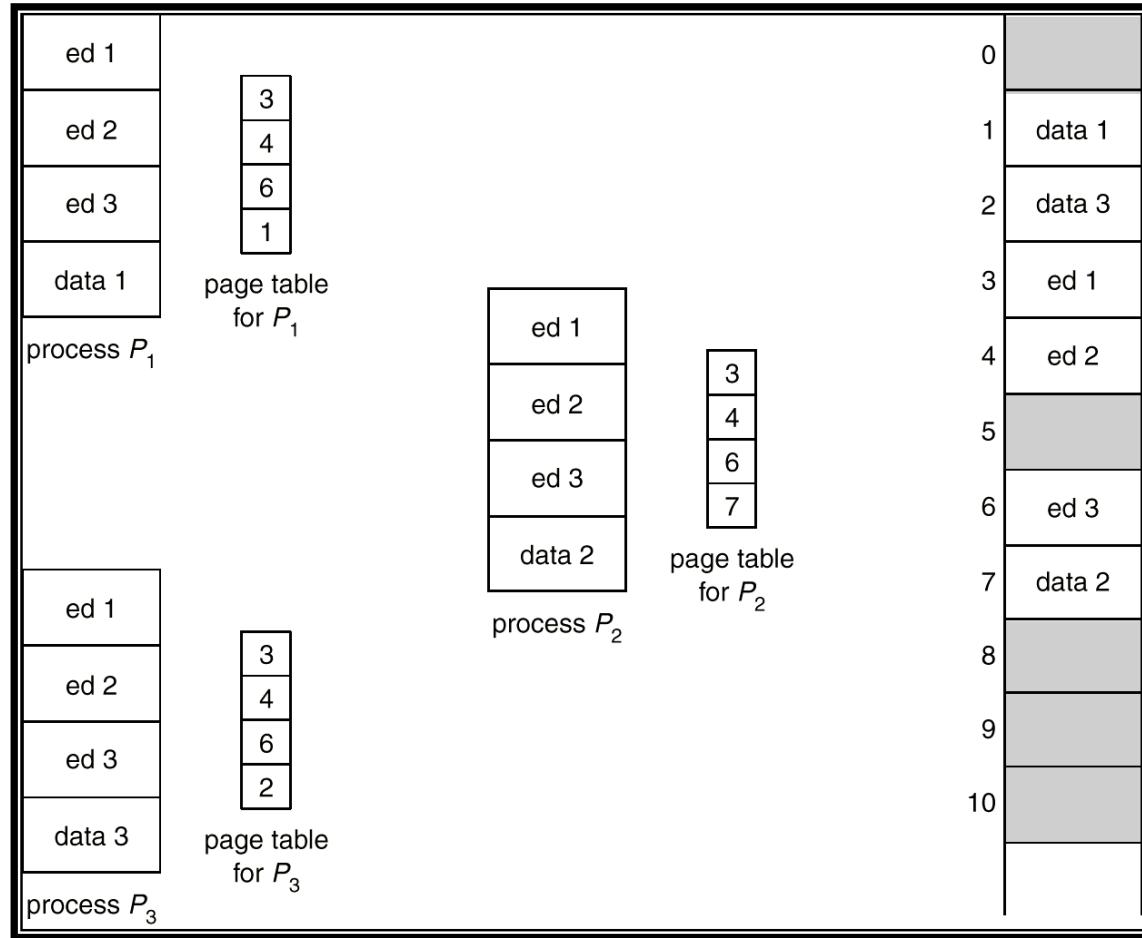
# Valid (v) or Invalid (i) Bit In A Page Table



# Sharing

- Sharing of pages is straightforward and a single physical copy of a shared page can be easily mapped into as many distinct address spaces as desired.
- Different processes may have different access rights to the shared page.

# Shared Pages Example



## Advantages:

- Managed entirely by OS and is transparent to programmers
- No compaction is needed thereby it eliminates external fragmentation
- Allocation and deallocation is simple and incurs less overhead
- Memory utilization is very high and is optimal

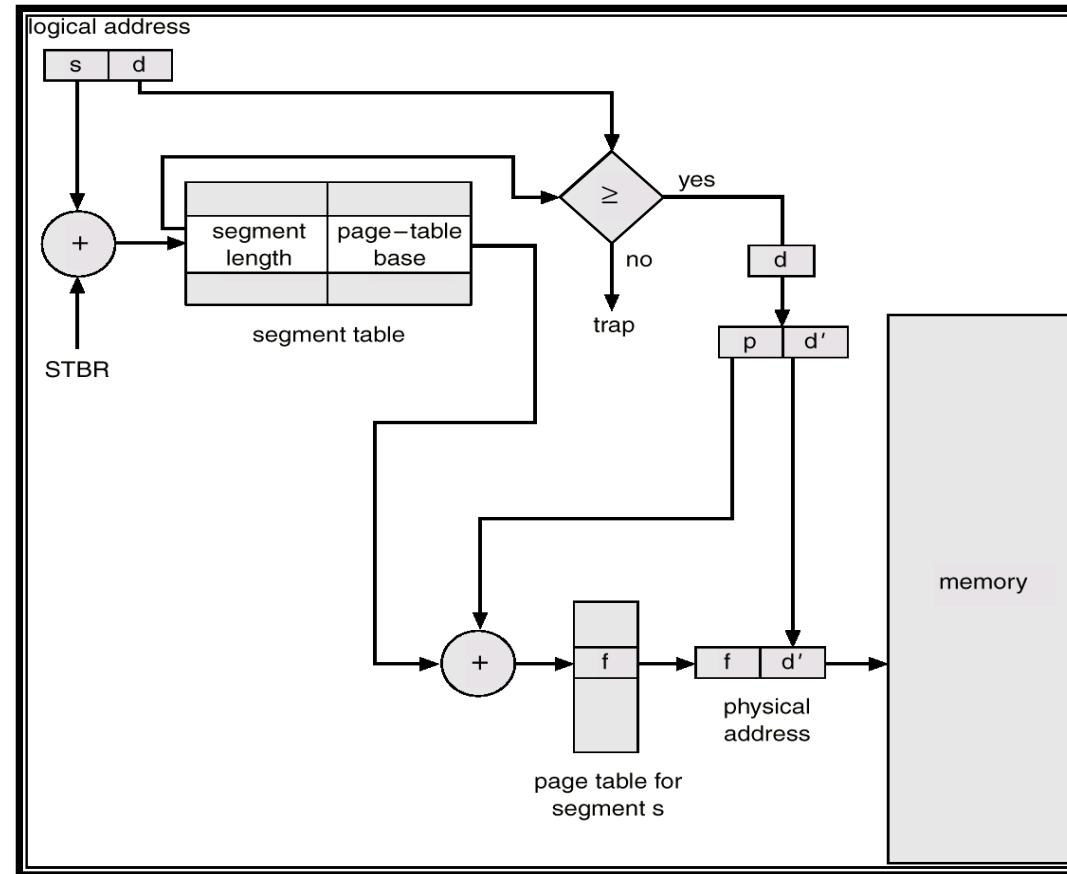
## Disadvantages:

- Storage overhead
- PT per process and MMT lead to page fragmentation
- Table fragmentation is quite large when page size is small
- TLB is expensive and reduces effective memory bandwidth
- Sharing of pages is restrictive

# Segmentation with Paging – MULTICS

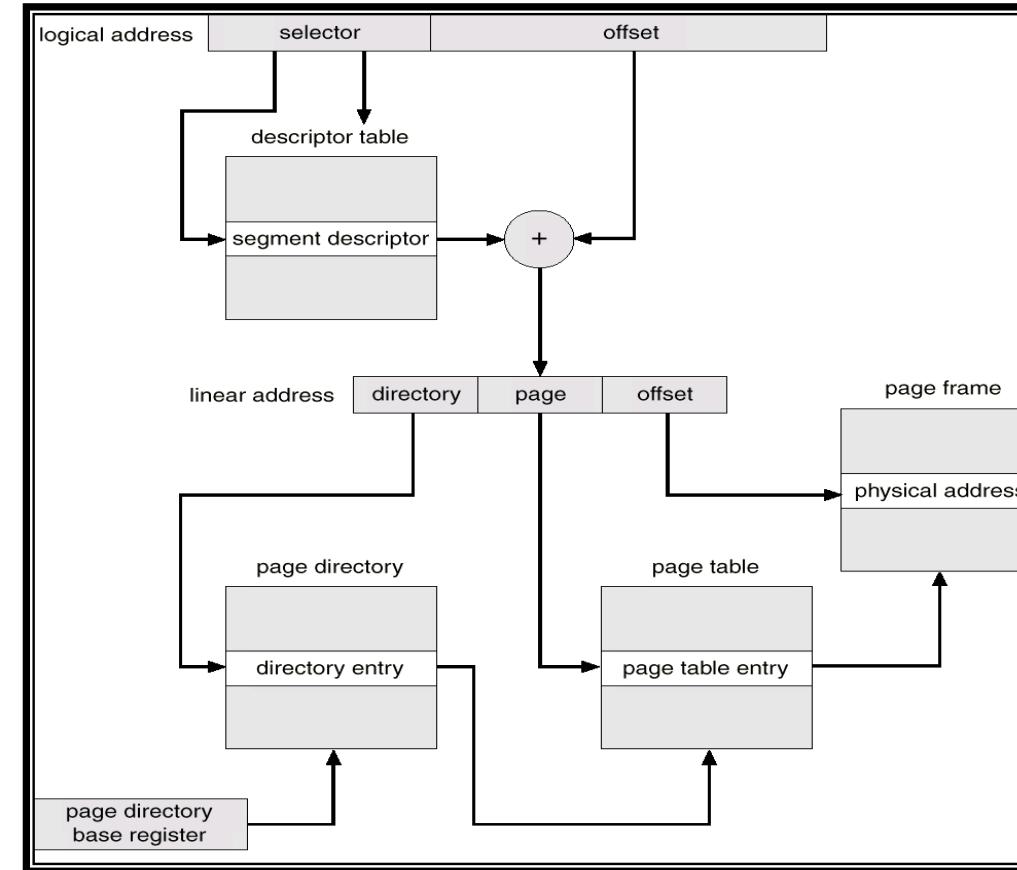
- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.

# MULTICS Address Translation Scheme



# Segmentation with Paging – Intel 386 Address Translation

segmentation with  
paging for memory  
management with a  
two-level paging  
scheme.

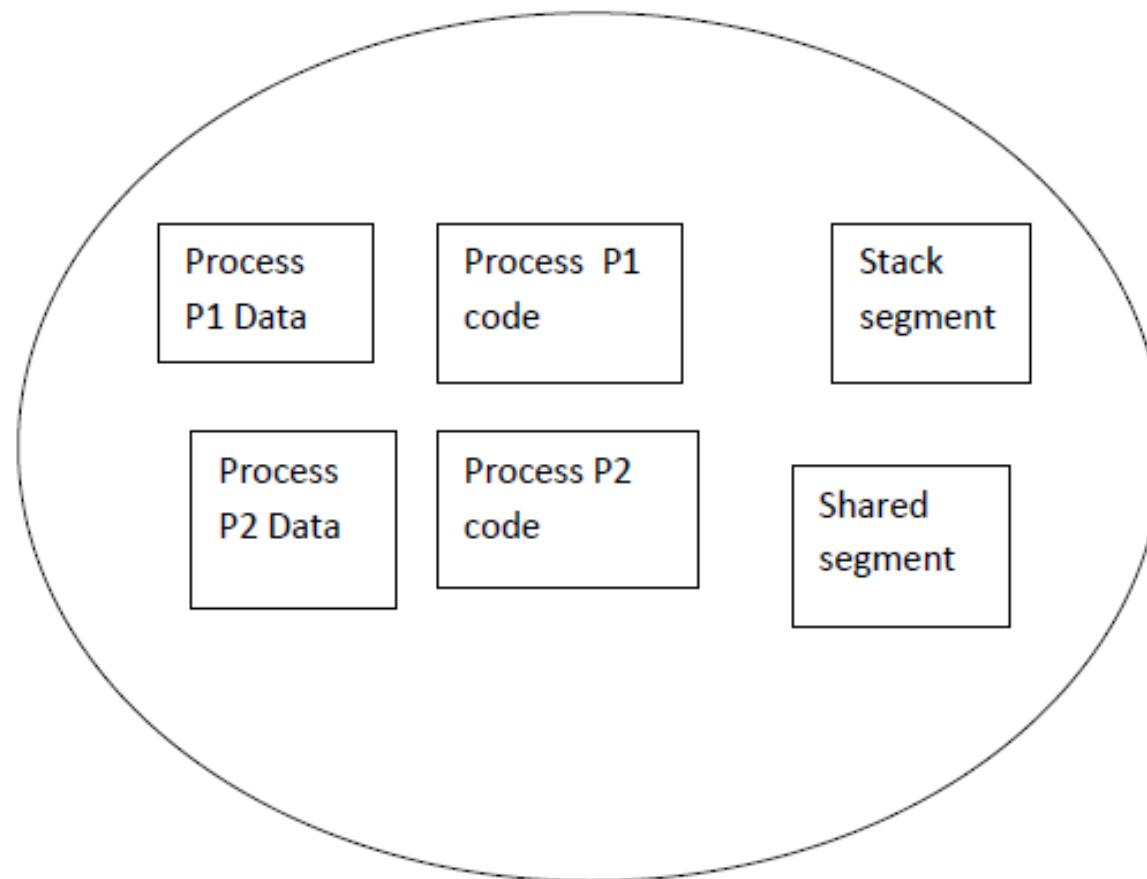


# Segmentation Memory Management

# Why Segmentation?

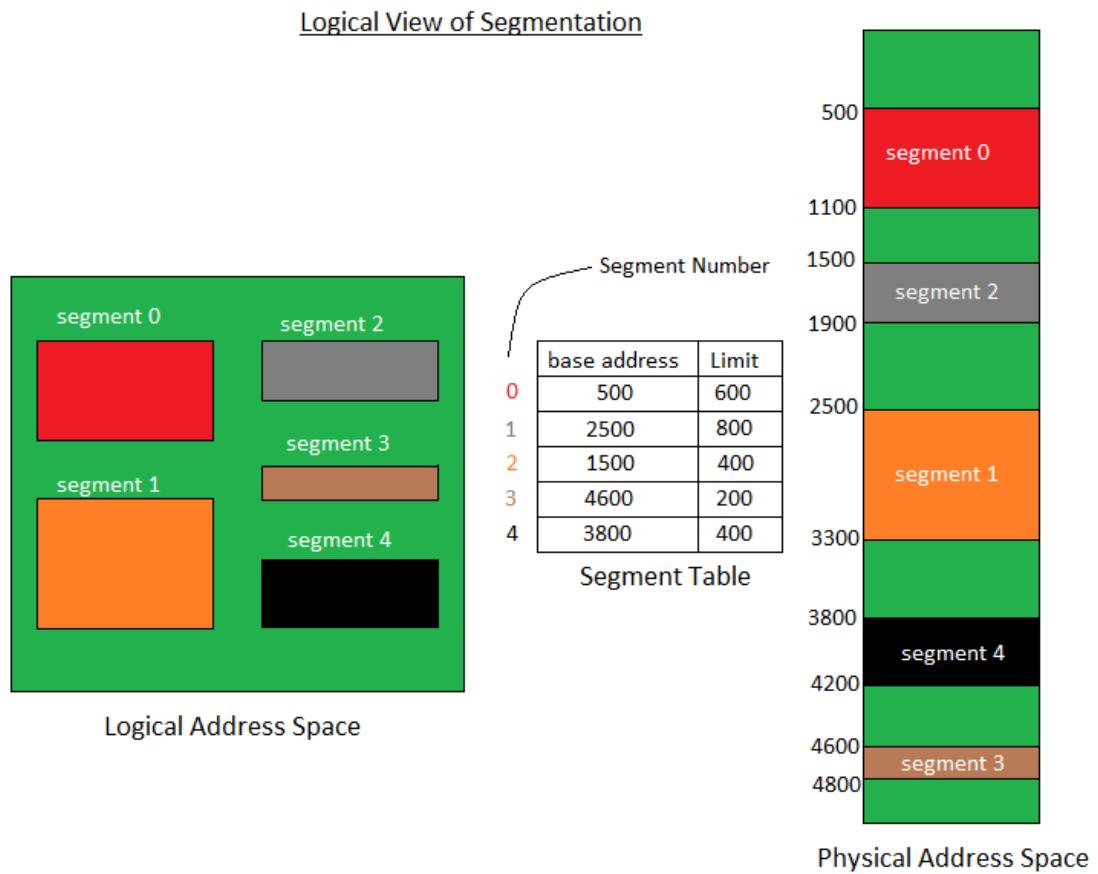
- A program can be visualized as a collection of methods, procedures or functions.
- It may also include various data structures such as objects, arrays, stacks, variables etc.
- Users view the program as a collection of modules and data.
- The extent of external and internal fragmentation and their negative impact on wasted memory should be reduced in systems where the average size of a request for allocation memory to segments is small.
- Segmentation is a way to reduce the average size of a request for memory by dividing the process's address space into blocks that may be placed into noncontiguous areas of memory.

# User's View of a Program



- It is a technique to break memory into logical pieces where each piece represents a group of related information.
- These related information are called segments and are formed at program translation.
- The segments may be data segments, code segments, shared segments and stack segments.
- These segments may be placed in separate noncontiguous areas of physical memory, but the items belonging to a single segment must be placed in contiguous areas of physical memory.
- Thus segmentation possesses some properties of both contiguous (individual segments) and noncontiguous (address space of a process) schemes for memory management.

- These segments are of varying size and thus eliminates internal fragmentation.
- The elements within a segment are identified by their offset from the beginning of the segment. External fragmentation still exists but to lesser extent.

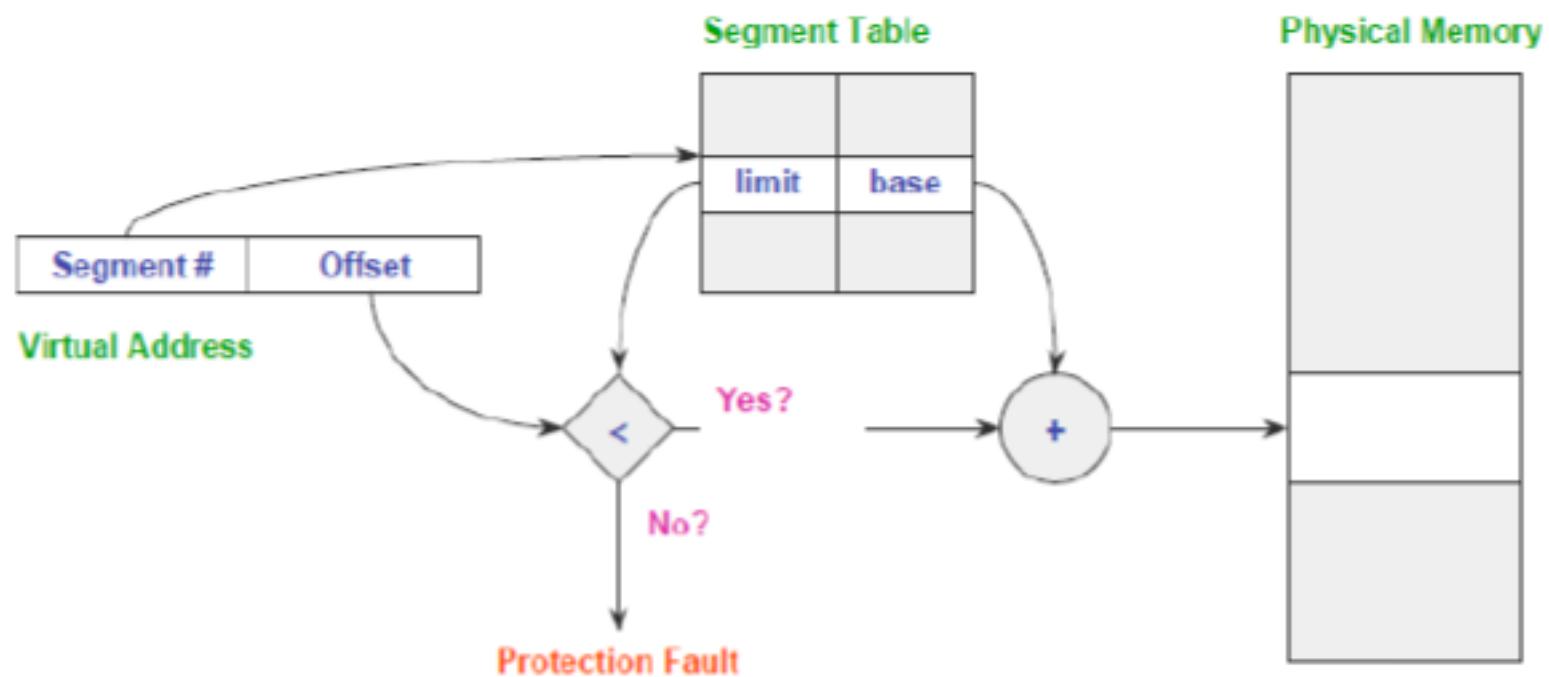


- Relocation.
  - dynamic
  - by segment table
- Allocation.
  - first fit/best fit
  - external fragmentation

- For relocation purposes, each segment is compiled to begin at its own virtual address 0.
- An individual item within a segment is then identifiable by its offset relative to the beginning of the enclosing segment.
- The unique designation of an item in a segmented address space requires the specification of both its segment and the relative offset.
- Address in segmented systems have two components:
  - Virtual Address : Segment name (number) and Offset within the segment.
  - Segment number - used to index the SDT and to obtain the physical base address.
  - Offset- used to produce physical address by adding with the base value

- When a segmented process is to be loaded onto the memory, OS attempts to allocate memory for all the segments of the process.
- It may create separate partitions to suit the needs of each segment and allocation of partition to segments is similar to dynamic partitioning.
- In segmented systems, an address translation scheme is used to convert a two dimensional virtual segment address into its single dimensional physical equivalent.
- The base (obtained during partition creation) and size ( specified in the load module) of loaded segment are recorded as a tuple called the segment descriptor.

- All segment descriptors of a given process are collected in a table called the Segment Descriptor Table (SDT).
- The size of a SDT is related to the size of the virtual address space of a process.
- SDT is treated as a special type of segment. Two registers are used to access the SDT.
  - Segment Descriptor table base register (SDTBR) - points to the base of the running process's SDT
  - Segment Descriptor table limit register (SDTLR)- provided to mark the end of the SDT pointed to by the SDTBR.



- Segmentation is multiple base limit version of dynamically partition memory. The price paid for segmenting the address space of a process is the overhead of storing and accessing SDTs.
- Mapping each virtual address requires two physical memory references for a single virtual (Program ) reference.
  - memory reference to access the segment description in the SDT.
  - memory reference to access the target item in physical memory.

## Segment Descriptor Caching:

- Hardware accelerators are used to speed the translation. Most frequently used segment descriptors are stored in registers.

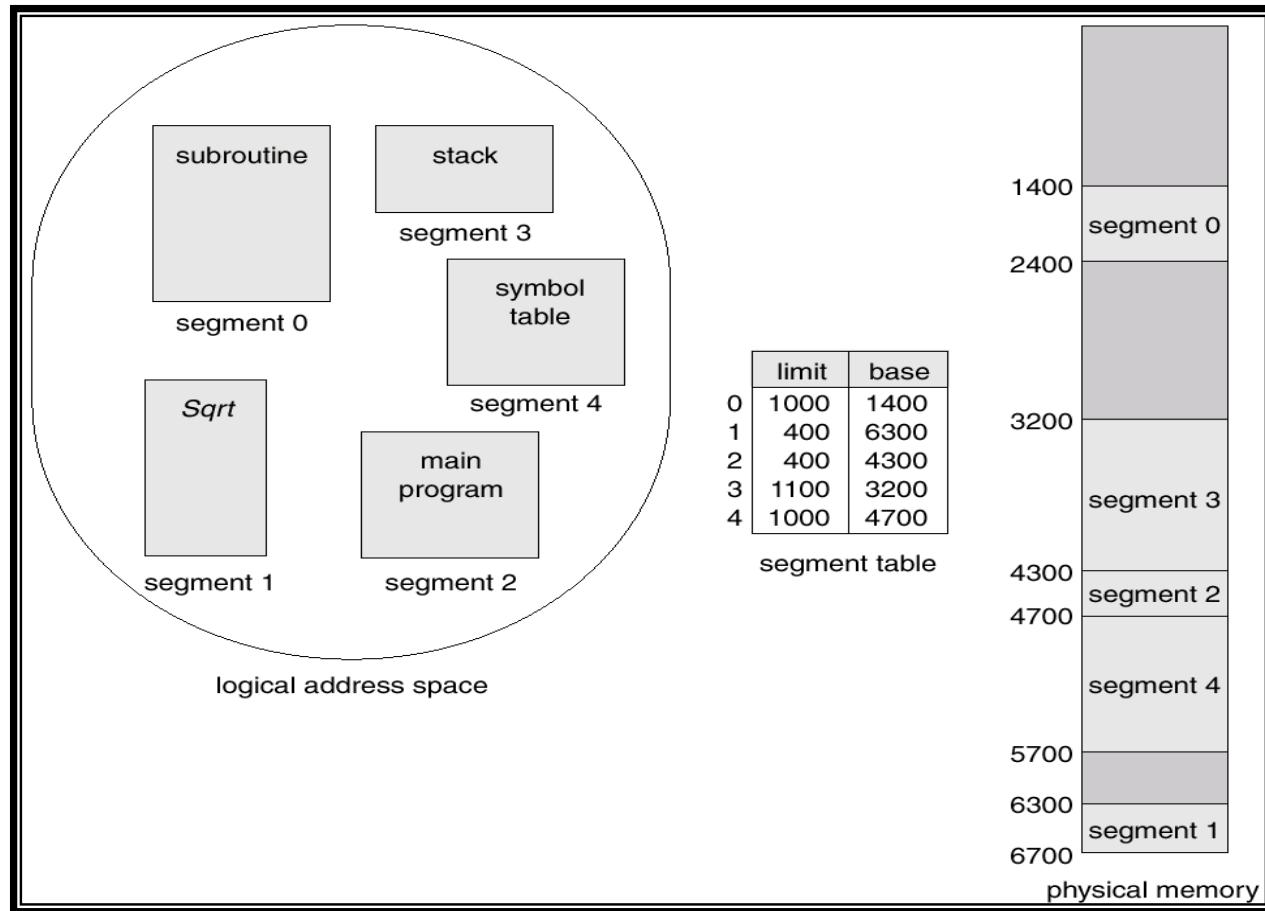
- Protection

The base limit form of protection is used. To provide protection within the address space of a single process, access rights such as read only, write only, execute only can be used depending on the information stored in the segments.

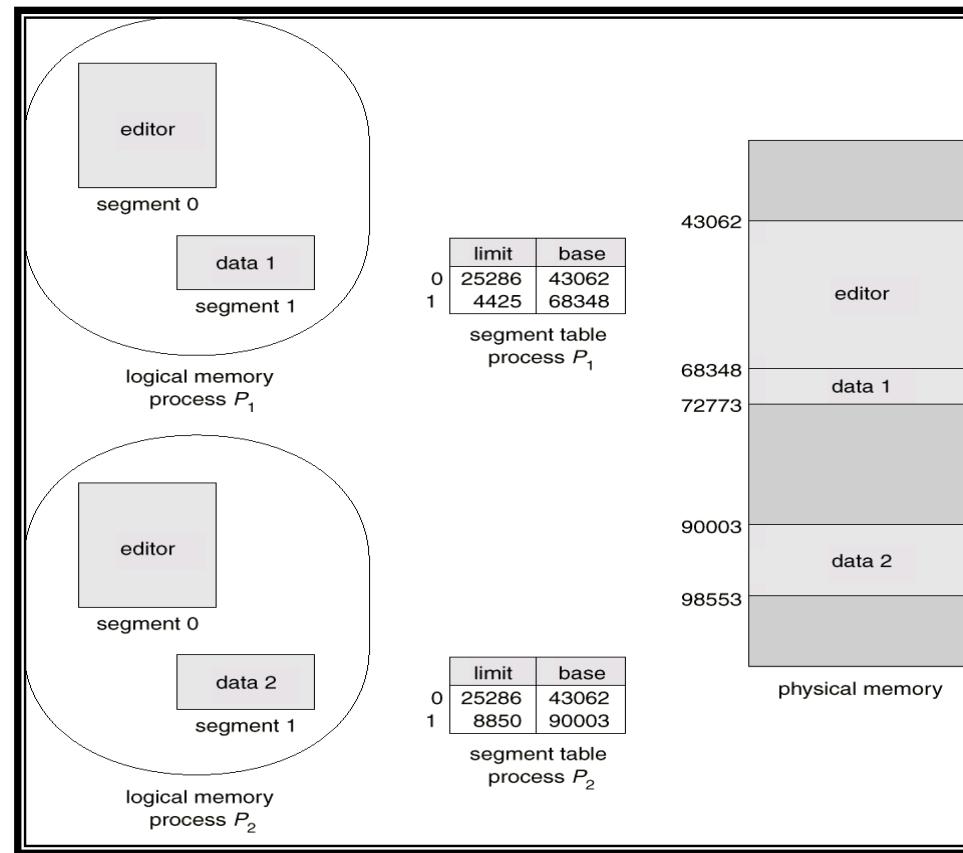
- Sharing

A shared segment may be mapped via the appropriate SDTs to the virtual address spaces of all processes that are authorized to reference it.

# Example of Segmentation



# Sharing of Segments



- Advantages:
  - Elimination of internal fragmentation
  - Support for dynamic growth of segments
  - Protection and sharing of segments
  - Modular program development
  - Dynamic linking and loading
- Disadvantages:
  - More complex strategies for compaction need to be employed.
  - The two step translation of virtual addresses to physical addresses has to be supported by dedicate hardware to avoid a drastic reduction in the effective memory bandwidth.
  - No single segment may be larger than the available physical memory.
  - Large data structures are split into several segments which results in run time overhead.

# Virtual Memory

- It is a memory management scheme where only a portion of the virtual address space of a resident process may actually be loaded into physical memory.
- The sum of the virtual address spaces of active processes in a virtual memory system can exceed the capacity of the available physical memory provided that the physical memory is large enough to hold a minimum amount of the address space of each active process.

- Virtual memory can be accomplished by maintaining an image of the entire virtual address space of a process on secondary storage and by bringing its sections into main memory when needed.
- The choice of which sections to bring in, when to bring in and where to place them is made by OS. The allocation of physical memory is on demand basis.

- Demand paging
  - Do not require all pages of a process in memory
  - Bring in pages as required
- Page fault
  - Required page is not in memory
  - Operating System must swap in required page
  - May need to swap out a page to make space
  - Select page to throw out based on recent history

- We do not need all of a process in memory for it to run
- We can swap in pages as required
- So - we can now run processes that are bigger than total memory available!
- Main memory is called real memory
- User/programmer sees much bigger memory - virtual memory

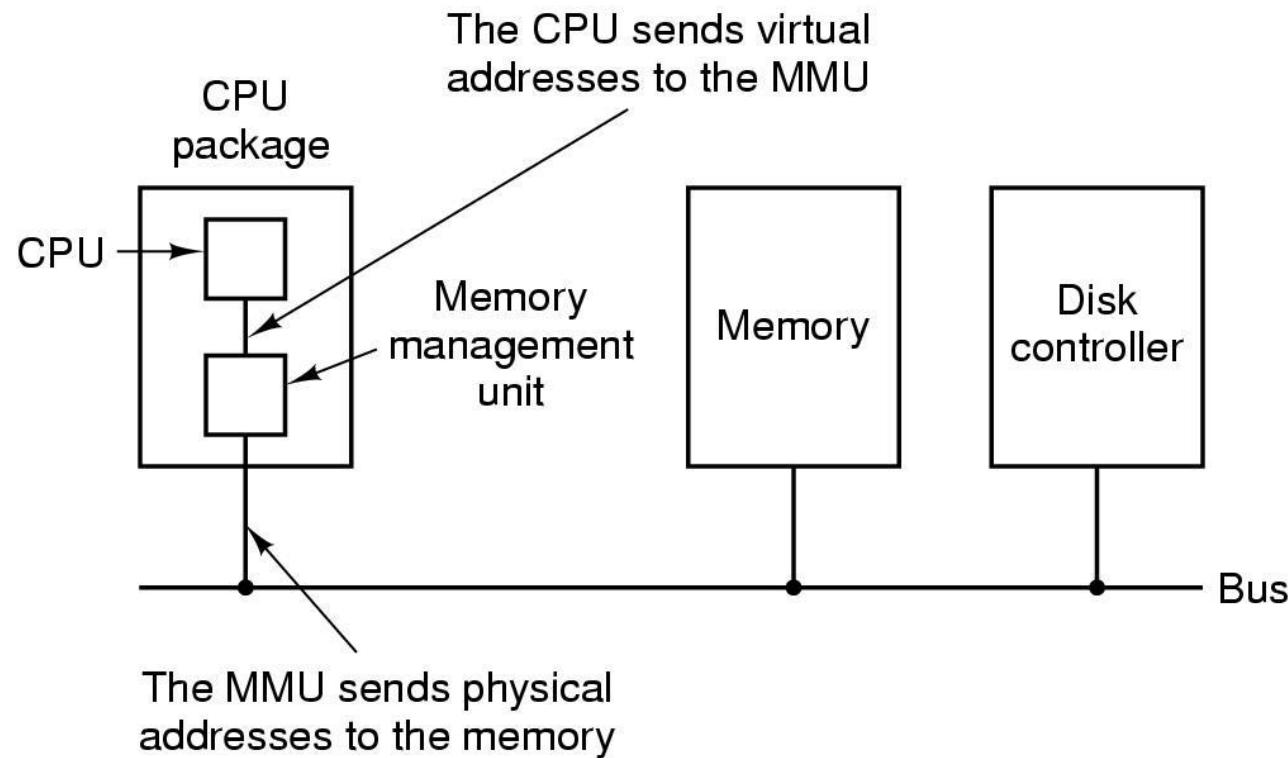
Illusion of larger memory has many advantages.

1. Programmers are practically relieved of the burden of trying to fit a program into limited memory.
2. Some program may run without reprogramming or recompilation on systems with significantly different capacities of installed memory.
3. A process may be loaded into a space of arbitrary size. This may be used to reduce external fragmentation without the need to change the scheduled order of process execution.

4. OS can speed up the execution of important programs by allocating more memory.
5. Degree of multiprogramming can be increased by reducing the real memory for the resident processes.

The maximum program execution speed of a virtual memory system can be equal but never exceed the execution speed of the same program with virtual memory turned off.

# Virtual Memory



The position and function of the MMU

Virtual memory can be implemented as an execution of paged or segmented memory management or as a combination of both.

Accordingly address translation is performed by means of PT, SDT or both.

### **Process of Address Mapping:**

Assume Virtual memory is implemented using paging memory management.

Let Virtual address space be  $V = | 0,1, \dots, V-1 |$  and Physical address space be  $M = | 0,1,.. M-1 |$

At any time the mapping hardware must realize the function  $f : V \rightarrow M$  such that

$f(x) = r$  if item  $x$  is in physical memory at location  $r$  else missing item exception if item  $x$  is not in physical memory.

- The detection of missing item is done by checking the presence indicator bit of each entry of PTs. Before loading the process OS clears all presence bits in the related PT.
- When pages are moved to main memory on demand, corresponding presence bits are set. When a page is evicted from main memory its presence bit is reset.
- The address translation hardware checks the presence bit during the mapping of each memory reference. If the bit is set, the mapping is completed as in paging scheme.

- If the bit is not set then the hardware generates a missing item exception which is referred as page fault in paging systems.
- When a running process experiences a page fault then the process is suspended until the missing page is brought to the main memory.
- Since the disk access time is longer than memory access time, another process is scheduled by OS.
- The disk address of the faulted page is provided by the File Map table (FMT). FMT contains secondary storage addresses of all pages.
- One FMT is maintained for each active process and has the number of entries identical to PT of the process. OS uses the virtual page number to index the FMT and to obtain the related disk address.

- Virtual memory needs more hardware provisions compared to paging and segmentation.
- One instruction may require several pages.
- For example, a block move of data. In some cases page fault may occur during part way through an operation and may have to undo what was done.
- Example: an instruction crosses a page boundary.

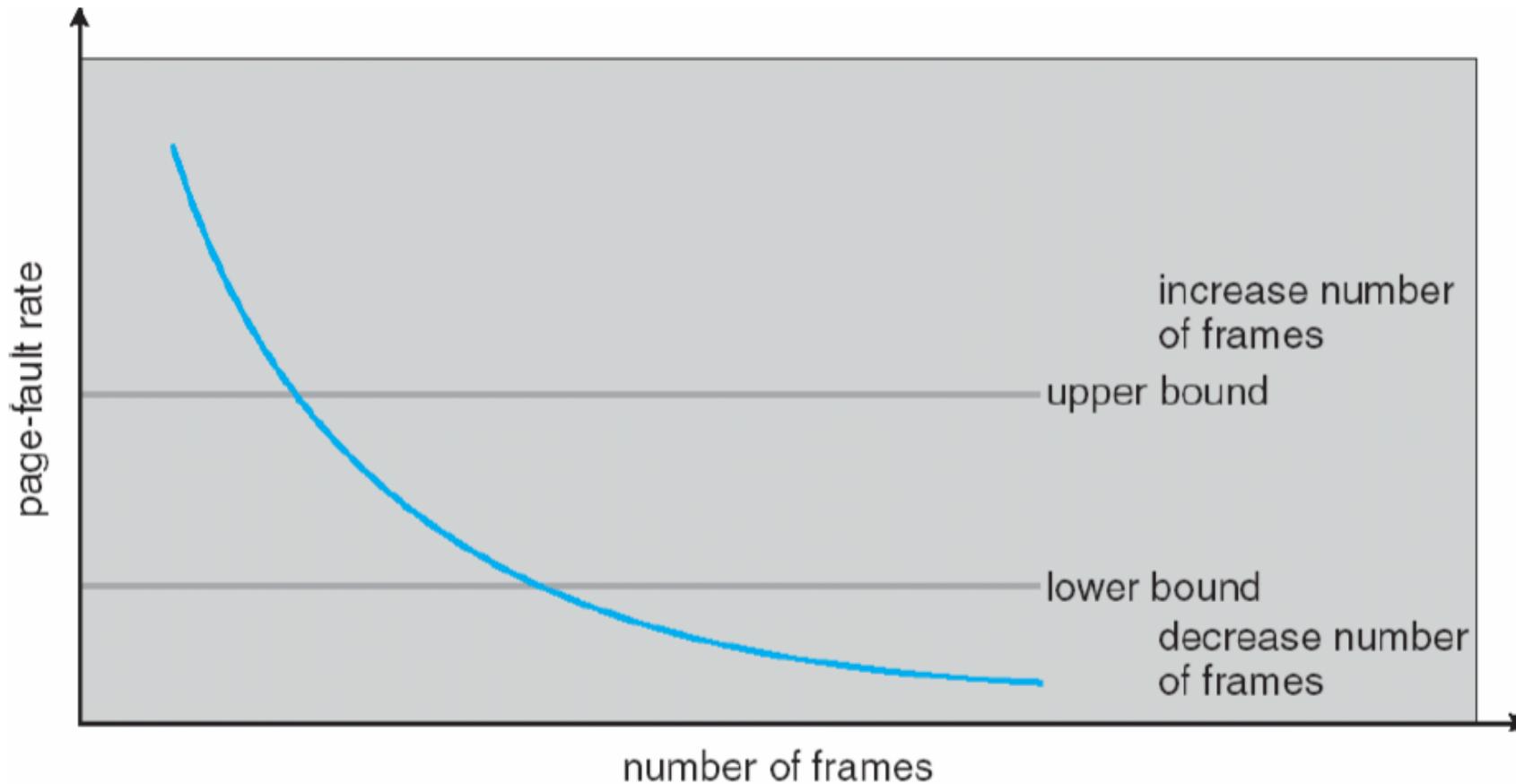
# Management of Virtual Memory

- The allocation of subset of page frames to the virtual address space of a process requires the incorporation of certain policies into the virtual memory manager.
- Allocation policy: How much real memory to allocate to each process?
- Fetch Policy: What are the items to be brought and when to bring them?
- Placement policy: Where to place the incoming item?
- Replacement Policy: If there is no free memory, which item is to be evicted so that the new item can be stored in the real memory.

# Allocation Policy

- OS should determine how many page frames should be allocated to a process.
- If more page frames are allocated then the advantages are reduced page fault frequency and improved turnaround time.
- If too few pages are allocated to a process, then page fault frequency and turnaround times may deteriorate to unacceptable levels and also in systems supporting restarting of faulted instructions page fault frequency may increase.
- The allocation module may fix two thresholds lower and upper for each process. The actual page fault rate experienced by a running process may be measured and recorded in PCB.
- When a process exceeds the upper threshold then more page frames will be allocated. If the process page fault rate reaches the lower threshold then allocation of new page frames may be stopped

# Page fault rate vs allocation of page frames



- The fetch policy is mostly on demand and *Demand paging* is the most common virtual memory management system.
- It is based on the locality model of program execution. As a program executes, it moves from locality to locality.
- *Locality* is defined as a set of pages actively used together. A program is composed of several different localities which may overlap.
- For example, a small procedure when called, defines a new locality. A while-do loop when being executed defines a new locality.

- The Placement policy follows the underlying memory management scheme i.e. paging or segmentation.

# Replacement Policy

- A page replacement algorithm *determines how the victim page (the page to be replaced) is selected when a page fault occurs.* find some page in memory, but not really in use, swap it out.
- There is a possibility that same page may be brought into memory several times. The aim is *to minimize the page fault rate.*

Steps in handling page faults including page replacement are:

1. Check the PT and FMT of the process, to determine whether the reference is a valid or an invalid memory access.
2. If the reference is invalid then the process is terminated; if it is valid and page have not yet brought in, find the location of the desired page on the disk.
3. Find a free frame. If there is a free frame use it. Otherwise, use a page-replacement algorithm to select a victim frame. Write the victim page to the disk; change the PT and MMT accordingly.
4. Schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, modify the FMT, PT and MMT to indicate that the page is now in memory.
6. Restart the user process.

# Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed
  - make room for incoming page
- Modified page must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon

- The efficiency of a page replacement algorithm is evaluated by running it on a particular string of memory references and computing the number of page faults.
- Reference strings are either generated randomly, or by tracing the paging behaviour of a system and recording the page number for each logical memory reference.
- The performance of a page replacement algorithm is evaluated by running it on a particular string of memory references and computing the number of page faults.

# First-In-First-Out (FIFO)

- A FIFO 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. It is easy to implement.
- A FIFO queue/ list is used to hold all pages in memory. The page at the head of the queue is replaced. When a page is brought into memory, it is inserted at the tail of the queue.
- FIFO algorithm is easy to understand and to program.
- Drawback is Belady's Anomaly - When the number of page frames allotted increases the page fault also increases for some cases.

# Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream: A B C A B D A D B C B

Ref:	A	B	C	A	B	D	A	D	B	C	B
Page:											
1	A					D				C	
2		B					A				
3			C					B			

- FIFO: 7 faults.
- When referencing D, replacing A is bad choice, since need A again right away

# Modeling Page Replacement Algorithms

## Belady's Anomaly

All pages frames initially empty

Youngest page

Oldest page

	0	1	2	3	0	1	4	0	1	2	3	4
0												
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

9 Page faults

(a)

Youngest page

Oldest page

	0	1	2	3	0	1	4	0	1	2	3	4
0												
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												

10 Page faults

(b)

- FIFO with 3 page frames
- FIFO with 4 page frames
- P's show which page references show page faults

# Least Recently Used (LRU)

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement. Easy to implement, keep a list, replace pages by looking back into time.
- The OS using this method, has to associate with each page, the time it was last used which means some extra storage.
- In the simplest way, the OS sets the reference bit of a page to "1" when it is referenced. This bit will not give the order of use but it will simply tell whether the corresponding frame is referenced recently or not. The OS resets all reference bits periodically.

# LRU Algorithm Implementations

Counter implementation:

- A time-of-use field is associated with each page-table entry, and a logical clock or counter is added to the CPU.
- The clock is incremented for every memory reference. Whenever a reference to a page is made, the content of the clock register is copied to the time-of-use field in the page table for that page.
- The page with the smallest time value is replaced. It requires a search of the page table to find LRU page and a write to memory for each memory access.

## Stack implementation:

- A stack is used to store page numbers. Whenever a page is referenced, it is removed from the stack and put on the top.
- Top of the stack is always the mostly recently used page and the bottom is the LRU page.
- It is implemented by a double linked list with a head and tail pointer because entries must be removed from the middle of the stack. The tail pointer points to the bottom of the stack (which is LRU page).

Consider the page reference string A B C B B A D A B F B. If LRU page replacement is followed and the number of page frames allocated is 3, find the total number of page faults.

Total number of page faults = 5

A	B	C	B	B	A	D	A	B	F	B
A	A	A	A	A	A	A	A	A	A	A
	B	B	B	B	B	B	B	B	B	B
		C	C	C	C	D	D	D	F	F
*	*	*				*			*	

A	B	C	B	B	A	D	A	B	F	B
A	B	C	B	B	A	D	A	B	F	B
	A	B	C	C	B	A	D	A	B	F
		A	A	A	C	B	B	D	A	A
*	*	*				*			*	

# Optimal Page Replacement Algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. It has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time. (Optimal but unrealizable)
- In this algorithm, the victim is the page which will not be used for the longest period. Estimate by logging page - use on previous runs of process although this is impractical
- It will never suffer from Belady's anomaly.
- OPT is not possible to be implemented in practice because it requires future knowledge. However, it is used for performance comparison.

# Page Buffering algorithm

- The replaced pages will first go to one of two page lists in main memory, depending whether it has been modified or not.
- The advantage of this scheme is that if in a short time, the page is referenced again, it may be obtained with little overhead.
- When the length of the lists surpasses a specific limit, some of the pages that went into the lists earlier will be removed and written back to secondary memory. Thus the two lists actually act as a cache of pages.
- Another advantage is the modified pages may be written out in cluster rather than one at a time, which significantly reduces the number of I/O operations and therefore the amount of time for disk access pool.

# Counting Algorithms

It keeps a counter of the number of references that have been made to each page. Two schemes use this concept.

## 1. Least Frequently Used (LFU) Algorithm:

Page with the smallest count is the one which will be selected for replacement.

This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again

## 2. Most Frequently Used (MFU) Algorithm:

It is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

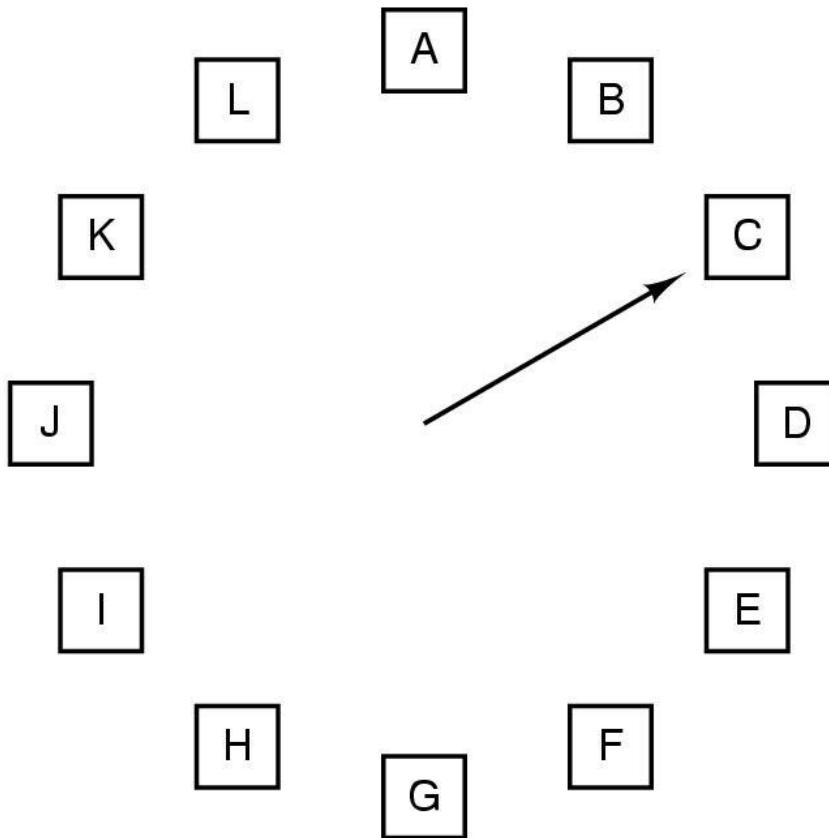
# Second-Chance Algorithm/ Clock Algorithm

- The clock/ second chance policy is basically a variant of the FIFO policy, except that it also considers to some extent the last accessed times of pages.
- When a page has been selected, its reference bit is inspected. If the value is 0 replace the page. If the value is 1, then the page is given a second chance and move on to select the next FIFO page.

When a page gets a second chance:

1. Its reference bit is cleared.
2. Its arrival time is reset to the current time.
3. It will not be replaced until all other pages are replaced or given second chance.

# The Clock Page Replacement Algorithm



When a page fault occurs,  
the page the hand is  
pointing to is inspected.  
The action taken depends  
on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

# Enhanced Second-Chance Algorithm (Not Recently Used)

Each page has Reference bit, Modified bit : bits are set when page is referenced, modified. It uses the reference bit and the modify bit as an ordered pair.

With 2-bits, four classes are possible:

1. (0,0)– neither recently used nor modified (best page to replace).
2. (0,1)– not recently used but modified the page will need to be written out before replacement.
3. (1,0)– recently used but clean (probably will be used again).
4. (1,1)– recently used and modified probably will be used again and write out will be needed before replacing it.

The steps of the algorithm are as follows:

1. Scan the frame buffer and select the first frame with (0, 0) if any.
2. If step 1 fails, scan again, look for a frame with (0,1) and select it for replacement if such a frame exists. During the scan, the reference bits of 1 are set to 0.
3. If step 2 fails, all the reference bits in the buffer are 0. Repeat step 1 and if necessary step 2. This time, a frame will be definitely chosen to be replaced.

The recently accessed pages are given higher priority than those that have been modified, based on the consideration that though the latter need to be written back to secondary memory first before replacement, they still involve less overhead than probable reloading a recently accessed but replaced page.

# Frame Allocation

In order to decide on the page replacement scheme of a particular reference string, the number of page frames available should be known.

In page replacement, some frame allocation policies may be followed.

- Global Replacement: A process can replace any page in the memory.
- Local Replacement: Each process can replace only from its own reserved set of allocated page frames.

In case of local replacement, the operating system should determine how many frames should the OS allocate to each process.

The number of frames for each process may be adjusted by using two ways: Equal ; Proportional

# Design Issues for Paging Systems

## Local versus Global Allocation Policies (1)

	Age
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

A0
A1
A2
A3
A4
A5
B0
B1
B2
B3
B4
B5
C1
C2
C3

(c)

- Original configuration
- Local page replacement
- Global page replacement

# Thrashing

- Too many processes in too little memory
- Operating System spends all its time swapping
- Little or no real work is done
- Disk light is on all the time
- Solutions
  - Good page replacement algorithms
  - Reduce number of processes running
  - Fit more memory

# Working Set Model

To prevent thrashing, a process must be provided with as many frames as it needs. For this, a model called *the working set model* is developed which depends on the locality model of program execution.

The pages used by a process within a window of time are called its working set. A parameter,  $\Delta$ , called the working set window size is used. The set of pages in the last  $\Delta$  page references is the working set of a process.

Choice of  $\Delta$  is crucial. If  $\Delta$  is too small, it will not cover the entire working set. If it is too large, several localities of a process may overlap.

The working set principle states that

1. A program should be run if and only if its working set is in memory.
2. A page may not be removed from memory if it is the member of the working set of a running process.

# Protection and Sharing

- In virtual systems protection and sharing retains the characteristics of the underlying memory management systems such as paging or segmentation. However, the frequent moving of items between main memory and secondary memory may complicate the management of tables.
- If a portion of a shared object is selected for replacement then the concerned tables should be updated accordingly. All copies of the mapping information must be kept in synchrony and updated to reflect the changes of residence of shared objects.

Advantages:

User's point of view:

    Illusion of large address space almost eliminates the considerations imposed by limited physical memory.

    Automatic management of memory makes the program run faster.

OS's point of view:

    Ability to vary the amount of physical memory in use by any program.

    CPU utilization is increased and wastage of memory is reduced.

-

## Disadvantages:

Complex hardware and software needed to support virtual memory.

Both time and space complexities are more when compared to other memory management schemes.

Higher table fragmentation.

Possibility of thrashing leads to complex page replacement algorithms.

Average Turnaround time is increased due to missing item exceptions.

# File System

- Users make use of computers to create, store, retrieve and manipulate information.
- The file abstraction provides a uniform logical view of physical contents from a wide variety of storage devices that have different characteristics.
- A file in a computer system has a name for its identification, space to store data, a location for convenient access, access restrictions and other attributes and support mechanisms.

- OS implements a software layer on top of the I/O subsystem (device drivers) for users to access data with ease from storage devices. The software layer is called the file management system or file system.
- The file management system contains files, directories and control information (metadata) and supports convenient and secured access on files and directories.

The functions of file system are

- Organization of files
- Execution of file operations
- Synchronization of file operations
- Protection of file contents
- Management of space in the file system.

# File concepts

## File

- A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks which normally represents programs and data.
- In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

## Naming

- A symbolic file name given to a file, for the convenience of users, is a string of characters. When a file is named, it is independent of the process, the user and the system it created. It is the only information kept in human readable form.

## File structure

- A file has a certain defined structure, which depends on its type of information stored in the file, like source program, object programs, executable programs, numeric data, text, payroll records, graphic images etc.
  - A text file is a sequence of characters organized into lines.
  - A source file is a sequence of procedures and functions.
  - An object file is a sequence of bytes organized into blocks that are understandable by the machine.
  - When operating system defines different file structures, it also contains the code to support these file structure.

**Attributes** define the characteristics of a file varying with respect to OS and useful for protection, security and usage monitoring.

The following are the file attributes:

- *Name*: The symbolic file name is the only information kept in human readable form
- *Type*: This information is needed for those systems that support different types.
- *Location*: It is a pointer to a device used 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 controls 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.

## Logical file structure

There are three possible forms of atomic units in a file.

- *Bytes*: File is a sequence of bytes called as flat file. (No internal structure)
- *Fixed length record*: Many OS require files to be divided into fixed length records. Each record is a collection of information about one thing. Easy to deal with but do not reflect the realities of data.
- *Variable length records*: These are used to meet the various workload lengths but the problem is unless the location of the file is known it is hard to perform search operation. To overcome this problem keyed file is used. Each record has a specified field which is the key field which is used for finding the location of the file.

# File Meta data

- A file contains information, but the file system also keeps information about the file,
- i.e. meta data (information about information) such as Name, Type, Size and Owner of the file; Group(s) of users that have special access to the file; Access rights; Last Read time; Last Written time; The time of the file was created; which disk the file is stored on and where the file is stored on disk.

# File Types

File type refers to the ability of the operating system to distinguish different types of file such as text files source files and binary files etc. The file system supports various classes of file types. A file may or may not have certain internal structure depending on its type.

Many operating systems support many types of files. Operating system like MS-DOS and UNIX have the following types of files –

## Ordinary files

- These are the files that contain user information.
  - These may have text, databases or executable program.
  - The user can apply various operations on such files like add, modify, delete or even remove the entire file.
- **Directory files**
    - These files contain list of file names and other information related to these files.

## Special files

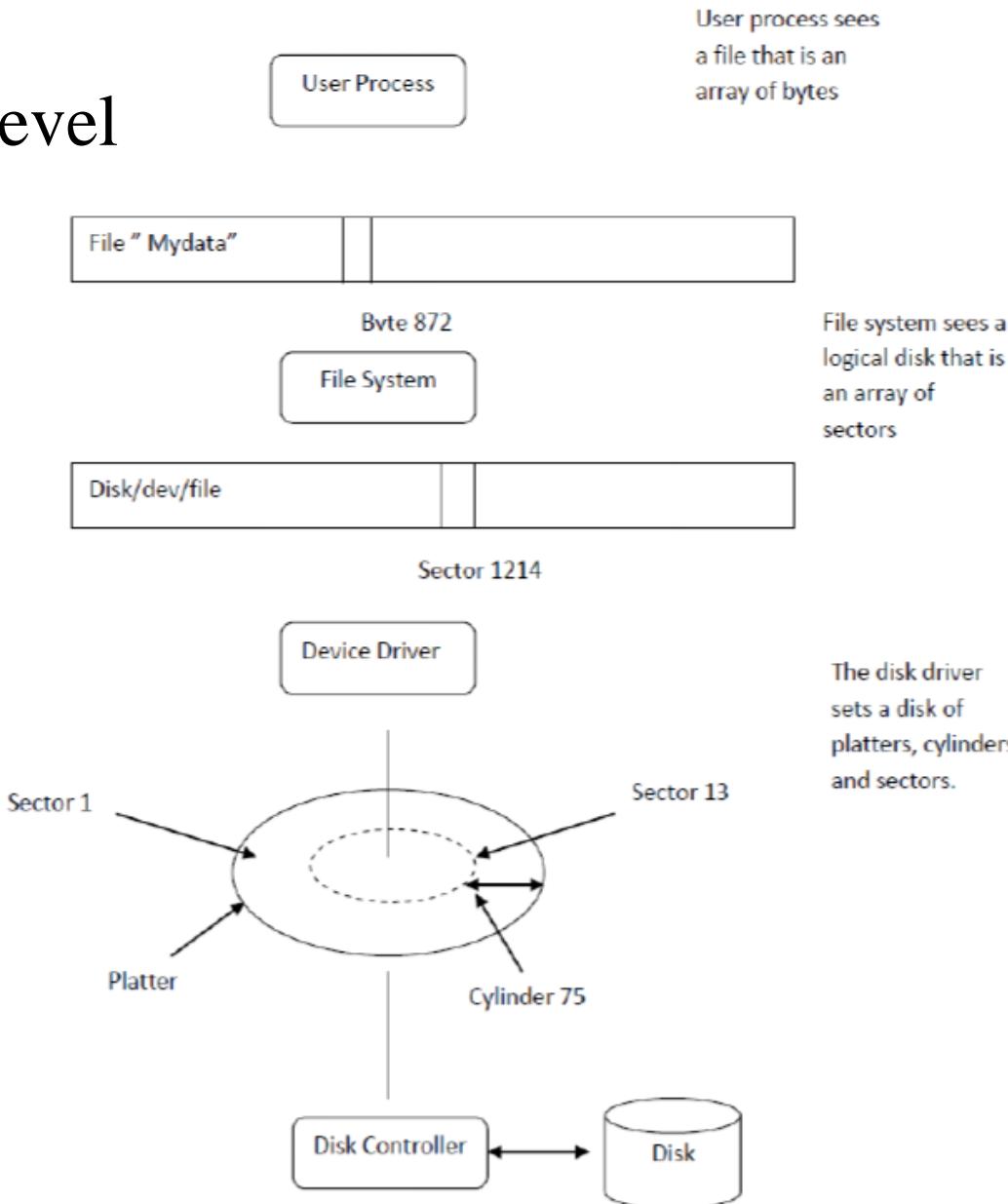
- These files are also known as device files.
- These files represent physical device like disks, terminals, printers, networks, tape drive etc.
- These files are of two types –
  - **Character special files** – data is handled character by character as in case of terminals or printers.
  - **Block special files** – data is handled in blocks as in the case of disks and tapes.

## **File System**

The file system consists of two distinct parts:

- i) a collection of files, each storing related data and
- ii) a directory structure which organizes and provide information about all the files in the system.

# view of a file at each level



# File operations

## Create:

- Essential if a system is to add files. Need not be a separate system call (can be merged with open).
- Requires the name of the file/directory being created, the name of the directory and some file attributes. Two steps are necessary to create a file:
  - space in the file system must be found for the file.
  - an entry for the new file must be made in the directory i.e. the directory entry records the name of the file and the location in the file system.

## Delete:

- Essential if a system is to delete files.
- Requires the file/directory name to be deleted. To delete a file, the directory is searched for the named file and having found the associated directory entry, all file spaces are released and the directory entry is erased.

## Open

- Not essential. An optimization in which the translation from file name to disk locations is performed only once per file rather than once per access.
- Open operation requires a filename to be opened. File system checks for permission to access the file and if the user has the permission, it creates a file descriptor that will be used by the application for future reference.

## Close:

- Not essential. Free resources.
- The close operation requires a file descriptor that was obtained in the open operation. It releases the file descriptor and other related resources allocated for the open file session.

## Read

- Essential. Must specify filename, file location, number of bytes, and a buffer into which the data is to be placed. Several of these parameters can be set by other system calls and in many OS's they are.
- A system call is issued that specifies the name of the file and where in the memory the file should be put for reading.
- A read operation takes as parameters file descriptor, a positive integer number and a buffer address.
- The operation copies into the buffer those many number of consecutive bytes starting at the current file pointer position from the file. It repositions the file pointer past the last byte it read.

## Write

- Essential if updates are to be supported.
- A system call is made specifying both the name of the file and the information to be written to the file.
- A write operation takes as parameters a file descriptor, a byte string and the size of the string. The byte string is written in the file identified by the file descriptor.
- It overwrites the file content starting at the current file pointer position within the file. It allocates more space to the file when it needs to write past the current last byte in the file. It repositions the file pointer after the last byte written.

## Truncate

- To erase the contents of the file but keep attributes same, truncating can be used i.e. attributes of the file remain same but the length of the file is reset to zero.
- A truncate operation takes as its parameters a file descriptor and a positive integer number. It reduces the size of the corresponding file to the specified number. If needed, it frees up space from the file.

## Memory map

- Memory mapping a file creates a region in the process address space, and each byte in the region corresponds to a byte in the file.
- Conventional memory read and write operations on the mapped sections by applications are treated by the system as file read and write operations respectively.
- When the mapped file is closed, all the modified data are written back to the file and the file is unmapped from the process address space.

## File Pointer

- On systems that do not include a file offset as part of the read and write system calls, the system must track the last read / write location as current file position pointer.
- This pointer is unique to each process operating on the file, and therefore must be kept separate from the disk file attributes.

## Reposition

- adjusts a file pointer to a new offset. The operation takes a file descriptor and an offset as parameters and sets the associated file pointer to the offset. The directory is searched for the appropriate entry and current file position is set to a given value.

## Seek

- Not essential (could be in read/write). Specify the offset of the next (read or write) access to this file. Repositioning within a file does not need to involve any actual I/O. This file operation is also known as file seek.

# Directory

# Directories

Files in an OS are generally named using a hierarchical naming system based on directories. Almost all OS use a hierarchical naming system. In such a system, a path name consists of component names separated with a separator character.

- A directory is a name space and the associated name maps each name into either a file or another directory.
- Directories contain bookkeeping information about files.
- Directories are usually implemented as files and in OS point of view there is no distinction between files and directories.

# Directory operations

- Create: Produces an ``empty'' directory. Normally the directory created actually contains . and .., so is not really empty
- Delete: Requires the directory to be empty (i.e., to just contain . and ..). Commands are normally written that will first empty the directory (except for . and ..) and then delete it. These commands make use of file and directory delete system calls.
- Opendir: Same as for files (creates a ``handle'')
- Closedir: Same as for files
- Readdir: In the old days (of unix) one could read directories as files so there was no special readdir (or opendir/closedir). It was believed that the uniform treatment would make programming (or at least system understanding) easier as there was less to learn.
- Rename: As with files
- Link: Add a second name for a file;
- Unlink: Remove a directory entry. But if there are many links and just one is unlinked, the file remains.

- The structure of the directories and the relationship among them are the main areas where file systems tend to differ, and it is also the area that has the most significant effect on the user interface provided by the file system.
- The most common directory structures used by multi-user systems are:
  - **Single-level directory**
  - **Two-level directory**
  - **Tree-structured directory**
  - **Acyclic directory**

## **Single-Level Directory**

- In a single-level directory system, all the files are placed in one directory. This is very common on single-user OS's.
- Limitations
  - Unique name is a problem
    - More number of files
    - More than one user

## **Two-Level Directory**

- In the two-level directory system, the system maintains a master block that has one entry for each user. This master block contains the addresses of the directory of the users.

### **Limitations**

- This structure effectively isolates one user from another. This is an advantage when the users are completely independent, but a disadvantage when the users want to cooperate on some task and access files of other users.

## **Tree-Structured Directory**

- In the tree-structured directory, the directory themselves are files. This leads to the possibility of having sub-directories that can contain files and sub-subdirectories.

### **Limitations**

- An interesting policy decision in a tree-structured directory structure is how to handle the deletion of a directory.
- If a directory is empty, its entry in its containing directory can simply be deleted.
- However, suppose the directory to be deleted is not empty, but contains several files, or possibly sub-directories.
- Some systems will not delete a directory unless it is empty.

# Acyclic-Graph Directory

- The acyclic directory is an extension of the tree-structured directory structure. In the tree-structured directory, files and directories starting from some fixed directory are owned by one particular user.
- In the acyclic structure, this prohibition is taken out and thus a directory or file under directory can be owned by several users.

# Path Name

- An absolute path name is a name that gives the path from the root directory to the file that is named.
- When hierarchical file system is used, the absolute path may be long if many levels of subdirectories are present.
- Hierarchical naming system use compound names with several parts. Each part is looked up in a flat name space usually called a directory. If this look up leads to another directory, then the next part of the compound name is looked up in that directory.
- To reduce the length of the path name, working/current directory concept is used. This allows the use of relative path names.
- A relative path name is a path name that starts at the working directory rather than the root directory.

## Aliases

- Hierarchical directory systems provide a means for classifying and grouping files.
- All the files in a single directory tend to be related in some way, but sometimes a file is related to several different groups and it is an inconvenient restriction to have no place it in just one group.
- This problem can be handled with the concept of a file alias. It is a file name that refers to a file that also has another name. There may be two or more absolute path names.

# File Access Mechanisms

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files –

Sequential access

Direct/Random access

Indexed sequential access

## Sequential access

A sequential access is that in which the records are accessed in some sequence, i.e., the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.

## Direct/Random access

Random access file organization provides, accessing the records directly.

Each record has its own address on the file with by the help of which it can be directly accessed for reading or writing.

The records need not be in any sequence within the file and they need not be in adjacent locations on the storage medium.

## Indexed sequential access

This mechanism is built up on base of sequential access.

An index is created for each file which contains pointers to various blocks.

Index is searched sequentially and its pointer is used to access the file directly.

# File System Implementation

# File System Implementation

- The file system is implemented using files, directories and the files that are currently opened.
- Each one will be implemented as a data structure and a set of procedures to manipulate the data structures

The open file table contains a structure for each open file. An open file structure contains the following information

- Current file position
- Other status information about the open file (eg. Whether the file is locked or not)
- Pointer to the file descriptor that is open

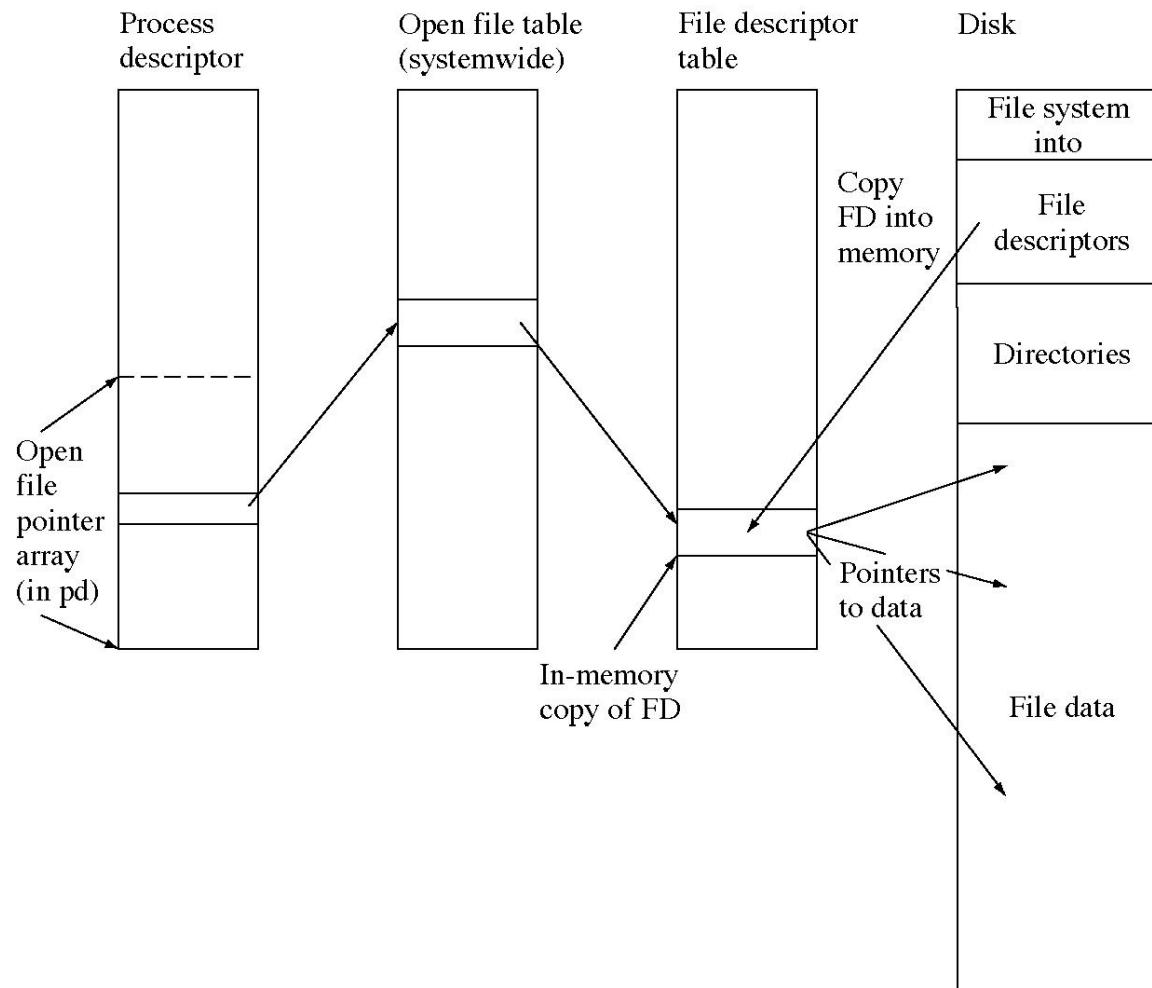
Most open files are connected to files. Files exist on disk and consist of two parts

- File descriptor : contains all the meta data about the file
- File data: Kept in disk blocks

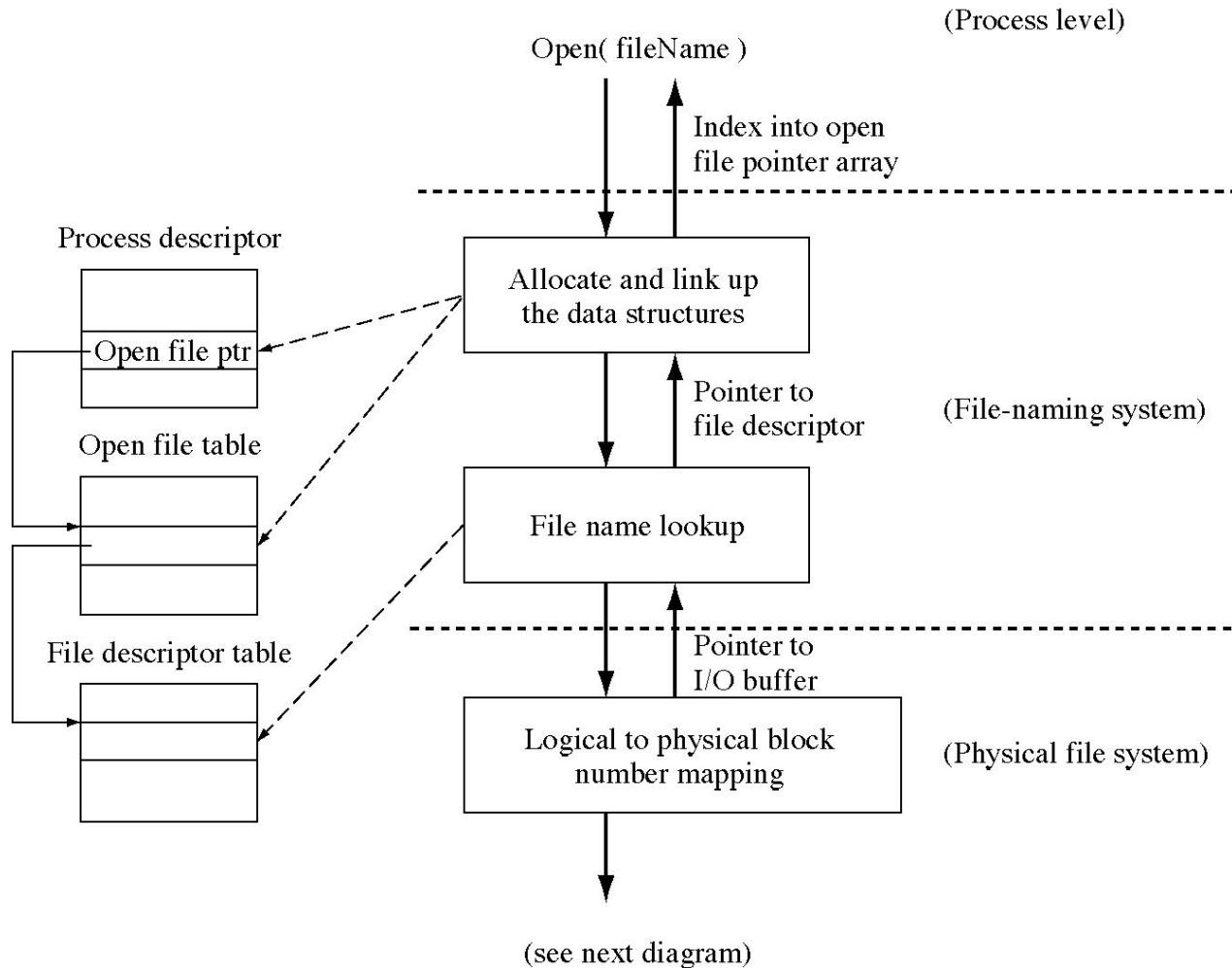
File Descriptor data structure contains the following information

- Owner of the file
- File protection information
- Time of creation, last modification and last use
- Other file meta data
- Location of the file data

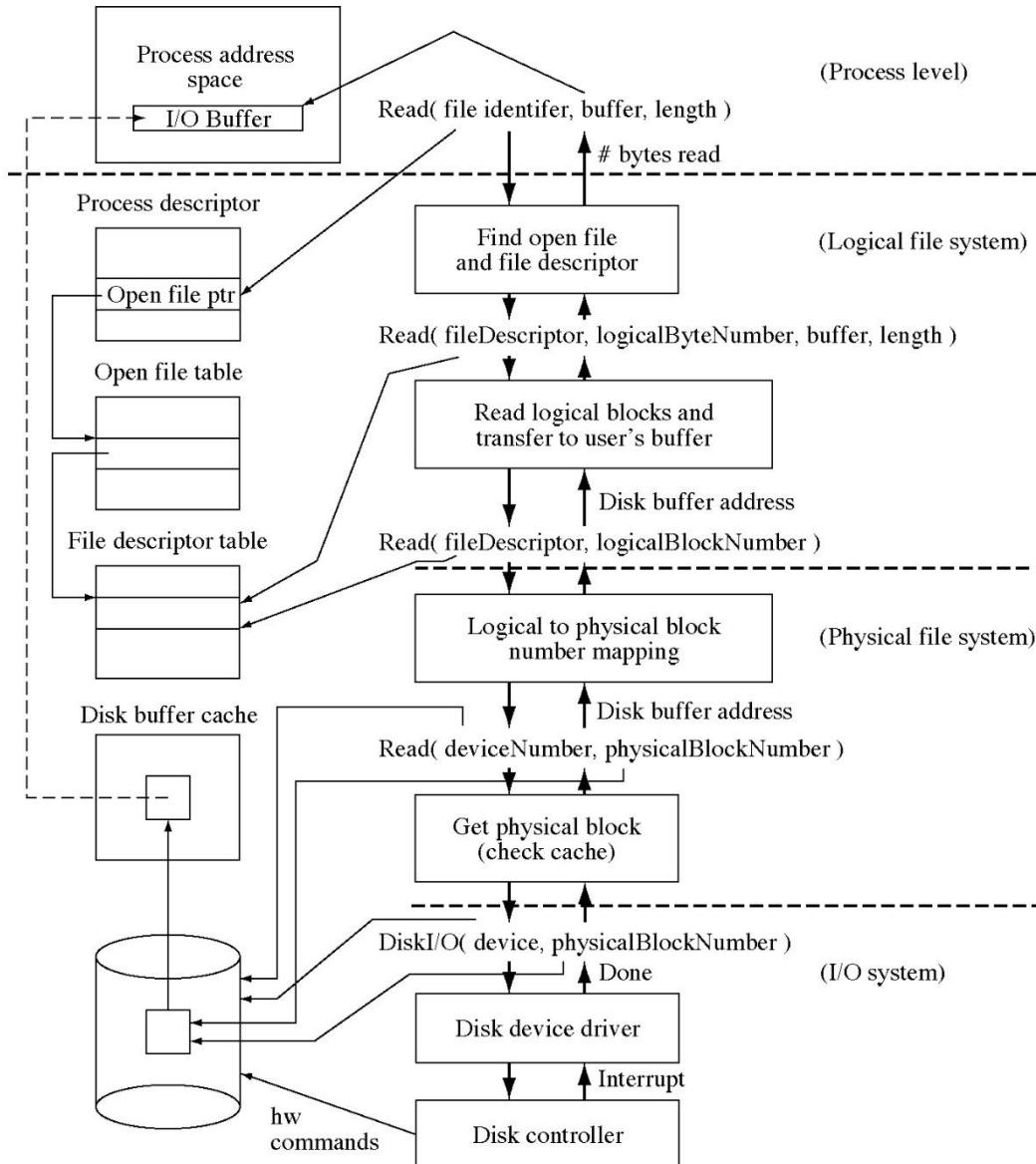
# File system data structures



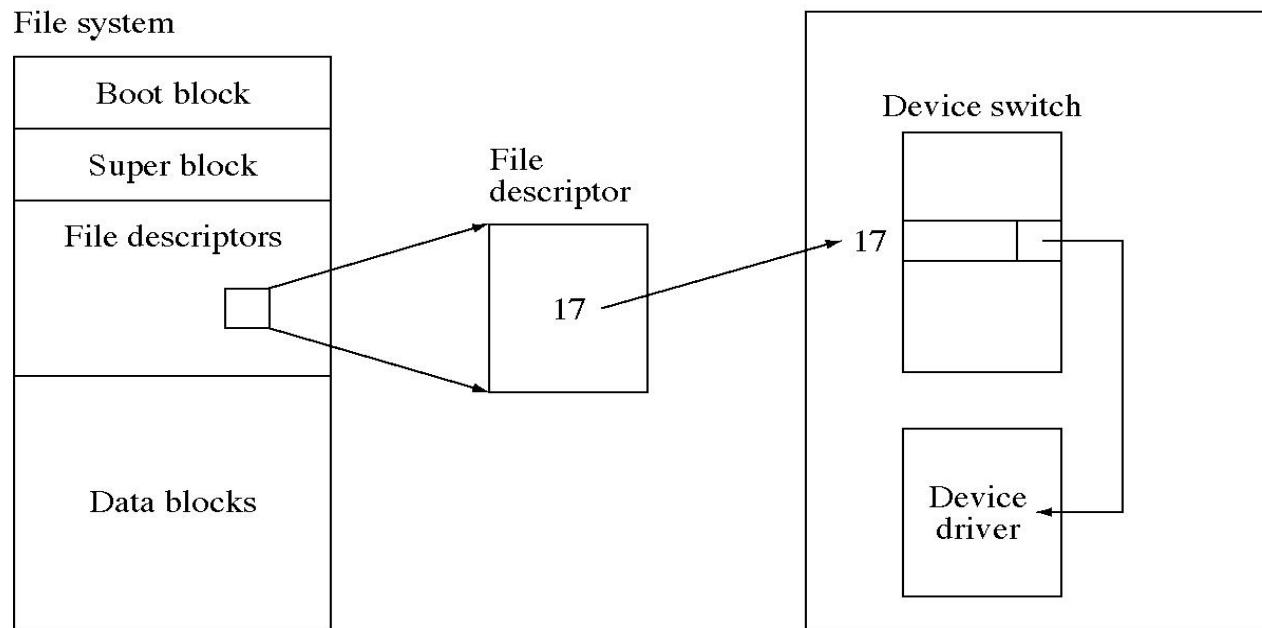
# Flow of control for an open



# Flow of control for a read



# Connecting files and devices



# Directory Implementation

A directory is a table that maps component names to file descriptors. The steps involved in a path lookup are

- 1) Let FD be the root if the path starts with '/' or FD be current directory and start at the beginning of the path name.
- 2) If the path name is at the end then return FD.
- 3) Isolate the next component in the path name and let it be C. Move past the component in the path name.
- 4) if FD is not a directory, then return an error.
- 5) Search through the directory FD for the component name C. This involves a loop that reads the next name/fd pair and compares the name with C. Loop until a match is found or the end of the directory is reached.
- 6) If no match is found, then return an error.
- 7) If a match was found, then its associated file descriptor becomes the new FD. Go back to step2.

# Free Space Management

# Space Allocation

Files are allocated disk spaces by operating system. Operating systems deploy following three main ways to allocate disk space to files.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

## **Contiguous Allocation**

- Each file occupies a contiguous address space on disk.
- Assigned disk address is in linear order.
- Easy to implement.
- The difficulty with contiguous allocation is finding space for a new file. If the file to be created is  $n$  blocks long, then the OS must search for  $n$  free contiguous blocks.
- First-fit, best-fit, and worst-fit strategies are the most common strategies used to select a free hole from the set of available holes.
- External fragmentation is a major issue with this type of allocation technique.
- Another problem with contiguous allocation is determining how much disk space is needed for a file.

## **Linked Allocation**

- Each file carries a list of links to disk blocks.
- Directory contains link / pointer to first block of a file.
- This pointer is initialized to NIL (the end-of-list pointer value) to signify an empty file.
- A write to a file removes the first free block and writes to that block. This new block is then linked to the end of the file.
- To read a file, the pointers are just followed from block to block.
- No external fragmentation
- Effectively used in sequential access file.
- Inefficient in case of direct access file.

## **Indexed Allocation**

- Provides solutions to problems of contiguous and linked allocation.
- A index block is created having all pointers to files.
- Each file has its own index block which stores the addresses of disk space occupied by the file which is an array of disk sector of addresses. The  $i^{\text{th}}$  entry in the index block points to the  $i^{\text{th}}$  sector of the file.
- Directory contains the addresses of index blocks of files.
- To read the  $i^{\text{th}}$  sector of the file, the pointer in the  $i^{\text{th}}$  index block entry is read to find the desired sector.
- Indexed allocation supports direct access, without suffering from external fragmentation. Any free block anywhere on the disk may satisfy a request for more space.

# Free Space List

- Since there is only a limited amount of disk space, it is necessary to reuse the space from deleted files for new files.
- To keep track of free disk space, the system maintains a free-space list.
- The free-space list records all disk blocks that are free (i.e., are not allocated to some file).
- To create a file, the free-space list has to be searched for the required amount of space, and allocate that space to a new file. This space is then removed from the free-space list.
- When a file is deleted, its disk space is added to the free-space list.

# Free Space List

## Bit-Vector

- Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by a 1 bit. If the block is free, the bit is 0; if the block is allocated, the bit is 1.
- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be: 11000011000000111001111110001111...
- The main advantage of this approach is that it is relatively simple and efficient to find 'n' consecutive free blocks on the disk.
- Unfortunately, bit vectors are inefficient unless the entire vector is kept in memory for most accesses.
- Keeping it main memory is possible for smaller disks such as on microcomputers, but not for larger ones.

## Linked List

- In this approach all the free disk blocks are linked together, keeping a pointer to the first free block. This block contains a pointer to the next free disk block, and so on.
- In the previous example, a pointer could be kept to block 2, as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on.
- This scheme is not efficient; to traverse the list, each block must be read, which requires substantial I/O time.

# Grouping

- A modification of the free-list approach is to store the addresses of 'n' free blocks in the first free block.
- The first  $n-1$  of these is actually free. The last one is the disk address of another block containing addresses of another  $n$  free blocks.
- The importance of this implementation is that addresses of a large number of free blocks can be found quickly.

# Counting

- Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when contiguous allocation is used.
- Thus, rather than keeping a list of free disk addresses, the address of the first free block is kept and the number 'n' of free contiguous blocks that follow the first block.
- Each entry in the free-space list then consists of a disk address and a count.

# I/O Software

- The I/O software should provide interface between devices and the rest of the system and also the interface should be device independent.
- The other issues to be considered in design of I/O software are uniform naming of devices, error handling, blocking/ non-blocking transfer of data, handling shared or exclusive devices.
- The synchronization between CPU and I/O controller is performed by the interrupt processing or by busy wait handshaking. In some cases, data may be directly transferred from device to memory or vice versa.

# Performing I/O

## **Polling/Programmed I/O**

- CPU directly controls the I/O operation by issuing commands on behalf of a process to an I/O module.
- It constantly monitors a command execution by reading the controller status register.
- Process is busy-waiting for the operation to complete

## **Interrupt driven I/O**

- CPU issues a command on behalf of a process to an I/O module and continues to execute next instructions from the same process if the I/O operation is non-blocking else if the I/O operation is blocking then OS suspends the current process, and assigns CPU to another process.
- The controller has an embedded interrupt circuit and after completion of I/O operation, it interrupts the CPU on command process completion and CPU reads the command execution status information from the status ports.

## **Direct memory access (DMA)**

- When a large amount of data is to be transferred between the host system and an I/O controller, the direct mode of data transfer by CPU (byte by byte or word by word) is inefficient and interrupt handling adds substantial overhead in data transfer.
- If handshake mode of data transfer is used CPU spends a considerable amount of time reading the device status and output ports.
- So to transfer large volumes of data without involving CPU, an additional hardware (Direct Memory Access device) is needed.
- DMA module controls exchange of data between memory and an I/O module.
- A DMA device assists the I/O controllers in transferring data between them and with the main memory without involving CPU.
- Processor is interrupted only after entire block has been transferred

The steps involved are:

- CPU sets up a memory buffer for data transfer and sends a request to transfer a block of data to the DMA module. The information sent are
  - Request type – read or write; using the read or write control line between the CPU and DMA module
  - Address of the I/O device, on the data line
  - Starting location in memory for read/write; communicated on data lines and stored by DMA module in its address register
  - Number of words to read/write; communicated on data lines and stored in the data count register
- CPU engages in some other activities.
- The DMA module takes over control of system bus to perform the data transfer to/from the buffer and on completion, I/O controller interrupts the CPU.

Three popular modes of DMA transfer:

- Cycle stealing: A DMA device transfers data using bus interleaving, i.e. it steals the host bus for its purpose.
- Burst mode or Block mode: The DMA controller transfers a block of data making uninterrupted use of the bus. Other devices are not permitted to access the bus during the burst mode of transfer.
- Fly by mode or Single access mode: This mode transfers data at high speed between source and destination. The data transfer is done in a single cycle and during the transfer, the DMA controller simultaneously enables control signals to both the source and destination.

# I/O Software Layer

Layers of the I/O system and the main function of each layer

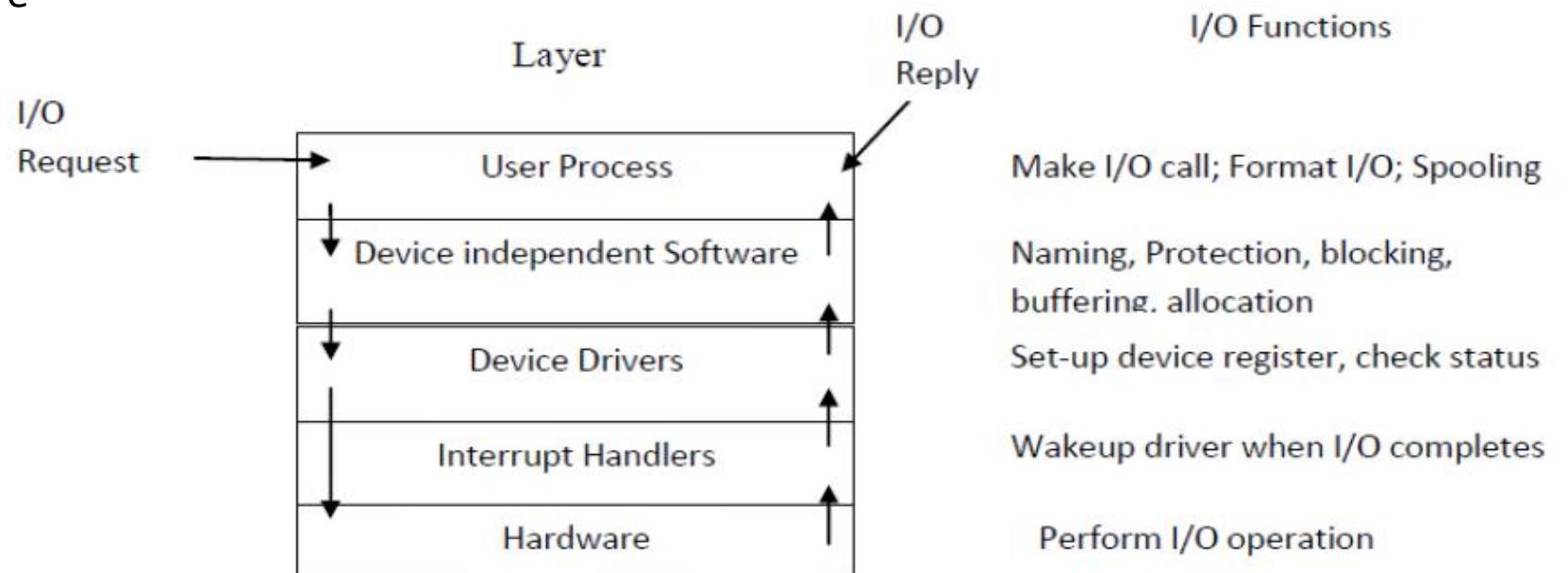
Four layers

Interrupt Handler

Device Driver

Device independent software

User level I/O software



## **Interrupt Handler:**

- Interrupt driven I/O are commonly used and they should be hidden from OS. The best way to hide them is to block the driver that start the I/O operation until the I/O has completed and the interrupt occurs.
- When the interrupt happens the interrupt procedure does whatever it has to do in order to handle the interrupt. Then it can unblock the driver that started it.

Steps that must be performed in software after the hardware interrupt has completed.

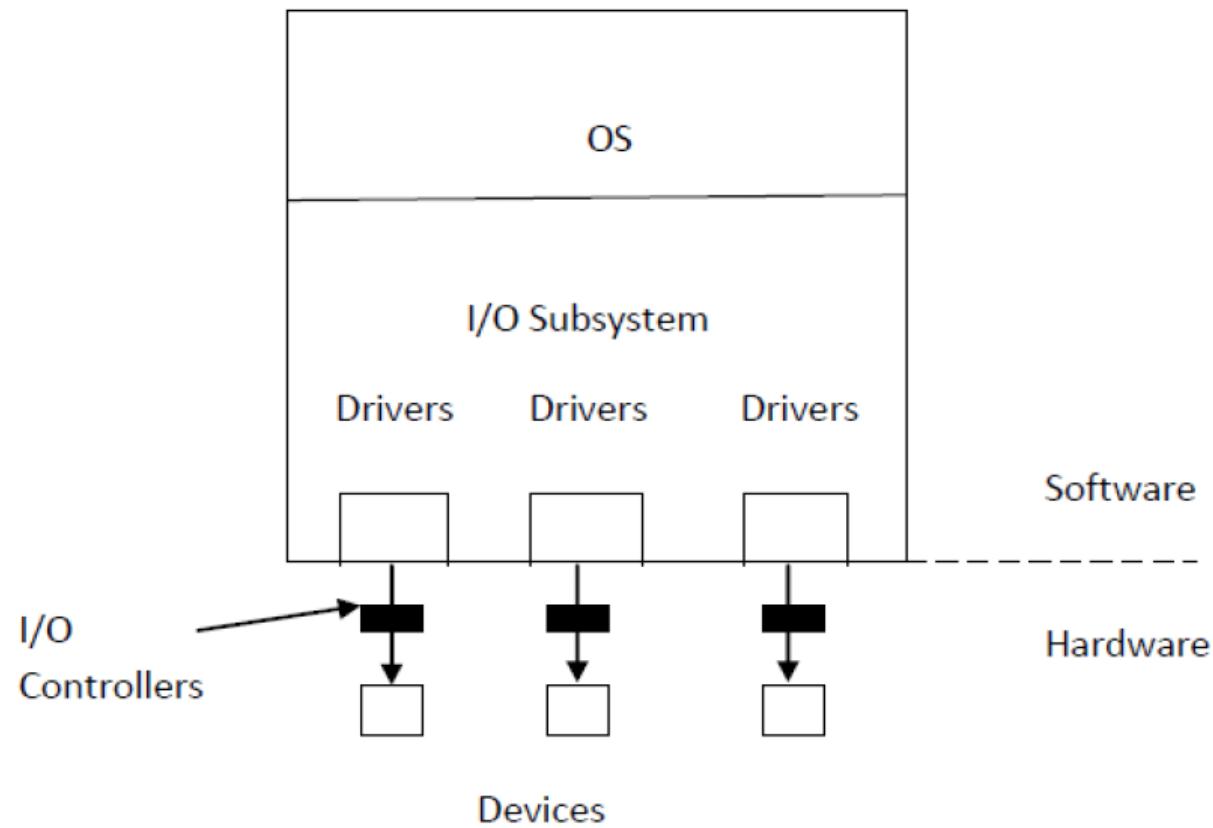
1. Save the contents of all registers (including PSW).
2. Set up a context for the interrupt service procedures. (i.e. setting up the TLB, MMU, page table etc.)
3. Set up a stack for the interrupt service procedure.
4. Acknowledge the interrupt controller and re-enable the interrupts if there is no centralized interrupt controller.
5. Copy the registers values to the process table.

6. Run the interrupt service procedure. It will extract information from the interrupting device controller's register.
7. Choose the next process to run.
8. Set up the MMU context for the process to run next.
9. Load the new process registers including PSW.
10. Start running the new process.

## **Device drivers**

- Device drivers reside in the kernel space and interact with the rest of the kernel through a specific interface.
- A device driver manipulates the I/O controller and the devices connected to it.
- It implements the software interface that enables the OS to access data from I/O devices by executing the driver program
- An interface is a collection of routines and data structures that operate in a well-defined way.
- Each driver is customized to a specific I/O controller. The drivers implement uniform interface for the OS by hiding the differences among the I/O controllers.

# Device driver interface to I/O Devices



- A device driver can be visualized as an abstract object or a monitor accessed by a predefined set of operations.
- Device driver software consists of a set of private data structures, a set of device dependent routines and Kernel Device Interface Model (KDIM) device independent routines.
- When the driver routines are executed, the driver is said to be controlling activities of the I/O controller and its connected devices.

- The advantages of extra layer between hardware and applications are:
- Users need not study low-level programming characteristics of hardware devices and so programming is easier;
- System security is increased; kernel can check accuracy of request at the interface level before attempting to satisfy it
- Uniform interface makes programs more portable; programs compiled and executed correctly on every kernel that offers the same set of interfaces

## **Device independent I/O software**

Some parts of I/O software is device specific, where as other parts are device independent. The exact boundary between the drivers and the device independent software is system dependent.

Functions of the Device independent I/O software:

- Uniform interfacing for device drivers
  - How to make all I/O devices and drivers look more or less the same
- All drivers have the same interface.
  - How I/O devices are named
- In Unix, i-node contains major device number which is used to locate appropriate driver
  - How does the system prevent users from accessing devices that they are not entitled to access
- Protection using permissions to access.

- Buffering
  - is needed to reduce the number of interrupts for reading and writing data from /to I/O devices such as block and character devices
  - is part of user space and/or kernel space.
- Error Reporting
  - Many errors are device specific and must be handled by the appropriate driver, but the framework for error handling is device independent.
  - Errors may be programming errors or I/O errors
  - Different ways of handling errors are ignoring errors, killing the calling process or displaying error messages and terminating the I/O operation.

- Allocating and releasing dedicated drivers.
  - Some devices such as CD-ROM recorders are exclusive in nature. It is upto the OS to examine requests for device usage and accept or reject them, depending on whether the requested device is available or not.
- Providing a device independent block size.
  - Hiding the different sector sizes of disks; number of bytes transferred by devices such as modem. Network interfaces etc.

## **User – space I/O software**

- Small portion of the I/O software either consists of libraries linked together with the user programs (I/O system calls) or spooling system for managing serially accessible devices.

# I/O Systems

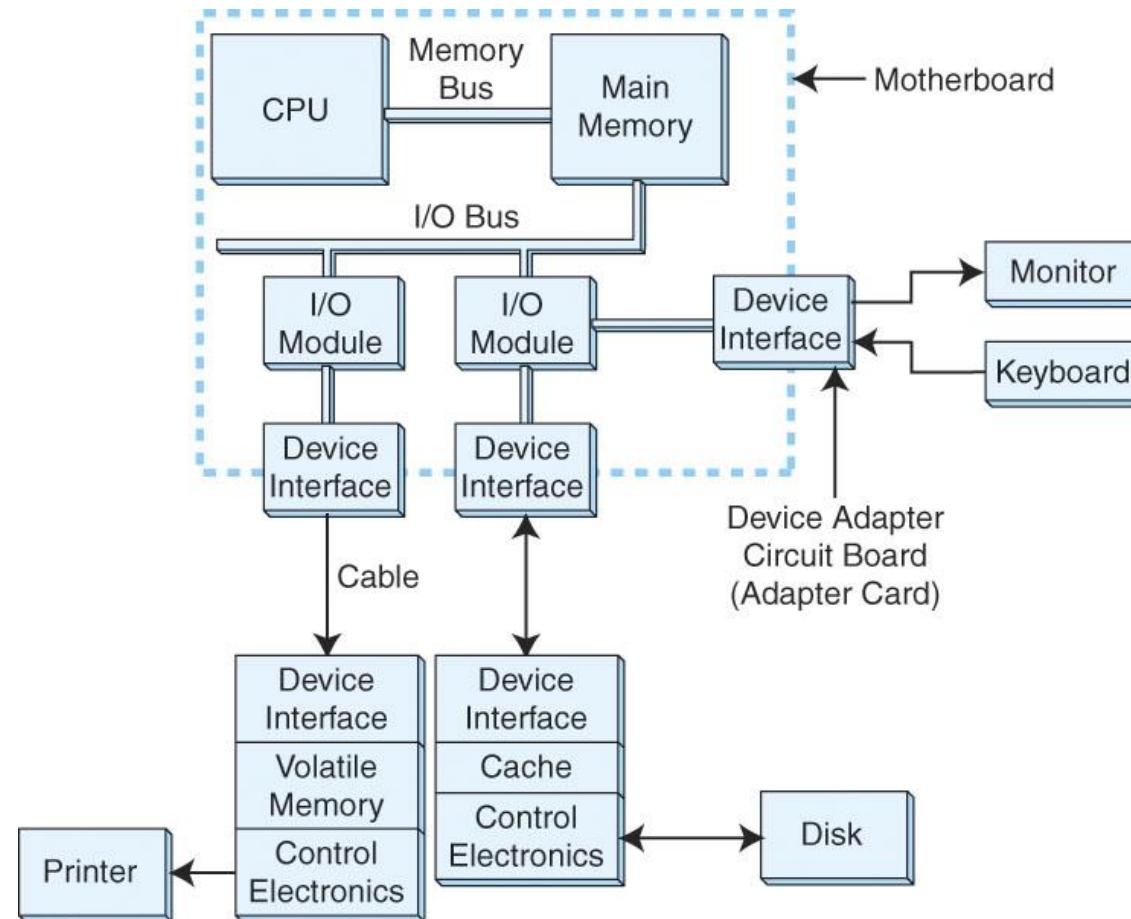
- Modern computers use various types of I/O devices. The devices are used in different environments and designing interfaces to these devices is very difficult. These devices are managed by special software called device drivers.
- The functions of the I/O systems are to manage variety of device drivers, to schedule I/O requests to devices, to allocate devices to requesting processes, to match the speed of the I/O devices with CPU/Memory using buffering /caching data etc.

- The I/O subsystem is treated as an independent unit in the computer
  - The CPU initiates I/O commands generically
    - Read, write, scan, etc
    - This simplifies the CPU
  - I/O modules are components that connect an I/O device to the I/O bus
    - The I/O module is an intermediary between CPU and the I/O device, and possibly between memory and the I/O device
  - This allows us to tailor I/O devices to specific uses without having to worry about how the CPU might be able to handle that new type of device
  - In addition, while the CPU may initiate an I/O operation, once begun, the I/O module takes over so that the CPU can get back to doing whatever it was doing

- I/O subsystem will typically include
  - Blocks of memory dedicated to I/O buffering
  - I/O bus(es)
  - I/O devices
  - Specialized interfaces (for instance for keyboard and monitor) and interface cards
  - Possibly other connections (network, cable, etc)

- Main tasks of I/O system:
  - Present **logical** (abstract) view of devices
    - Hide details of hardware interface
    - Hide error handling
  - Facilitate **efficient** use
    - Overlap CPU and I/O
  - Support **sharing** of devices
    - Protection when device is shared (disk)
    - Scheduling when exclusive access needed (printer)

# I/O Architectures



# I/O Hardware

- Any I/O device is hosted by one and only one I/O bus which is the data path connecting the CPU and an I/O device.
- I/O bus is connected to an I/O device by a hierarchy of hardware components, including I/O ports, interfaces, and device controllers.

- An I/O controller is a peripheral device that enables the main processor to transfer data between the host system and the I/O devices.
- It is a special purpose processor and carries out I/O operations in parallel with CPU execution of programs.
- The CPU interacts with an I/O controller through a set of controller interface registers called I/O ports.
- The ports constitute the I/O address space of a controller and I/O address spaces of all I/O controllers together constitute the I/O address space of the computer system.

- I/O ports are classified into four groups: Command, Status, Input and Output.
- Command and status registers are used to start and stop all devices connected to the controller, to initialize them and to diagnose any problems encountered with them.

The steps to start an I/O operation:

1. CPU gives appropriate commands to ports and input data to Input ports.
2. Controller executes the command and writes back the command execution status to the status ports and output data to output ports.
3. CPU reads the status and output ports respectively to find the command completion status and the output produced.

- The devices have different physical organizations with various physical characteristics such as the recording medium, storage capacity, interface operations and the access speed.
- Examples of I/O devices are disks, tapes, CDs, printers, network interface cards, keyboards, monitors etc.
- Some devices are purely input devices like keyboard, some are purely output devices like monitors and some are both like disks.

# Classification of devices

- Human readable
- Machine readable
- Communication

# Categories of I/O Devices

- Human readable
  - Used to communicate with the user
  - Printers
  - Video display terminals
    - Display
    - Keyboard
    - Mouse

# Categories of I/O Devices

- Machine readable
  - Used to communicate with electronic equipment
  - Disk and tape drives
  - Sensors
  - Controllers
  - Actuators

# Categories of I/O Devices

- Communication
  - Used to communicate with remote devices
  - Digital line drivers
  - Modems

Another way of classifying I/O devices based on the nature of their function is

- Storage devices - Data are stored persistently
- Communication devices - Data are transferred from one hardware component to another.

- The devices may also be categorized based on the physical attributes:
- Character or Block: A device may transfer one character/byte or one block (multiple bytes) of data at a time. Byte is smallest unit of data transfer for a character device and fixed block size is the smallest unit for a block device.
- Read-only, write-only or read-write: Some devices such as keyboards allow the system to read characters whereas printers are write-only which allow system to perform write operation; Devices such as disks allow the system to load as well as store data.

- Speed: Some devices such as keyboards have very low transferring speed (in terms of bytes per second) and some devices are faster like Ethernet cards that transfer millions of bytes per second.
- Transient or durable: Transient devices store data for shorter duration as in Ethernet cards whereas disks store persistent data.
- Sharable or exclusive: Some devices such as disks are sharable and can be accessed by processes concurrently. Some devices are exclusive in nature like Graphical plotters.

- Access pattern:
  - Sequential access devices: Data blocks are only accessible in a sequential order. e.g. Tape devices
  - Direct access devices: Any data block is directly accessible in any arbitrary order by providing its position. e.g. Disks
  - Random access devices: Data block are directly accessible but the access time is independent of the position of data. e.g. Flash devices.

# Differences in I/O Devices

- Data rate
  - May be differences of several orders of magnitude between the data transfer rates
- Application
  - Disk used to store files requires file management software
  - Disk used to store virtual memory pages needs special hardware and software to support it
  - Terminal used by system administrator may have a higher priority
- Complexity of control
- Unit of transfer
  - Data may be transferred as a stream of bytes for a terminal or in larger blocks for a disk
- Data representation
  - Encoding schemes
- Error conditions
  - Devices respond to errors differently

# I/O Buffering

- Reasons for buffering
  - Processes must wait for I/O to complete before proceeding
  - Certain pages must remain in main memory during I/O

# I/O Buffering

- Block-oriented
  - Information is stored in fixed sized blocks
  - Transfers are made a block at a time
  - Used for disks and tapes
- Stream-oriented
  - Transfer information as a stream of bytes
  - Used for terminals, printers, communication ports, mouse and other pointing devices, and most other devices that are not secondary storage

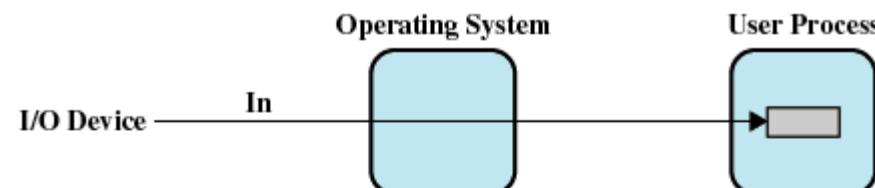
# Single Buffer

- Operating system assigns a buffer in main memory for an I/O request
- Block-oriented
  - Input transfers made to buffer
  - Block moved to user space when needed
  - Another block is moved into the buffer
    - Read ahead
  - User process can process one block of data while next block is read in
  - Swapping can occur since input is taking place in system memory, not user memory
  - Operating system keeps track of assignment of system buffers to user processes

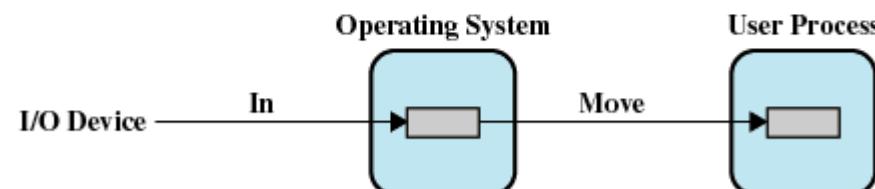
# Single Buffer

- Stream-oriented
  - Used a line at time
  - User input from a terminal is one line at a time with carriage return signaling the end of the line
  - Output to the terminal is one line at a time

# I/O Buffering



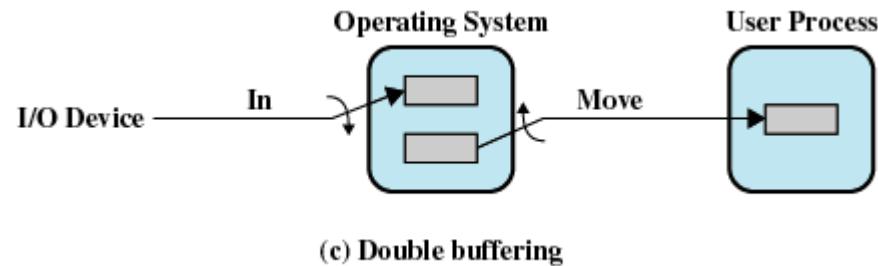
(a) No buffering



(b) Single buffering

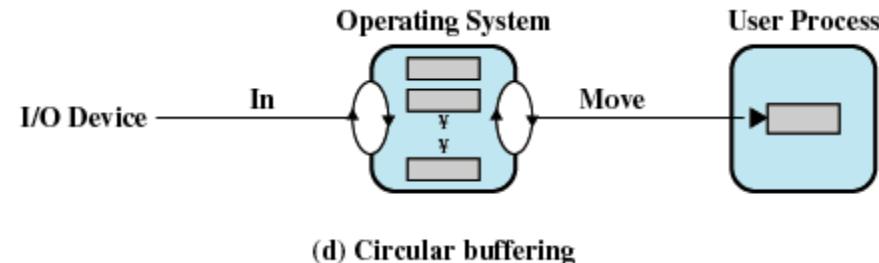
# Double Buffer

- Use two system buffers instead of one
- A process can transfer data to or from one buffer while the operating system empties or fills the other buffer



# Circular Buffer

- More than two buffers are used
- Each individual buffer is one unit in a circular buffer
- Used when I/O operation must keep up with process

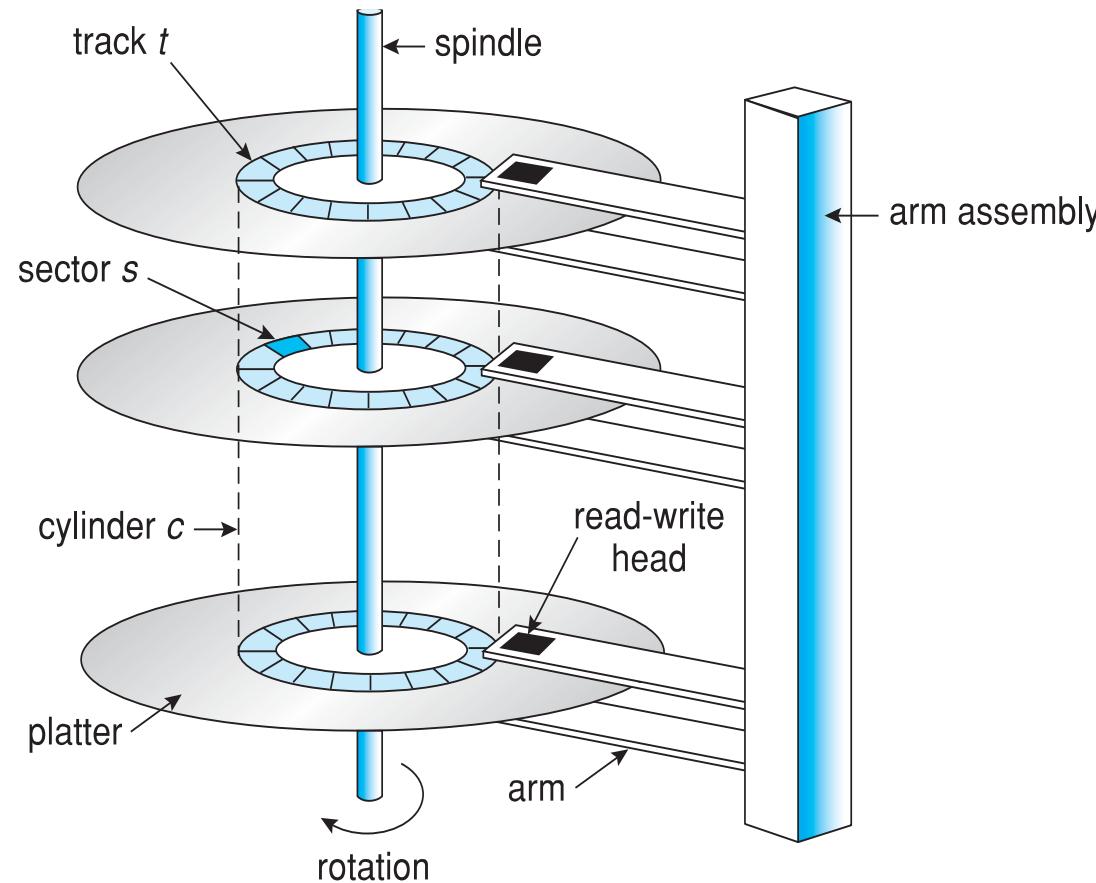


# Mass Storage Management

# Overview of Mass Storage Structure

- Magnetic disks provide bulk of secondary storage of modern computers
  - Drives rotate at 60 to 250 times per second
- Disks can be removable
- Drive attached to computer via I/O bus
  - Busses vary, including EIDE, ATA, SATA, USB, Fibre Channel, SCSI, SAS, Firewire
  - Host controller in computer uses bus to talk to disk controller built into drive or storage array

# Moving-head Disk Mechanism



# Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
  - Sector 0 is the first sector of the first track on the outermost cylinder.
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

The read/write data is a three-stage process:

- Position the head/arm over the proper track (into proper cylinder). The time required to move the heads from one cylinder to another, and for the heads to settle down after the move is called as Seek time/Positioning time (random-access time).
- Wait for the desired sector to rotate under the read/write head. The amount of time required for the desired sector to rotate around and come under the read-write head is referred as Rotational latency.
- Transfer a block of bits (sector) under the read-write head. The time required to move the data electronically from the disk to the computer is called Transfer time.

The disk access time is computed as follows:

Seek Time:  $T_s = m * n + s$

where  $n$  is the number of tracks traversed,  $m$  is the track traversal time and  $s$  is the startup time

Rotational Latency:  $T_r = 1 / (2 * r)$

where  $r$  is the number of revolutions per time unit

Transfer Time:  $T_t = b / (r * N)$

where  $b$  is the number of bytes to be transferred and  $N$  is the number of bytes on track

Disk Access Time:  $T = T_s + T_r + T_t$

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.

- **Magnetic Tape Storage:**

It is sequential access storage and it is both persistent and rewritable. This device is not suitable for transaction processing where random (direct) access is used for accessing data. Also the access time is very slow. These are used mostly for back up purposes.

- **Disk Storage:**

These are random access devices and the storage capacity varies from megabytes to terabytes. These devices exhibit variable access speed that depends on the relative positions of the read-write head and the requested data. Some examples of disks are magnetic disks, magneto-optical disks, floppies and CDs.

# Disk Attachments

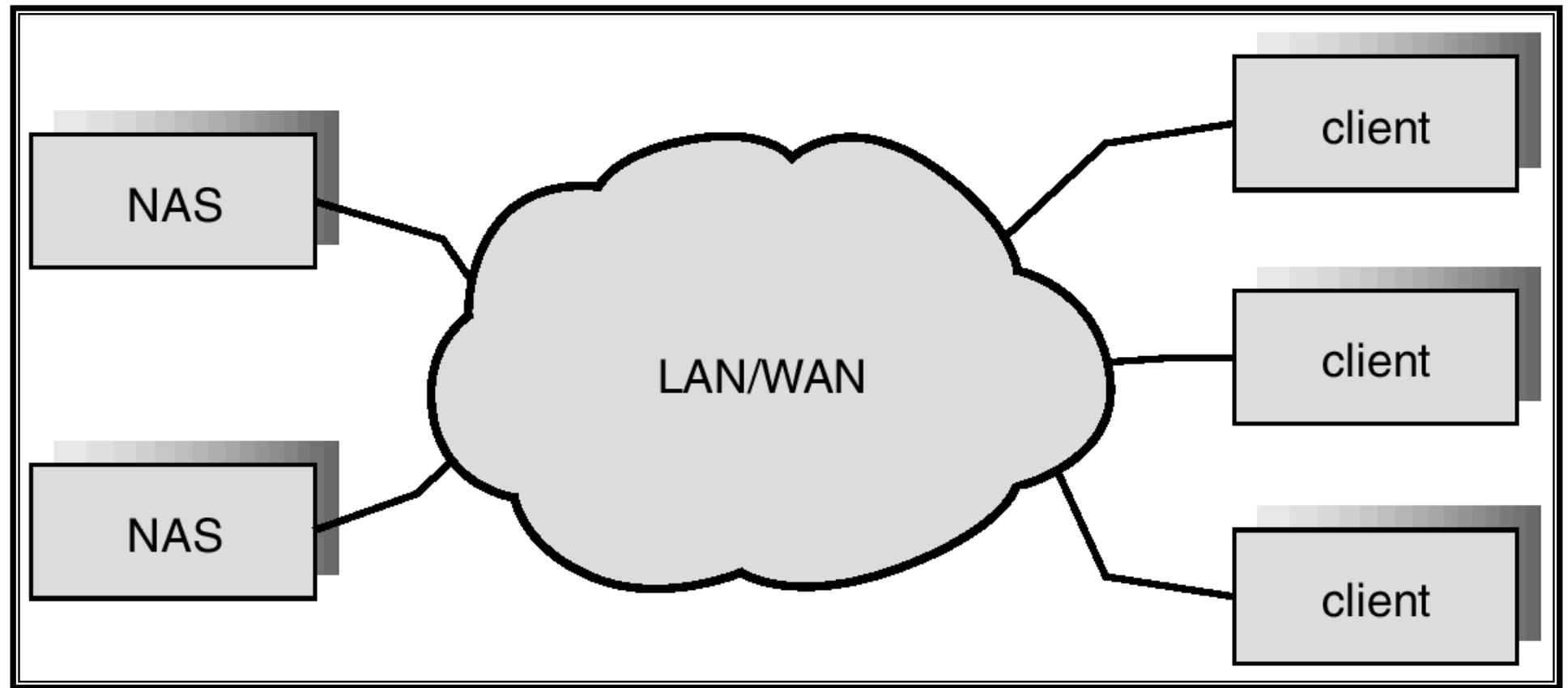
Disk drives can be attached either directly to a particular host machine or to a network.

## **Host-Attached Storage**

The storage is referred as Local disk and accessed through I/O Ports. The most common interfaces are IDE or ATA, each of which allow up to two drives per host controller.

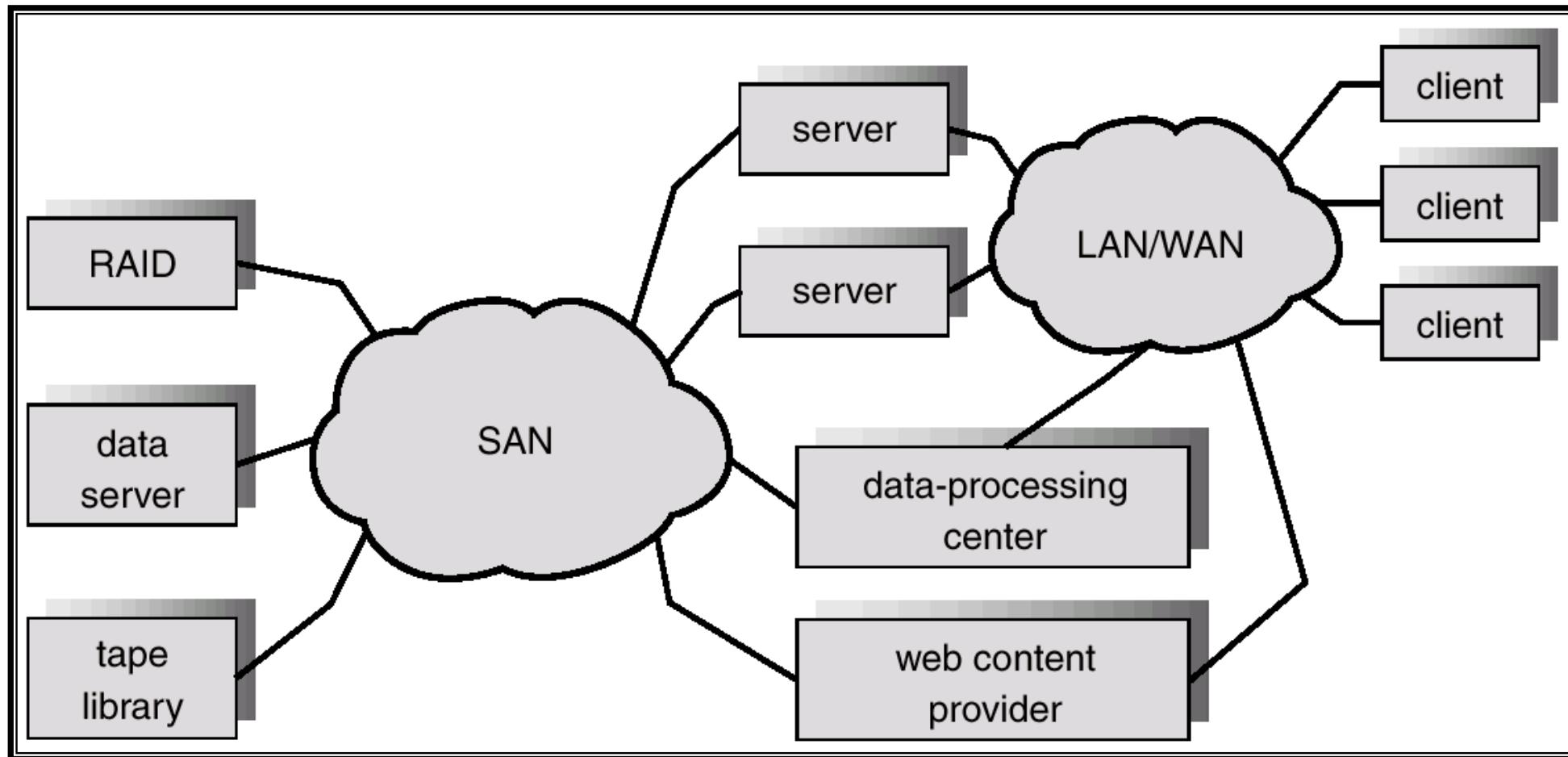
## **Network-Attached Storage**

The storage devices are part of a LAN/WAN and are accessed by the clients using Remote Procedure Call (RPC) typically with NFS file system mounting. These devices form a shared storage by using naming conventions and allow group access by the clients.



## Storage-Area Network

- A **Storage-Area Network (SAN)** connects computers and storage devices in a network, using storage protocols instead of network protocols.
- It is very flexible and dynamic, allowing hosts and devices to attach and detach on the fly and is also controllable allowing restricted access to certain hosts and devices.

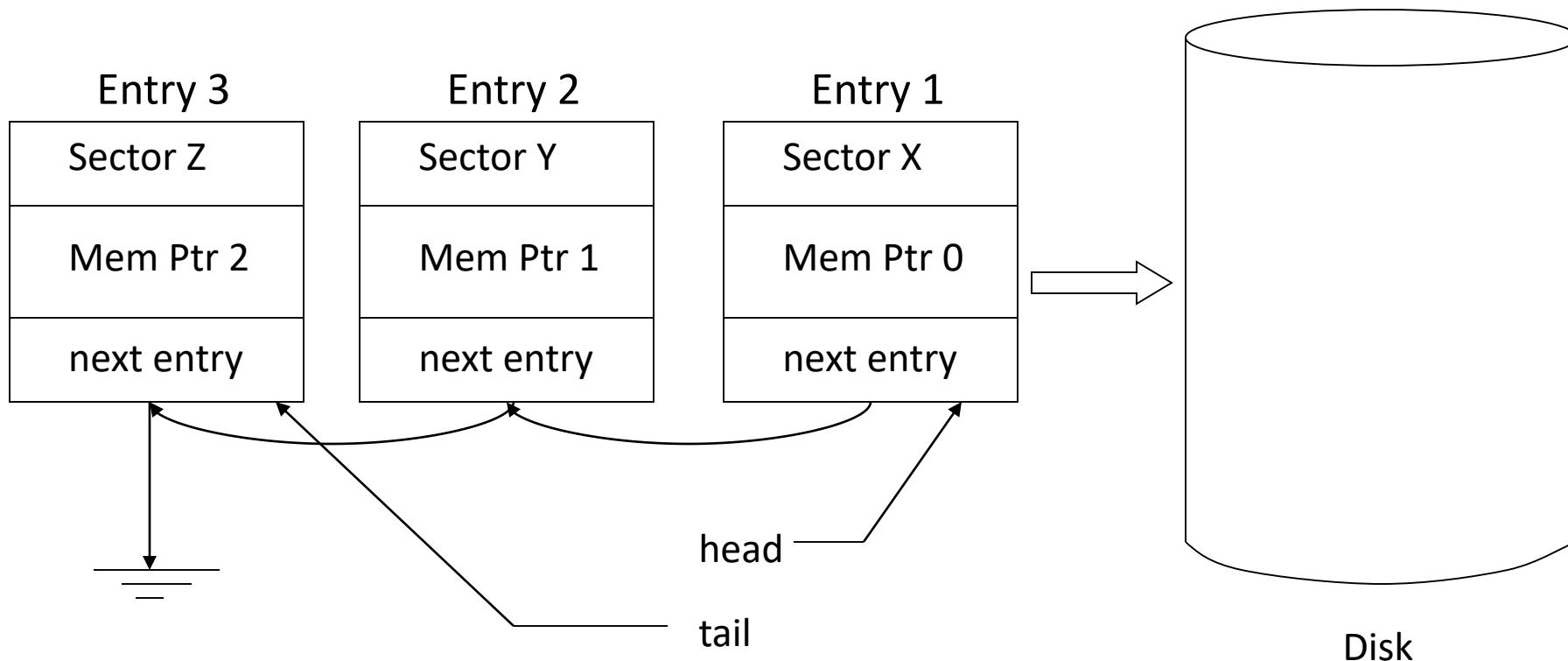


# Disk Scheduling

# Disk Queues

- Each disk has a queue of jobs waiting to access disk
  - read jobs
  - write jobs
- Each entry in queue contains the following
  - pointer to memory location to read/write from/to
  - sector number to access
  - pointer to next job in the queue
- OS usually maintains this queue

# Disk Queues



# Disk Scheduling

- Several algorithms exist to schedule the servicing of disk I/O requests.

Example

- A request queue is (0-199).

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

Head pointer 53

# FCFS

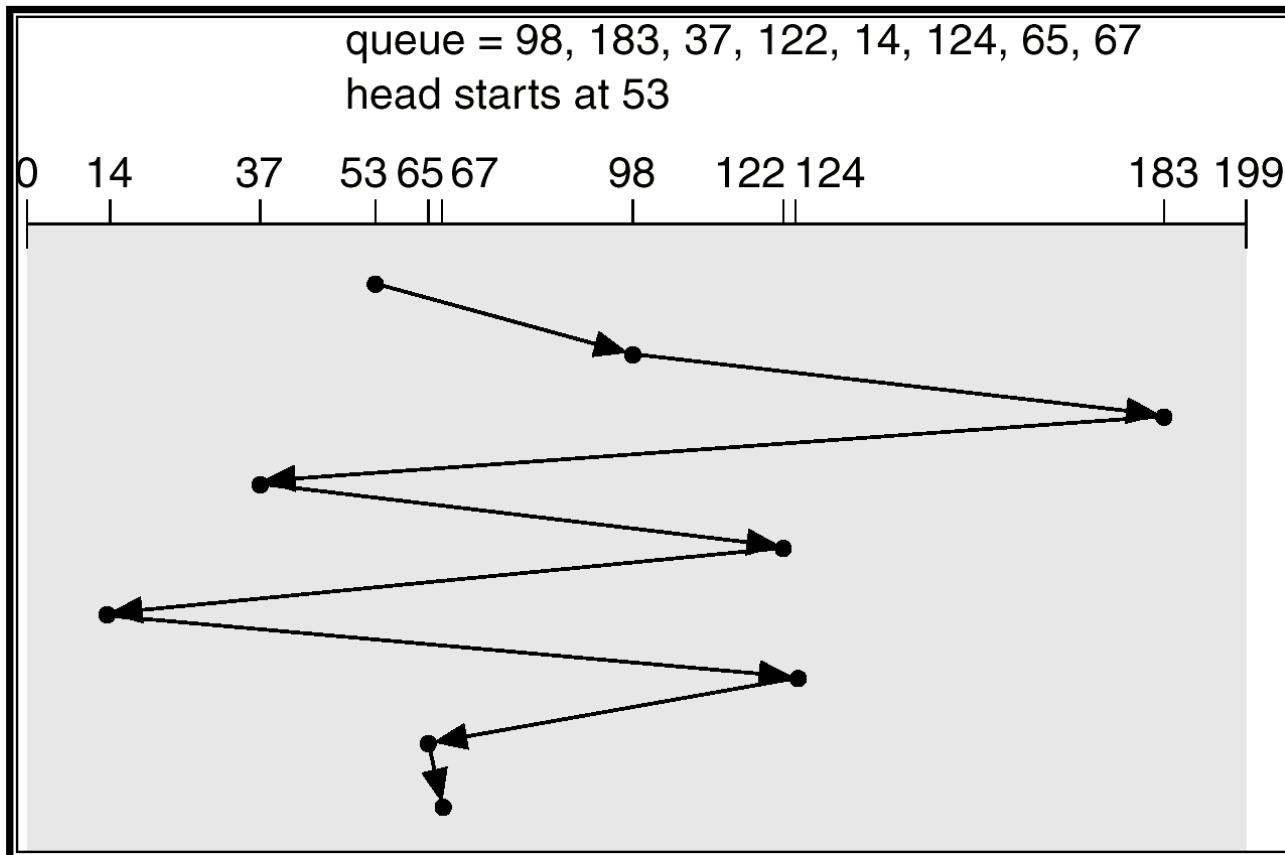
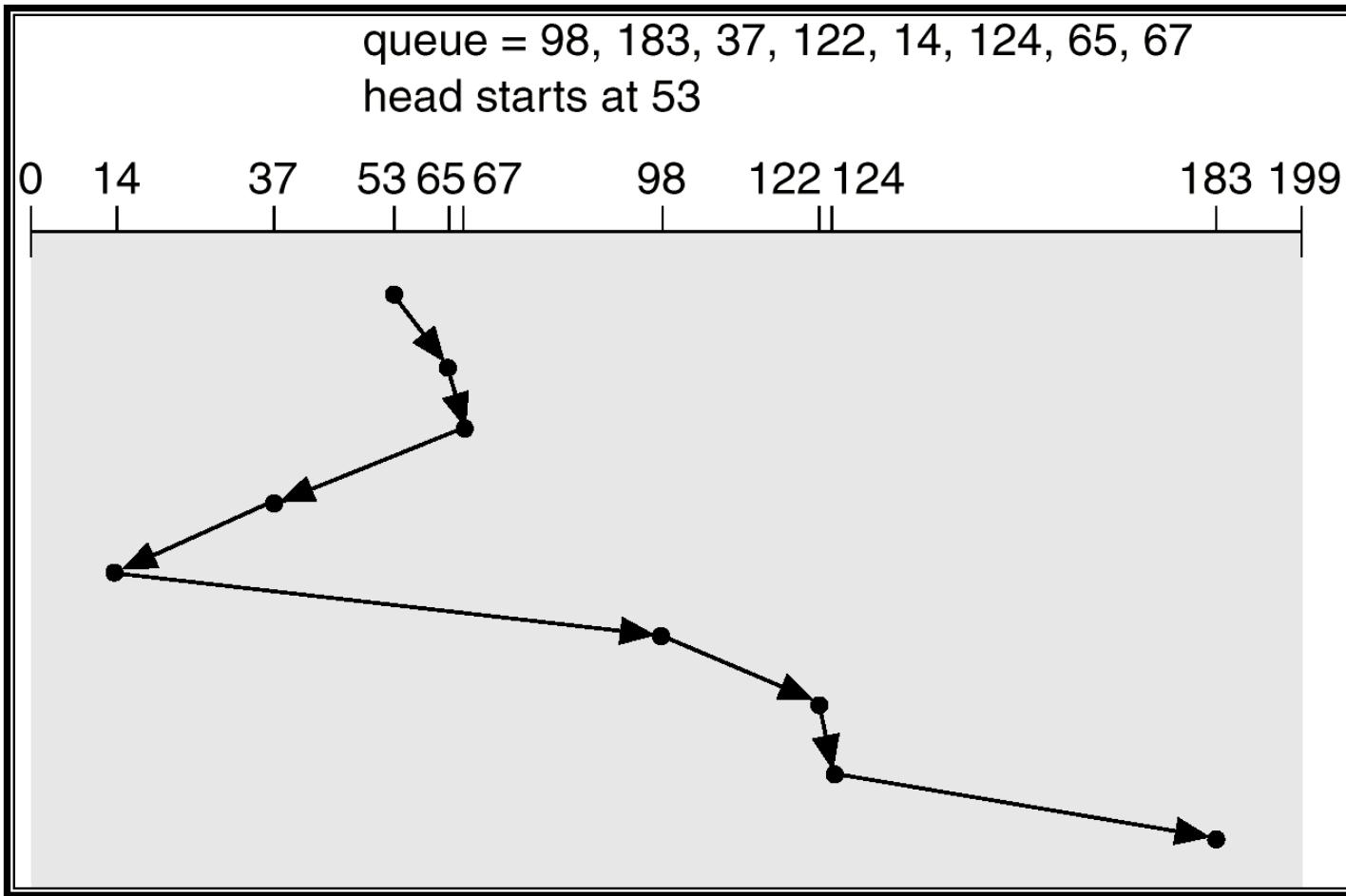


Illustration shows total head movement of 640 cylinders.

# SSTF

- Selects the request with the minimum seek time from the current head position.
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.
- total head movement of 236 cylinders.

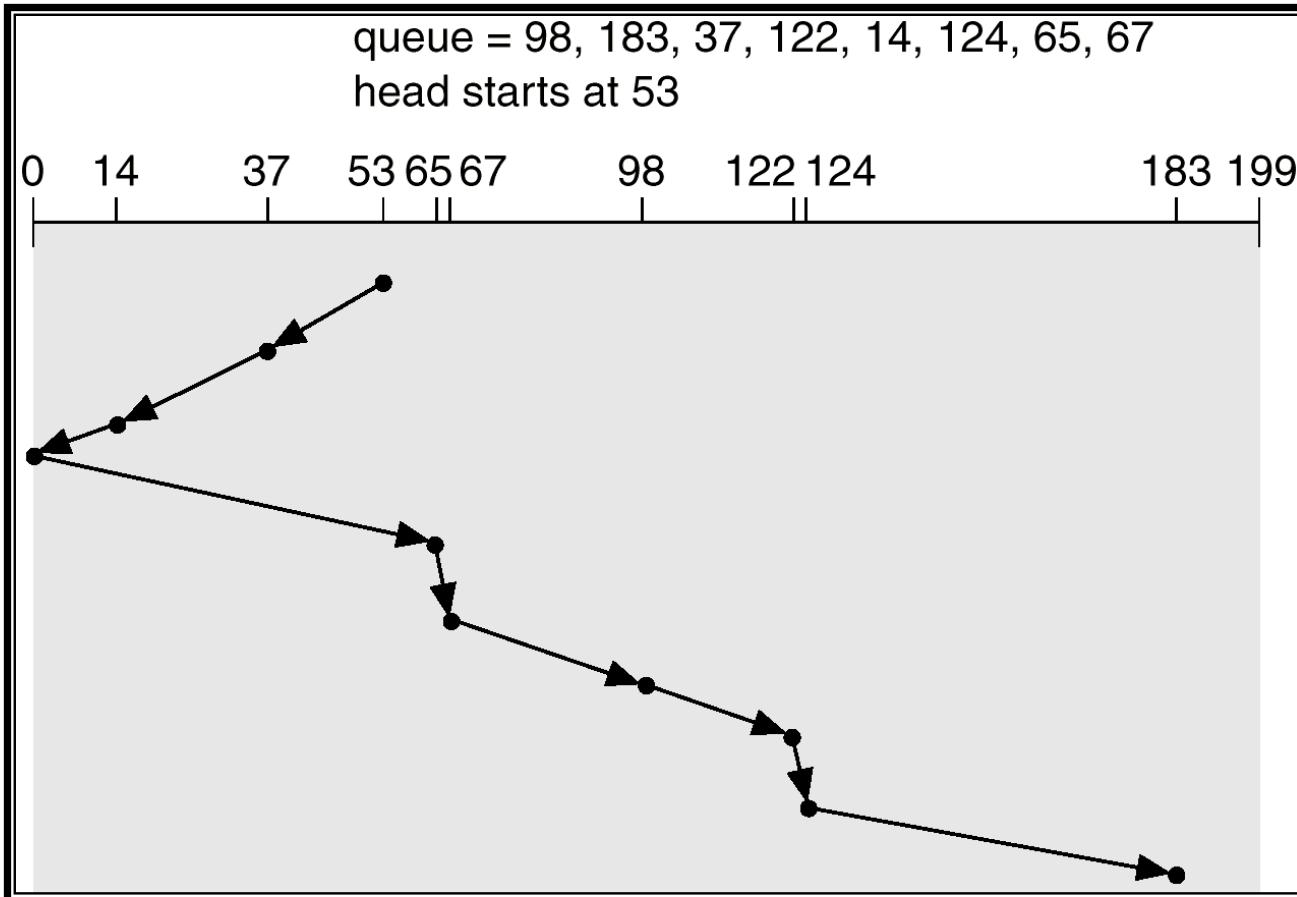
# SSTF



# SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Sometimes called the *elevator algorithm*.
- total head movement of 208 cylinders.

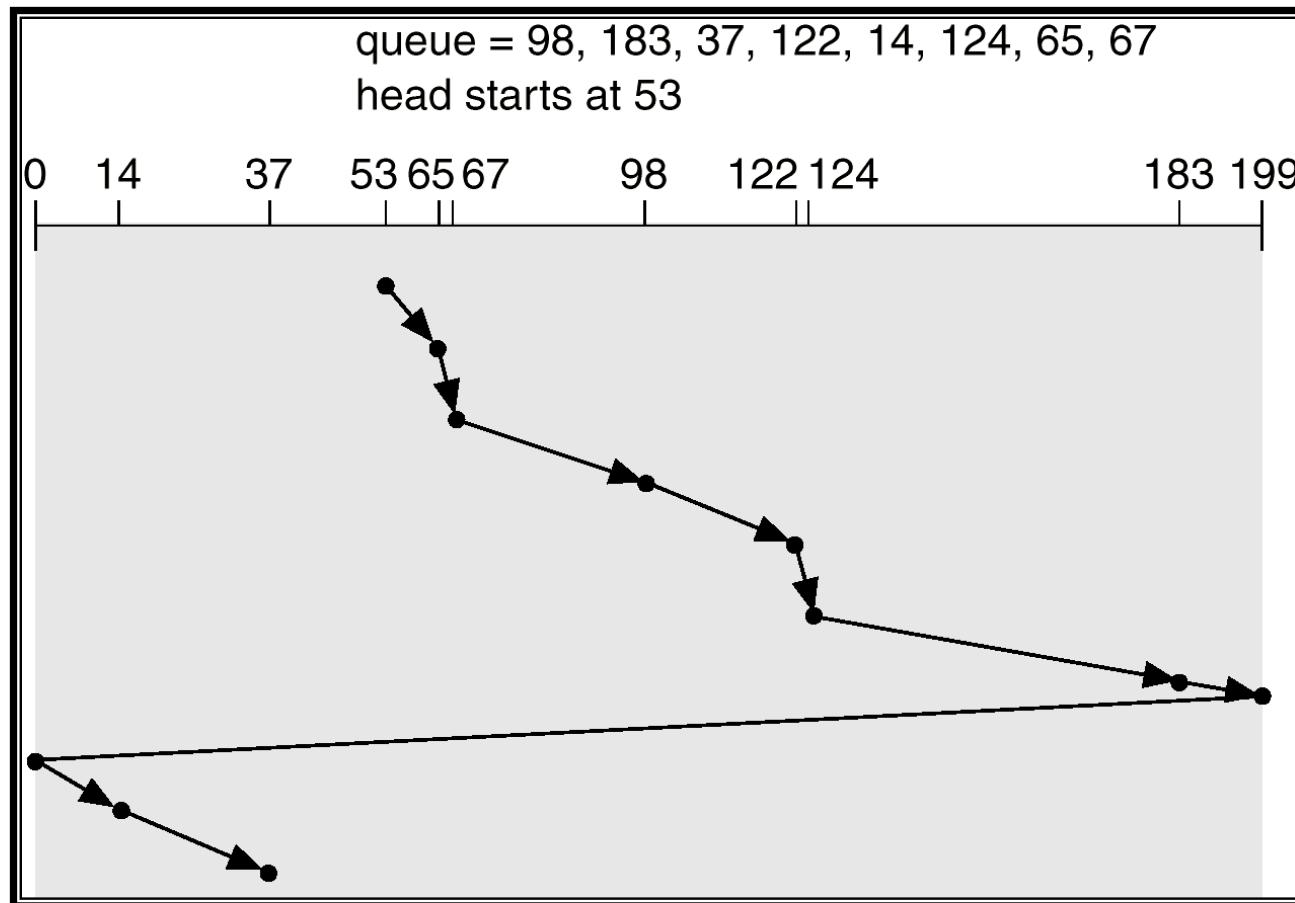
# SCAN



# Circular SCAN (C-SCAN)

- Provides a more uniform wait time than SCAN.
- The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.

# C-SCAN



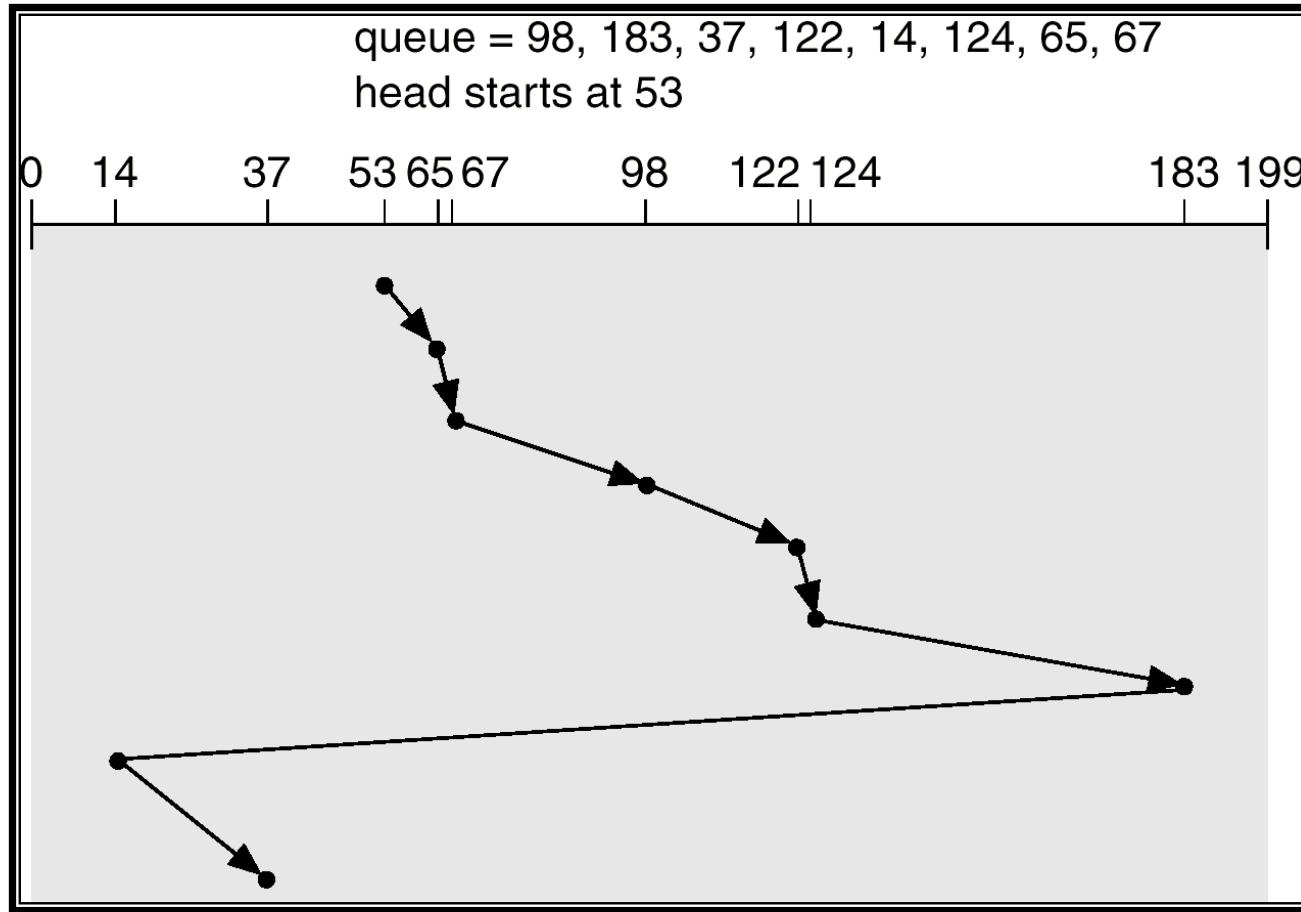
# LOOK

- The head moves in one direction and satisfies the request in that direction, if there is no request in that direction, it reverses its direction and serves request

# C-LOOK

- Version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.

# C-LOOK



# Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or LOOK is a reasonable choice for the default algorithm.

# Redundant Array of Inexpensive/Independent Disks (RAID)

- Secondary storage devices are slow and to improve their performance multiple devices in parallel such as arrays of disks are used.
- RAID is a group of hard drives together with some form of redundancy is employed to improve the performance and to provide reliability.
- Single large capacity disk is replaced with array of smaller capacity disks.
- Earlier small cheaper disks were used so it was inexpensive but now large expensive disks are used so it is independent.

Common characteristics:

- Array of physical disks are visible as single device to OS.
- Data is distributed across physical drives of array.
- Redundant disk capacity is used for error detection/correction.

Benefits:

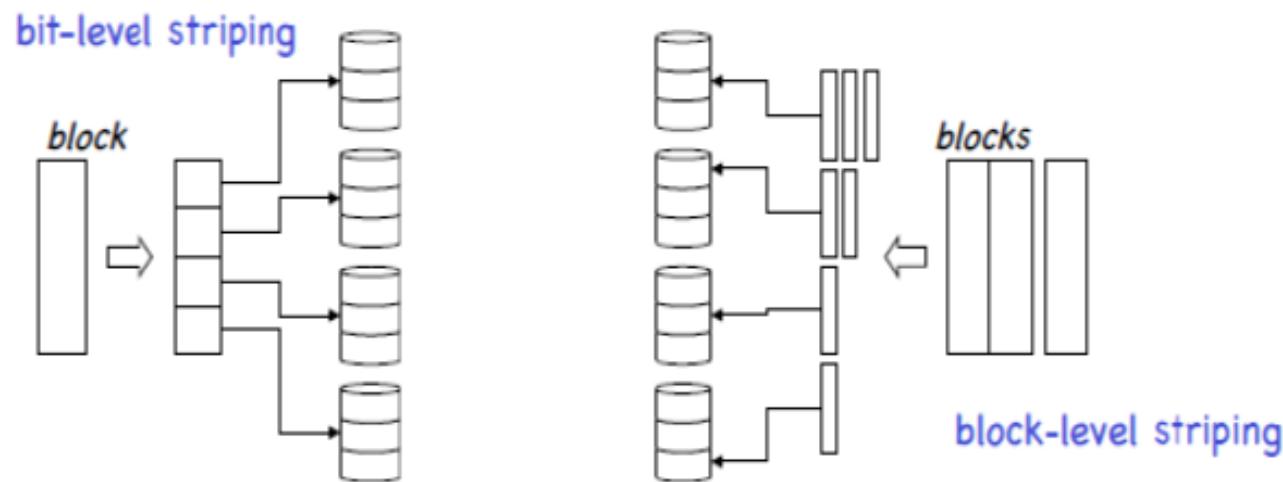
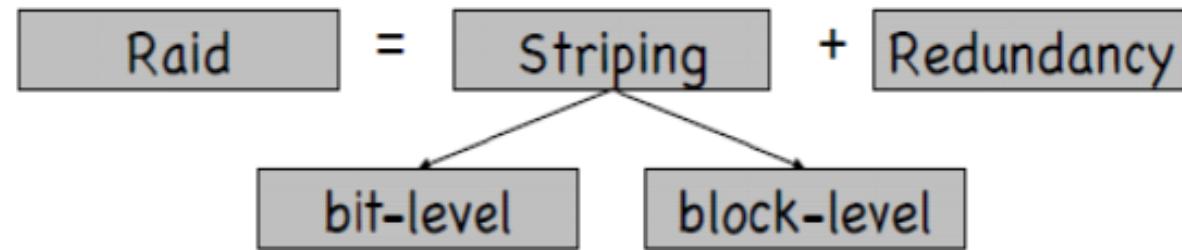
- Improved I/O performance; Enables incremental upgrade

Problem:

- Reliability is achieved through more devices that increase the probability of failure and the solution is redundancy.

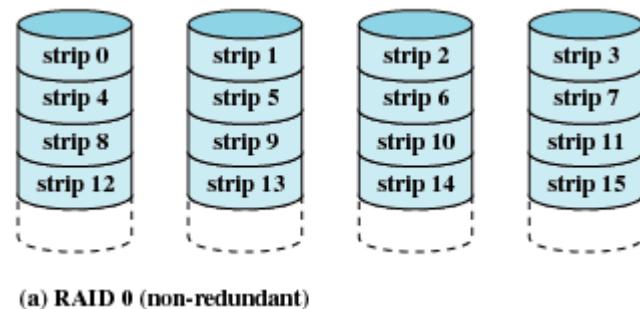
- RAID is arranged into six different levels.
- Several improvements in disk-use techniques involve the use of multiple disks working cooperatively.
- Disk striping uses a group of disks as one storage unit.
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data.
  - *Mirroring* or *shadowing* keeps duplicate of each disk.
  - *Block interleaved parity* uses much less redundancy.

# Bit level and block level striping



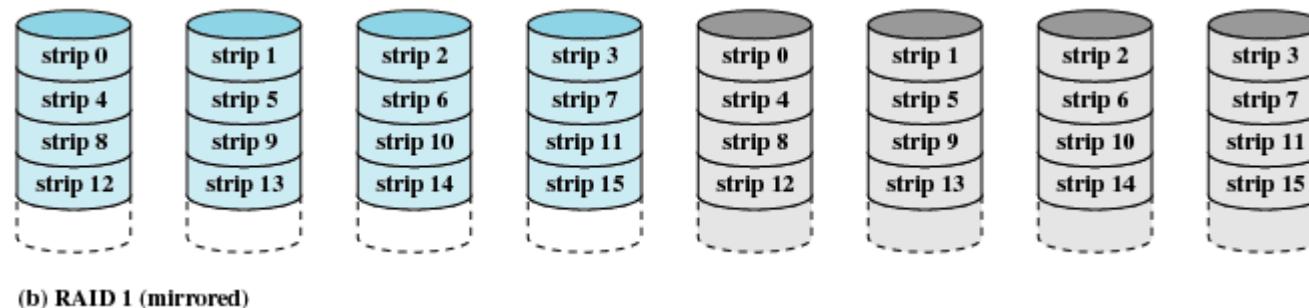
# RAID 0 (non-redundant)

- Block level striped set without Parity
- RAID 0 offers no redundancy, but improves disk access
- Here, files are broken into strips and distributed across disk surfaces (known as disk spanning) so that access to a single file can be done in parallel disk accesses



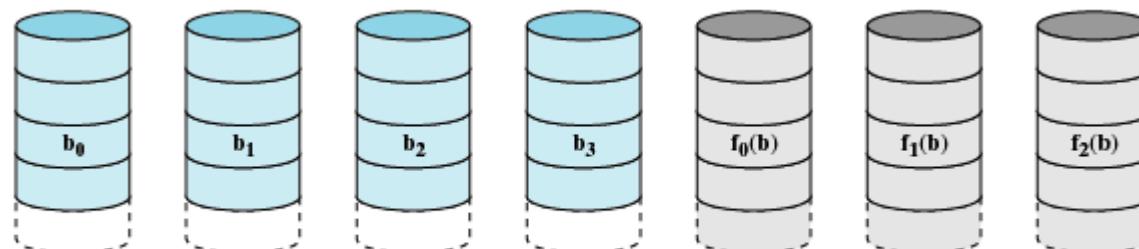
# RAID 1 (mirrored)

- Mirrored set without parity
- 100 percent redundancy leads to increase in cost.
- Read operation is done with multithreading and split reads. There is small penalty in write operation because of redundancy (writes require saving to both sets of disk).



# RAID 2 (redundancy through Hamming code)

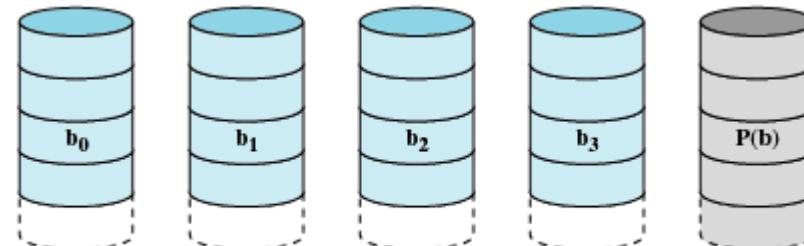
- Memory style error correcting parity
- Head and spindles are synchronized.
- Strips are small.
- It strips each byte into 1 bit per disk and uses additional disks to store Hamming codes for redundancy
- Error correction codes are corrected over bits of data disks.
- It is suitable for system with many failures.



(c) RAID 2 (redundancy through Hamming code)

# RAID 3 (bit-interleaved parity)

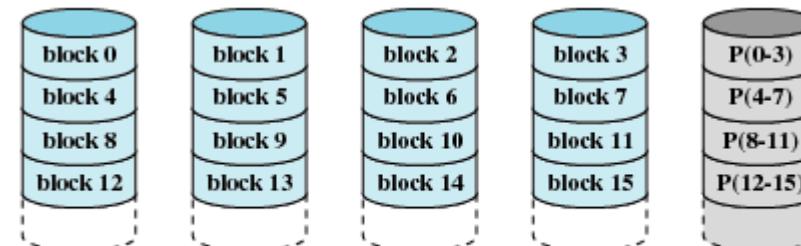
- Bit interleaved Parity Head and spindles are synchronized.
- Strips are small.
- Simple parity bits are used instead of Error correcting codes.
- most suitable for small computer systems that require some but not total redundancy



(d) RAID 3 (bit-interleaved parity)

# RAID 4 (block-level parity)

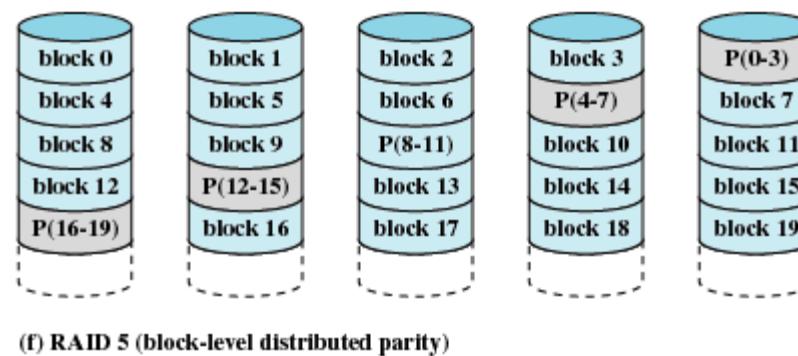
- Block Level Parity ; Same as RAID 3 but with block level striping
- Disks are not synchronized
- Strips are large and each strip contains parity information of all corresponding strips
- All parity information is placed on a single disk which creates a bottleneck and so defeats the advantage of parallel accesses



(e) RAID 4 (block-level parity)

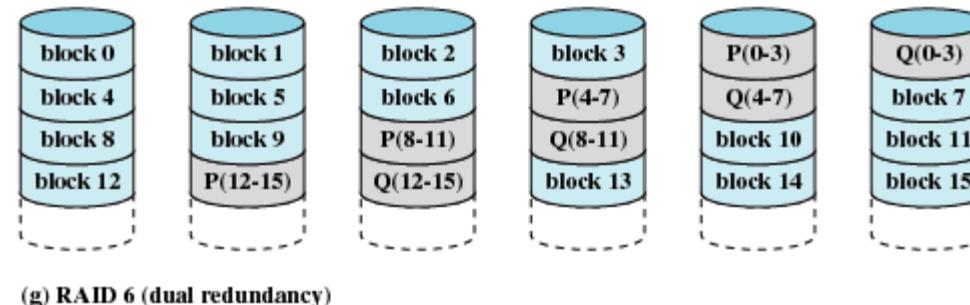
# RAID 5 (block-level distributed parity)

- Striped Set with Interleaved parity
- Same as RAID 4 but parity spread across all disks
- Disks are not synchronized
- Strips are large strips



# RAID 6 (dual redundancy)

- Striped Set with Dual Interleaved Parity
- Same as RAID 5 but uses 2 bits for storing parity
- Disks are not synchronized
- Strips are large
- ECC is used instead of parity
- Tolerates two failures



# Protection and Security

- The whole world is interconnected with the advent of internet technology.
- It has taken human life into much higher levels of sophistication and ease. But the computer systems are faced with new threats through internet.
- So Protection and Security of computer systems are becoming important as the world is closely interconnected.
- Operating System plays a major role in providing protection of user data and system resources as it is responsible for serving computer users locally and remotely through the Internet.
- With respect to OS, protection may be viewed as its internal requirement whereas Security deals with protecting a complete system from threats arising from the external environment in which the computer resides.

# Protection

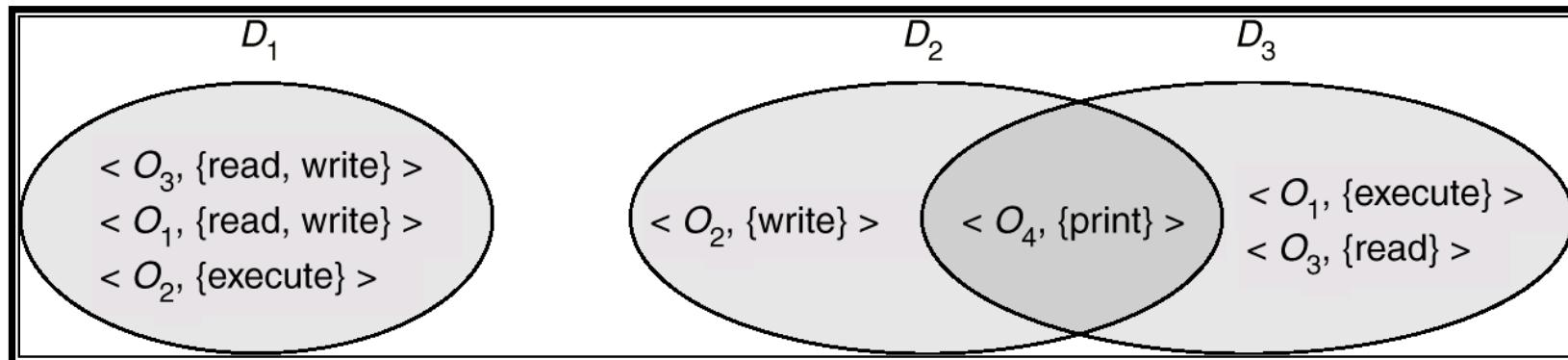
- What is Protection?
- The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use.
- Protection prevents accidental or intentional misuse of a system.
- Primarily the system must have a mechanism to identify users who are eligible to use the computer.
- It must have a means to authenticate eligible users who attempt to use the system.

# Protection Domain and its structure

- A computer can be viewed as a collection of processes and objects both hardware and software.
- A process can have access to those objects it needs to accomplish its task in the modes for which it needs access and during the time frame when it needs access.

# Domain Structure

- Access-right =  $\langle \text{object-name}, \text{rights-set} \rangle$   
where *rights-set* is a subset of all valid operations that can be performed on the object.
- Domain = set of access-rights



The three aspects to a protection mechanism are as follows:

- Authentication: identify a responsible party (principal) behind each action.
- Authorization: determine which principals are allowed to perform which actions.
- Access enforcement: combine authentication and authorization to control access.

# Authentication

- A secret piece of information used to establish identity of a user. Typically it is done with passwords and must not be stored in an encrypted form.
- The other form of authentication is using key which should not be forgeable or copyable. Once authentication is complete, the identity of the principal must be protected from tampering, since other parts of the system will rely on it.
- Once the log in process is over, the user id is associated with every process executed under that login: each process inherits the user id from its parent.

# Authorization

- Its goal is to determine which principals can perform which operations on which objects.
- Logically, authorization information is represented as an access matrix:
  - One row per principal.
  - One column per object.
- Each entry indicates what that principal can do to that object.

# Access Matrix

- View protection as a matrix (*access matrix*)
- Rows represent domains
- Columns represent objects
- $\text{Access}(i, j)$  is the set of operations that a process executing in Domain<sub>i</sub> can invoke on Object<sub>j</sub>

# Access Matrix

object domain	$F_1$	$F_2$	$F_3$	printer
$D_1$	read		read	
$D_2$				print
$D_3$		read	execute	
$D_4$	read write		read write	

# Use of Access Matrix

- If a process in Domain  $D_i$ , tries to do “op” on object  $O_j$ , then “op” must be in the access matrix.
- Can be expanded to dynamic protection.
  - Operations to add, delete access rights.
  - Special access rights:
    - *owner of  $O_i$* ,
    - *copy op from  $O_i$  to  $O_j$*
    - *control –  $D_i$  can modify  $D_j$  access rights*
    - *transfer – switch from domain  $D_i$  to  $D_j$*

# Use of Access Matrix (Cont.)

- Access matrix design separates mechanism from policy.
  - Mechanism
    - Operating system provides access-matrix + rules.
    - It ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced.
  - Policy
    - User dictates policy.
    - Who can access what object and in what mode.

# Implementation of Access Matrix

The access matrix is usually large and sparse. Different ways of implementing access matrix are:

- store the matrix by columns or by rows
- store only the non-empty elements
- Storing the matrix by columns corresponding to access control lists
- Storing the matrix by rows corresponding to capabilities

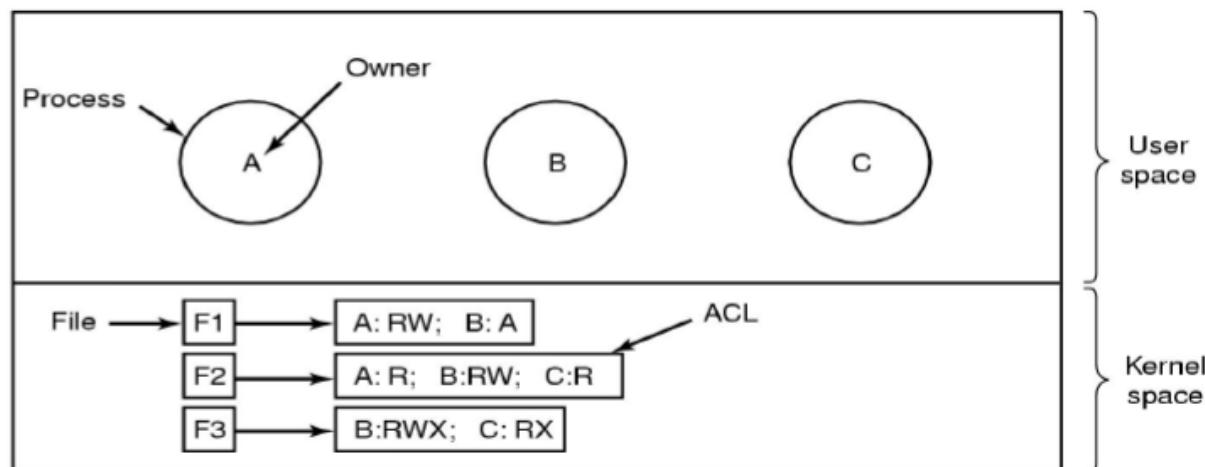
The simplest approach to implement access matrix is using one big global table or using linked lists with entries.

As the size of access matrix increases the global table/ list will be large and so it is stored in the following two compressed forms.

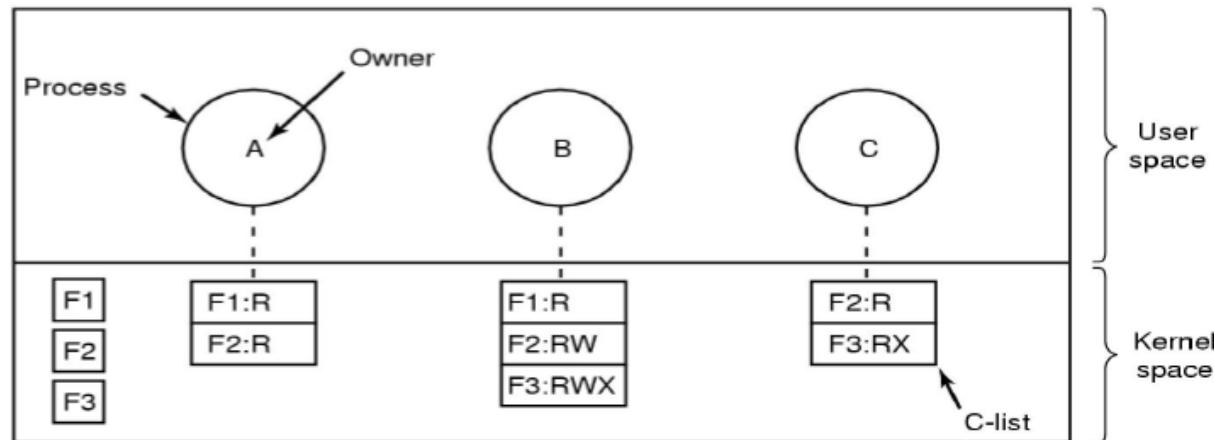
- Access control lists
- Capabilities

## Access control lists (ACLs):

- The information stored is about the users who are allowed to perform operations on each object. It is a list of < user, privilege> pairs
- The users may be single or group and the operations may be read, write or execute. ACLs are simple and are used in almost all file systems.



- Capabilities:
- They are organized by rows and have the information about each user and the objects that can have access by them and in what ways.
- Capabilities store a list of pairs with each user
- Capabilities have been used in experimental systems attempting to be very secure.



- Revocation of Access Rights:
- In a dynamic protection system, sometimes there is a need to revoke access rights to objects shared by different users.
- The revocation may be Immediate versus Delayed, Selective versus General, Partial versus Total, and Temporary versus Permanent.
- *Access List* – Delete access rights from access list.
  - Simple
  - Immediate

Schemes that implements revocation for capabilities include the following:

- Reacquisition: Periodically, capabilities are deleted from each domain.
- Back pointers: A list of pointers is maintained with each object, pointing to all capabilities associated with that object.
- Indirection: The capabilities point indirectly, not directly, to the objects.
- Keys: A key is a unique bit pattern that can be associated with a capability.

# Security

## What is security?

- It is the quality or state of being secure i.e. Protecting the system from its users and preventing the unauthorized disclosure or modification of data.

## Need for security

- Security is needed to prevent illegal users from using the system, to make unauthorized actions impossible, to confine errors to the immediate contexts of their occurrences and to detect and correct damage before it spreads to other parts of the system.

Computer security is about making its software behave appropriately.

Three factors that make this task difficult are

- Complexity: Software systems acquire complexity easily as software with more lines may contain more bugs that may lead to undesirable behaviour of the system.
- Extensibility: Most modern computing systems are extendable and evolve over time. Most software accept updates or extensions from remote hosts mainly through mobile codes which act as a carrier for attacks.
- Connectivity: The growing trend of Internet connectivity renders computers vulnerable to remote attacks.

# Categories of security

- Based on Usage of software
  - Software security is related to its design construction and testing.
  - Application security is concerned with protecting the software during operation.
- Based on the perspective of the system
  - Internal security
    - Ensuring security within the system
    - Deals with the security issues related to the behavior of the active entities within the system.
  - Boundary Security
    - Ensuring security along the boundary of the system
    - Deals who can enter the system from outside

- Based on Undesirable behaviours
  - Malicious behaviour attempts to read or destroy sensitive data or disrupt the system operation intentionally, usually initiated from outside.
  - Incidental behaviours need not be intentional and usually arise from within the system but the consequences are as serious as of malicious behaviour and may be due to hardware malfunction or undetected errors from software or natural disaster damage etc.

- Based on resources
  - Hardware security – Security attacks having resources such as processors, main memory, secondary storage devices, communication lines and devices.
  - Software security- Data and service (process) are subject to security risk. Confidentiality and integrity of data may be compromised. The service/process may behave in certain way, proliferated or destroyed.

- Based on User's point of view
  - Unauthorized use- Gaining access to the system or account using another person's system or pirated version of software
  - Denial of Service- Intentional prevention of authorized users obtaining their services on time.

- Based on source of attack
  - Attacker within system may be an unauthorized user or a malicious process
  - Attacker may be from outside through internet using legitimate channels or through illegitimate channels.

- A secure system should not allow:
  - unauthorized reading of information
  - unauthorized modification of information
  - unauthorized destruction of data
  - unauthorized use of resources
  - denial of service for authorized uses

- The key aspect of security is identifying the user to the system. Authentication is establishing identity and authorization is establishing rights of an identity.
- Authentication is based on one or more of three items:
  - user possession (a key or card),
  - user knowledge (a user identifier and password), and
  - a user attribute (fingerprint, retina pattern, or signature).

# Authentication

- User identity most often established through *passwords*, can be considered a special case of either keys or capabilities.
- Passwords must be kept secret.
  - Frequent change of passwords.
  - Use of “non-guessable” passwords.
  - Log all invalid access attempts.
- Passwords may also either be encrypted or allowed to be used only once.

# Threats - Program Threats

- In an environment where a program written by one user may be used by another user, there is an opportunity for misuse, which may result in unexpected behavior.

Common methods

Trojan horses

Trap doors

Stack and Buffer Overflow

## Trojan Horse:

- Code segment that misuses its environment.
- Exploits mechanisms for allowing programs written by users to be executed by other users.

The Trojan-horse problem is exacerbated by long search paths. The search path lists the set of directories to search when an ambiguous program name is given. The path is searched for a file of that name and the file is executed. All the directories in the search path must be secure, or a Trojan horse could be slipped into the user's path and executed accidentally.

## Trap Door:

- The designer of a program or system might leave a hole in the software that only Operating System is capable of using.
- Specific user identifier or password that circumvents normal security procedures.
- A clever trap door could be included in a compiler. The compiler could generate standard object code as well as a trap door, regardless of the source code being compiled.

## Stack and Buffer Overflow

- Exploits a bug in a program (overflow either the stack or memory buffers.)

# System Threats

- Worms – use spawn mechanism; standalone program
- Internet worm
  - Exploited UNIX networking features (remote access) and bugs in *finger* and *sendmail* programs.
  - Grappling hook program uploaded main worm program.
- Viruses – fragment of code embedded in a legitimate program.
- Denial of Service
  - Overload the targeted computer preventing it from doing any useful work.

- Worms
- A worm is a process that uses the spawn mechanism that repeatedly reduce system performance.
- The worm spawns copies of itself, using up system resources and may lock systems resources to all other processes. Since it may reproduce itself among systems and thus shut down the entire network.
- The worm was made up of two programs a grappling hook (also called bootstrap or vector) program and the main program.
- The grappling hook consists of 99 lines of C code compiled and run on each machine it accessed. The grappling hook is connected to the machine where it originates and upload a copy of the main worm onto the "hooked" system.
- The main program proceeded to search for other machines to which the newly infected system could connect easily.

## Viruses

- Another form of computer attack is a virus. Like worms, viruses are designed to spread into other programs and can completely destroy a system, i.e. modifying or destroying files, causing system crashes and program malfunctions.
- A virus is a fragment of code embedded in a legitimate program whereas a worm is structured as a complete, standalone program.
- Viruses are a major problem especially for single user systems. They are usually spread by users downloading viral programs from public bulletin boards or using secondary devices such as floppy disks, pen drives, etc. containing an infection.
- The best protection against computer viruses is prevention, or the practice of Safe Computing.

# Threat Monitoring

- Check for suspicious patterns of activity – i.e., several incorrect password attempts may signal password guessing.
- Audit log – records the time, user, and type of all accesses to an object; useful for recovery from a violation and developing better security measures.
- Scan the system periodically for security holes; done when the computer is relatively unused.

# Threat Monitoring

- Check for:
  - Short or easy-to-guess passwords
  - Unauthorized set-uid programs
  - Unauthorized programs in system directories
  - Unexpected long-running processes
  - Improper directory protections
  - Improper protections on system data files
  - Dangerous entries in the program search path (Trojan horse)
  - Changes to system programs: monitor checksum values

# FireWall

- A firewall is placed between trusted and untrusted hosts.
- The firewall limits network access between these two security domains.

# Intrusion Detection

- Detect attempts to intrude into computer systems.
- Detection methods:
  - Auditing and logging.
  - Tripwire (UNIX software that checks if certain files and directories have been altered – i.e. password files)
- System call monitoring

# Encryption

- Encrypt clear text into cipher text.
- Properties of good encryption technique:
  - Relatively simple for authorized users to encrypt and decrypt data.
  - Encryption scheme depends not on the secrecy of the algorithm but on a parameter of the algorithm called the encryption key.
  - Extremely difficult for an intruder to determine the encryption key.
- *Data Encryption Standard* substitutes characters and rearranges their order on the basis of an encryption key provided to authorized users via a secure mechanism.

# Encryption

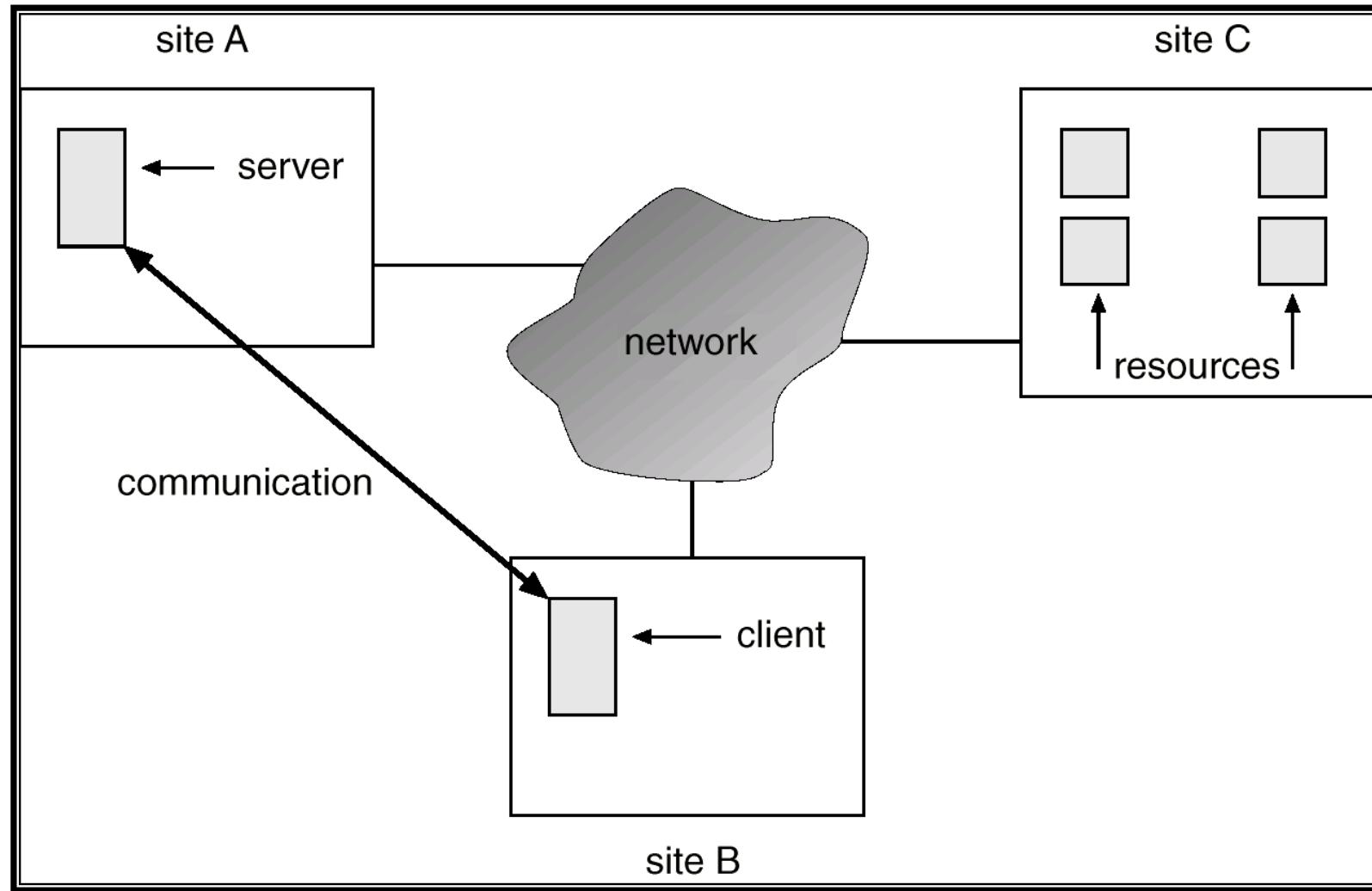
- Public-key encryption based on each user having two keys:
  - public key – published key used to encrypt data.
  - private key – key known only to individual user used to decrypt data.
- Must be an encryption scheme that can be made public without making it easy to figure out the decryption scheme.
  - Efficient algorithm for testing whether or not a number is prime.
  - No efficient algorithm is known for finding the prime factors of a number.

# Distributed Systems

# Definition of a Distributed System

- A distributed system is a collection of independent computers that appear to the users of the system as a single computer
- Two aspects:
  - Hardware: autonomous machines
  - Software: the users think of the system as a single computer

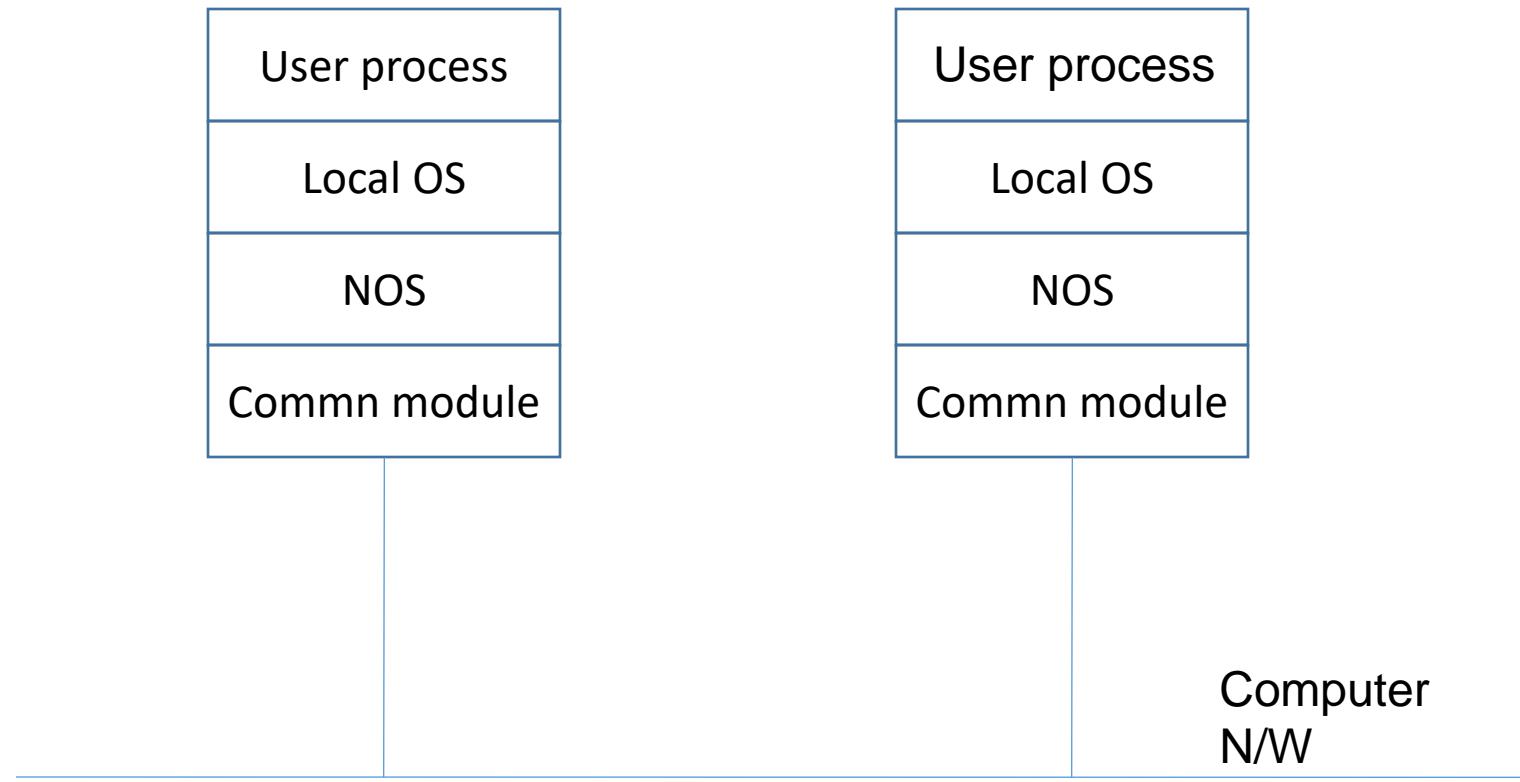
# A Distributed System



# Motivation

- Resource sharing
  - sharing and printing files at remote sites
  - processing information in a distributed database
  - using remote specialized hardware devices
- Computation speedup – *load sharing*
- Reliability – detect and recover from site failure, function transfer, reintegrate failed site
- Communication – message passing

# Network OS



# Network OS

- Users are aware of multiplicity of machines. Access to resources of various machines is done explicitly by:
  - Remote logging into the appropriate remote machine.
  - Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism.

## Characteristics

- Local OS
- Own computer or designated computer or remote login
- Explicit file transfer commands
- Little or no fault tolerance

Communication and information sharing

Shared Global File system

Use of file servers

Hierarchical file system

# Distributed OS

Users not aware of multiplicity of machines. Access to remote resources similar to access to local resources.

- Data Migration – transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task.
- Computation Migration – transfer the computation, rather than the data, across the system.

- Process Migration – execute an entire process, or parts of it, at different sites.
  - Load balancing – distribute processes across network to even the workload.
  - Computation speedup – subprocesses can run concurrently on different sites.
  - Hardware preference – process execution may require specialized processor.
  - Software preference – required software may be available at only a particular site.
  - Data access – run process remotely, rather than transfer all data locally.

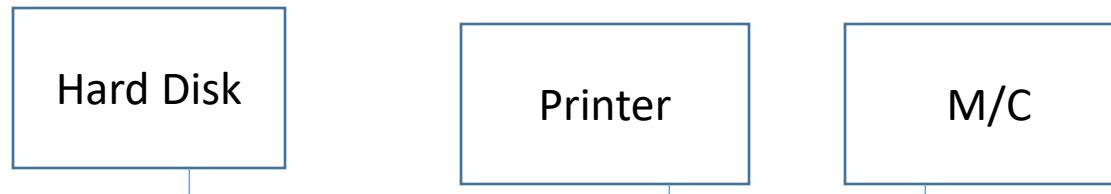
A DOS should

- Control n/w resource allocation
- Provide virtual computer
- Hide the distribution of resources
- Provide protection mechanisms
- Provide secure communication

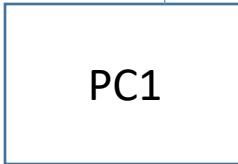
File Server

Print Server

Name Server



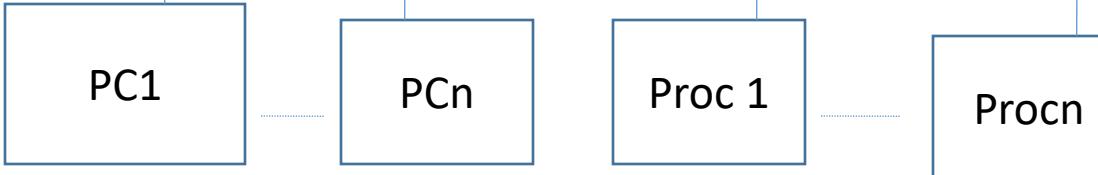
Communication Subsystem



User 1

User n

Processing servers



# Characteristics

- Global system wide OS
- Dynamic allocation of processes to CPU's
- File placement managed by OS
- Fault tolerance
- Single Global IPC mechanism

# Middleware based Distributed Systems

- Middleware- Additional layer on top of NOS implementing general purpose services and it hides more or less the heterogeneity of the collection of underlying platforms.

Middleware Services:

Communication facilities

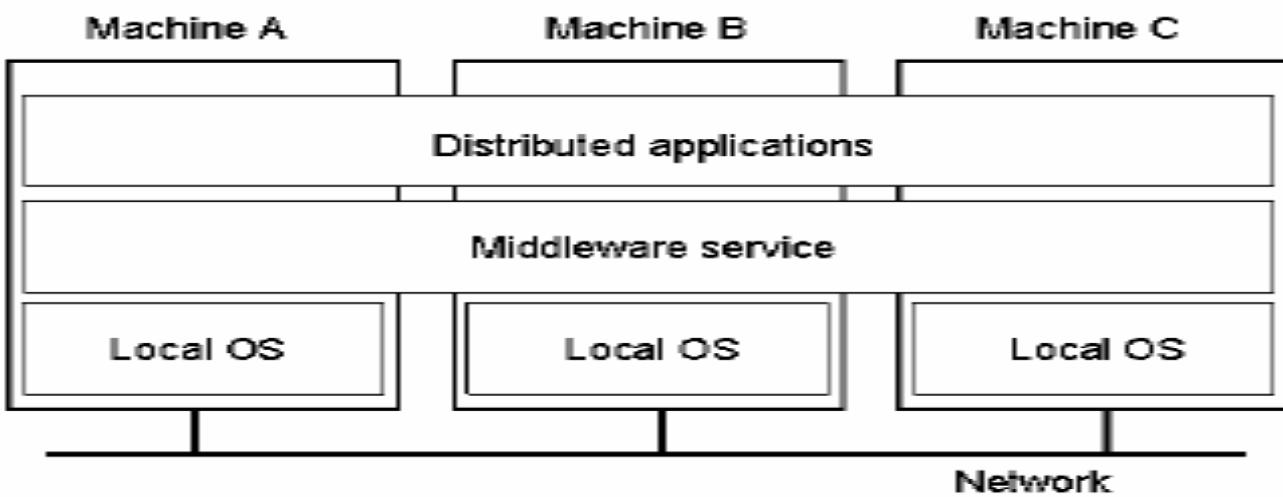
Naming

Persistence

Security

Openness

Scalability



A distributed system organized as middleware.

# Goals

- A distributed system should easily connect users to resources
- Why sharing?
- To make it easy for users to access remote resources, and to share them with other users in a controlled way
- Reason for sharing resources: economics
  - Typical resources : Printers, computers, storage facilities, data, files
    - Easier to collaborate and exchange information
      - Internet, groupware
- Problems with sharing
  - Security is becoming more and more important as connectivity and sharing increase
    - Tracking communication to build up a preference profile of a specific user
  - Unwanted Communication

# Goals

- It should hide the fact that resources are distributed across a network
- It should be open
- It should be scalable

# Design Issues

- Transparency – concealment from the user and the application programmer of the separation of components in a distributed systems so that the system is perceived as a whole rather than as a collection of independent components
- Flexibility- ease of modification and ease of enhancement
- Openness – The openness of a computer system is the characteristics that determines whether the system can be extended and re-implemented
- Scalability – A system is scalable, if it will remain effective when there is a significant increase in the numbers of resources and number of users
- Heterogeneity – Variety and differences
- Security- CIA
  - Confidentiality – protection against disclosure to unauthorized individuals
  - Integrity – Protection against alteration or corruption
  - Availability – Protection against interference with the means to access the resources.

# Transparency in a Distributed System

## Access transparency

- Hide differences in data representation and how a resource is accessed
- Intel (little endian format- *least* significant byte in the smallest address)/Sun SPARC (big endian- most significant byte in the smallest address) (order of bytes)
- OS with different file name conversions

# Transparency in a Distributed System

- **Location transparency**

Hide where a resource is located  
importance of naming, e.g., URLs

- **Migration transparency**

Hide that a resource may move to another location

- **Relocation transparency**

Hide that a resource may move to another location while in use  
example, mobile users

# Transparency in a Distributed System

- **Replication transparency**

Hide that a resource is replicated subsumes that all replicas have the same name (and thus location transparency)

- **Concurrency transparency**

Hide that a resource may be shared by several competitive users leave the resource in a consistent state more refined mechanism transactions

# Transparency in a Distributed System

- **Failure transparency**

Hide the failure and recovery of a resource

- **Persistent transparency**

Hide whether a (software) resource is in memory or Disk

Important problem: inability to distinguish between a dead resource  
and a painfully slow one

# *Design Issues: Transparency*

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource

# Openness

## Open distributed system

- Be able to interact with services from other open systems, irrespectively of the underlying environment
- Offers services according to standard rules that describe the **syntax and the semantics of these services**

## Rules formalized in protocols

- Services specified through interfaces (described in an Interface Definition Language (IDL) (but only the syntax part)
- Neutral and complete specifications (with regards to a potential implementation)

# Openness

- Interoperability: to what extend can work together
- Portability: to what extend an application developed for A can be executed on B that implements the same interface with A

# Openness

- Separate Policy from Mechanism
  - A system organized as a collection of relatively small and easily replaceable or adaptable components
  - Provide definitions of the internal parts of the system as well
  - A distributed system provides only **mechanisms.**

**Policies are specified by applications and users.**

Example policies:

- What level of consistency do we require for client-cached data?
- Which operations do we allow downloaded code to perform?
- Which QoS requirements do we adjust in the face of varying bandwidth?
- What level of secrecy do we require for communication?

# Scalability

- size (number of users and/or processes)
- geographical (maximum distance between nodes)
- administrative (number of administrative domains)
- Solution: Powerful server

# Scalability Problems

Centralized algorithms - Doing routing based on complete information

Centralized data - A single on-line telephone book

Centralized services - A single server

Major problem : Single point of failure.

Decentralized algorithms

- No complete information about the system state
- Make decision only on local information
- Failure of one machine does not ruin the algorithm
- No assumption of a global clock

# Scalability

- Synchronous communication
  - In WAN, Unreliable and point-to-point
- Geographical scalability
  - How to scale a distributed system across multiple, independent administrative domains:  
conflicting policies with respect to resource usage (and payment), management and security
- Expand to a new domain
  - Protect itself against malicious attacks from the new domain
  - The new domain has to protect itself against malicious attacks from the distributed system

# Scaling Techniques

- Hiding communication latencies
- Distribution
- Replication

# Scaling Techniques

## Hiding communication latencies:

- Try to avoid waiting for responses to remote service requests as much as possible
- asynchronous communication (do something else)
- moving part of the computation to the client process

# Scaling Techniques

## Distribution

- Taking a component, splitting into smaller parts, and spreading these parts across the system
- Example:
  - (1) The World Wide Web
  - (2) Domain Name Service (DNS)
- Hierarchically organized into a tree of domains
- Each domain divided into non overlapping zones
- The names in each domain handled by a single name server

# Replication Caching (client-driven)

- increase availability
- balance the load
- reduce communication latency
- but, consistency problems

# Flexibility

## Monolithic kernel

The collective kernel structure, Includes file system, directory, and process management

## Micro-kernel

- An interprocess communication mechanism
- Some memory management
- A small amount of low-level process management and scheduling
- Low level input/output
- File server
- Directory server
- Process server

# *Reliability / Dependability*

- Fault tolerance
  - Redundancy techniques
  - Distributed control
- Fault avoidance
  - Design of components
- Fault detection and recovery
  - Atomic transactions, stateless servers, ack and time out based retransmission.

- Performance
  - Response time, throughput, system utilization, bandwidth requirements
- Global knowledge
  - No shared memory, no global clock, how to find global state

# Distributed File System

# DISTRIBUTED FILE SYSTEMS

Clients, servers, and storage are dispersed across machines. Configuration and implementation may vary –

- a) Servers may run on dedicated machines, OR
- b) Servers and clients can be on the same machines.
- c) The OS itself can be distributed (with the file system a part of that distribution).
- d) A distribution layer can be interposed between a conventional OS and the file system.

Clients should view a DFS the same way they would a centralized FS; the distribution is hidden at a lower level.

Performance is concerned with throughput and response time.

# Goals

- 1 **Network transparency:** users do not have to aware the location of files to access them
  - **location transparency:** the name of a file does not reveal any kind of the file's physical storage location.
    - /server1/dir1/dir2/X
    - server1 can be moved anywhere
  - **location independence:** the name of a file does not need to be changed when the file's physical storage location changes.
    - The above file X cannot moved to server2 if server1 is full and server2 is no so full.
- 2 **High availability:** system failures or scheduled activities such as backups, addition of nodes

# Architecture

- Computation model
  - file servers -- machines dedicated to storing files and performing storage and retrieval operations (for high performance)
  - clients -- machines used for computational activities may have a local disk for caching remote files
- Two most important services
  - name server -- maps user specified names to stored objects, files and directories
  - cache manager -- to reduce network delay, disk delay problem: inconsistency
- Typical data access actions
  - open, close, read, write, etc.

# Data access in a distributed system

## Client side

- Client request to access data
- Check client cache
  - Data present Return to client
  - Else Check local disk
    - Data present Return to client
    - Else Send request to file server through n/w

## Server side

- Check server cache
  - Data present ; load data to client cache through n/w; return to client.
  - Else Issue disk read; Local server cache; load data to client cache through n/w; return to client.

# Distributed File system design

File service vs. file server

- **File service interface:** the specification of what the file system offers to its clients.
- Implemented by a user/kernel process called file server
- **File server:** a process that runs on some machine and helps implement the file service.
- A system may have one or several file servers running at the same time

# Remote Files

- **What is a file?**
  - Uninterpreted sequence of bytes
  - Can be structured as a sequence of records
- **Files can have attributes**
  - Owner, size, creation date and access permissions
- **File model**
  - Files can be modified or
  - Immutable files
- **File Protection**
  - Capability
  - Access control list
- **File Service Model**
  - **upload/download model**
  - files move between server and clients, few operations (read file & write file), simple, requires storage at client, good if whole file is accessed
  - **remote access model**
  - files stay at server, reach interface for many operations, less space at client, efficient for small accesses

# Directory Service

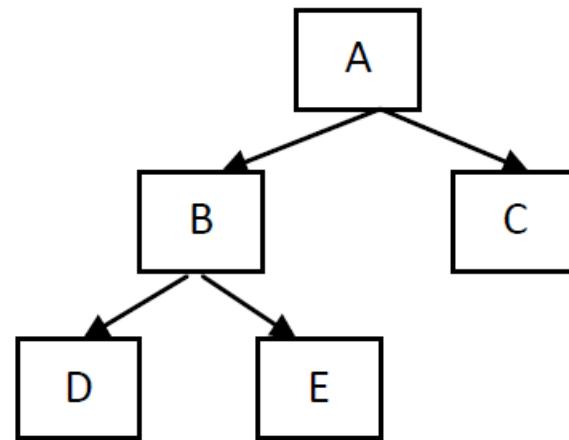
The directory service

- creating and deleting directories
- naming and renaming files
- moving files

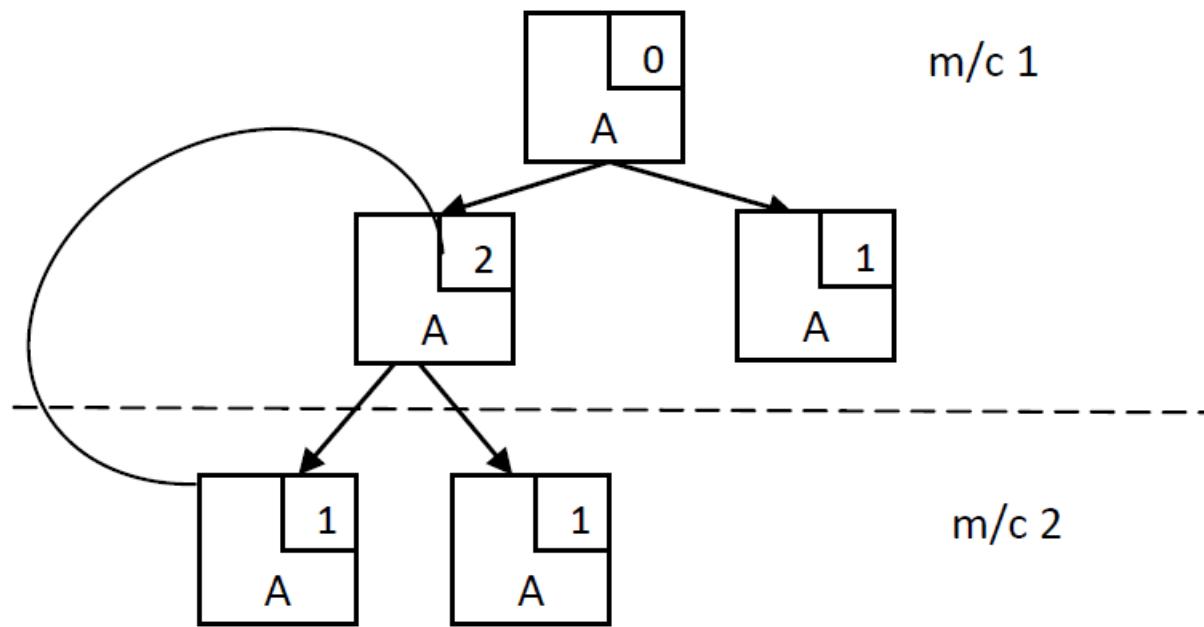
Clients can have the same view (global root directory)

or different views of the file system (remote mounting)

## Hierarchical file system Directory Tree



## Directory Graph



# Naming and Name Resolution

- a name space -- collection of names
- name resolution -- mapping a name to an object
  - same or different view of a directory hierarchy
- 3 traditional ways to name files in a distributed environment
  - concatenate the host name to the names of files stored on that host:  
system-wide uniqueness guaranteed, simple to locate a file; however,  
not network transparent, not location independent, e.g.,  
`/machine/usr/foo`
  - mount remote directories onto local directories:  
once mounted, files can be referenced in a location-transparent manner
  - provide a single global directory:  
requires a unique file name for every file, location independent,  
cannot encompass heterogeneous environments and wide geographical  
areas

# Two-Level Naming

- Symbolic name (external), e.g. prog.c; binary name (internal), e.g. local i-node number as in Unix
- Directories provide the translation from symbolic to binary names
- Binary name format
  - i-node: no cross references among servers
  - (server, i-node): a directory in one server can refer to a file on a different server
  - Capability specifying address of server, number of file, access permissions, etc
  - {binary\_name+}: binary names refer to the original file and all of its backups

# File Sharing Semantics

UNIX semantics:

- Total ordering of R/W events
- Value read is the value stored by last write
- Writes to an open file are visible immediately to others that have this file opened at the same time.
- Easy to achieve in a non-distributed system ; In a distributed system with one server and multiple clients with no caching at client.

# Session semantics:

- Writes to an open file by a user is not visible immediately by other users that have files opened already. Once a file is closed, the changes made by it are visible by sessions started later.
- Writes are guaranteed to become visible only when the file is closed
- Allow caching at client with lazy updating -> better performance
- If two or more clients simultaneously write: one file (last one or non-deterministically) replaces the other

- **Immutable files:**

create and read file operations (no write) - i.e. a sharable file cannot be modified.

File names cannot be reused and its contents may not be altered.

Simple to implement.

Writing a file means to create a new one and enter it into the directory replacing the previous one with the same name: atomic operations

Collision in writing: last copy or nondeterministically

## Transaction semantics:

- mutual exclusion on file accesses; either all file operations are completed or none is. Good for banking systems
- All changes have all-or-nothing property.  $W1,R1,R2,W2$  not allowed where  $P1 = W1;W2$  and  $P2 = R1;R2$

# Distributed File system

## Implementation

- System Structure
  - Clients and servers on different machines?
    - Combine File and directory services
    - Keep them separate
  - Lookups
    - Iterative lookup
    - Automatic lookup

# Stateless vs. Stateful

## Stateless Server

- ❑ requests are self-contained
- ❑ better fault tolerance
- ❑ open/close at client (fewer msgs)
- ❑ no space reserved for tables
- ❑ thus, no limit of open files
- ❑ no problem if client crashes

## Stateful Servers

- ❑ shorter messages
- ❑ better performance (info in memory until close)
- ❑ open/close at server
- ❑ file locking possible
- ❑ read ahead possible

# Caching

Four places to store files

- server's disk: slow performance
  - eliminates coherence problem
- server caching: in main memory
  - cache management issue, how much to cache, replacement strategy
  - still slow due to network delay
  - Used in high-performance web-search engine servers

## – Client caching in main memory

- can be used by diskless workstation
- faster to access from main memory than disk
- compete with the virtual memory system for physical memory space
- avoids disk access but still network access

## Three Options

- inside each process address space: no sharing at client
- in the kernel: kernel involvement on hits
- in a separate user-level cache manager: flexible and efficient if paging can be controlled from user-level

- client-cache on a local disk
  - large files can be cached
  - the virtual memory management is simpler
  - a workstation can function even when it is disconnected from the network

# Update algorithms for client caching

- write-through:
  - all writes are carried out immediately
  - writes sent to the server as soon as they are performed at the client -> high traffic, requires cache managers to check (modification time) with server before can provide cached content to any client
  - Reliable: little information is lost in the event of a client crash
  - Slow: cache not that useful
- delayed-write:
  - delays writing at the server
  - coalesces multiple writes; better performance but ambiguous semantics
  - possible to perform many writes to a block in the cache before it is written
  - if data is written and then deleted immediately, data need not be written at all (20-30 % of new data is deleted with 30 secs)

- **write-on-close:**
  - delay writing until the file is closed at the client
  - Implements session semantics
  - if file is open for short duration, works fine
  - if file is open for long, susceptible to losing data in the event of client crash
- **Central control:**
  - file server keeps a directory of open/cached files at clients -> Unix semantics, but problems with robustness and scalability; problem also with invalidation messages because clients did not solicit them

# Cache Coherence

- How to maintain consistency between locally cached data with the master data when the data has been modified by another client?
  - 1 Client-initiated approach -- check validity on every access:  
too much overhead first access to a file (e.g., file open)  
every fixed time interval
  - 2 Server-initiated approach -- server records, for each client,  
the (parts of) files it caches. After the server detects a  
potential inconsistency, it reacts.
  - 3 Not allow caching when concurrent-write sharing occurs.  
Allow many readers. If a client opens for writing, inform  
all the clients to purge their cached data.

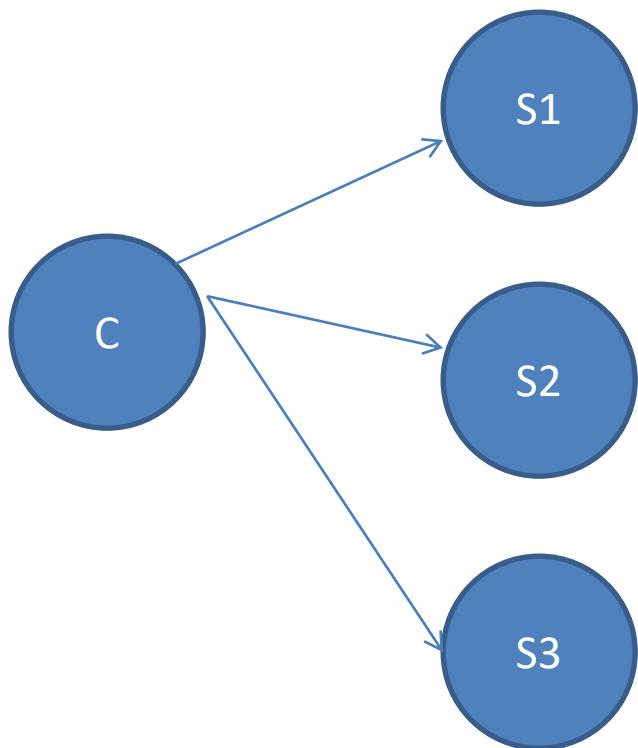
- Potential inconsistency:
  - In session semantics, a client closes a modified file.
  - In UNIX semantics, the server must be notified whenever a file is opened and the intended mode (read or write mode) must be indicated for every open.
  - Disable cache when a file is opened in conflicting modes.

# Replication

# File Replication

- Multiple copies are maintained, each copy on a separate file server
- Reasons:
  - To improve reliability; availability and performance.
- Replication transparency
  - explicit file replication: programmer controls replication
  - lazy file replication: copies made by the server in background
  - use group communication: all copies made at the same time in the foreground

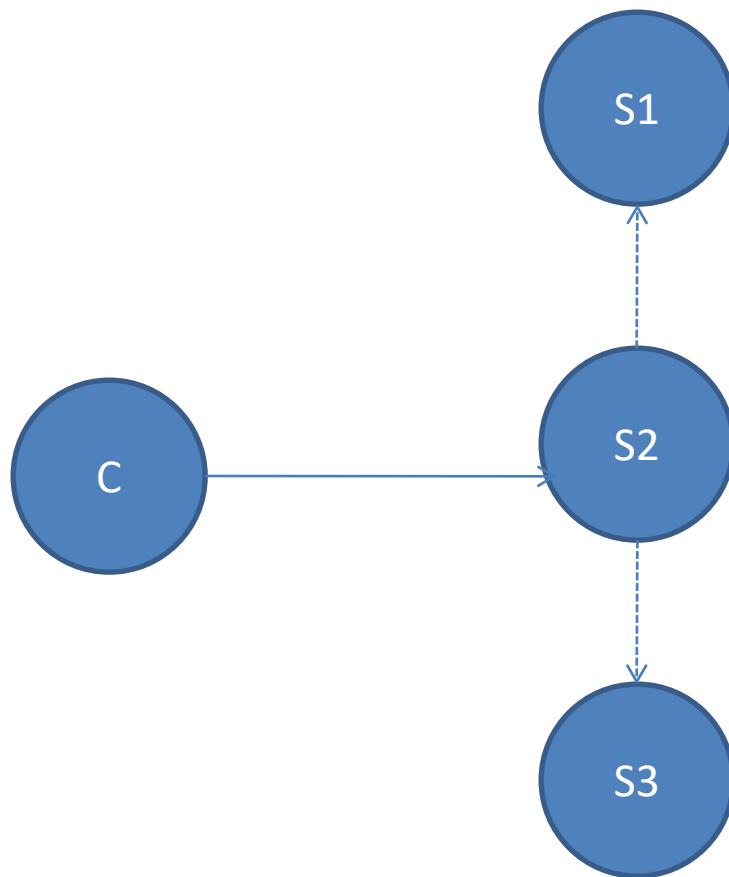
# Explicit File Replication



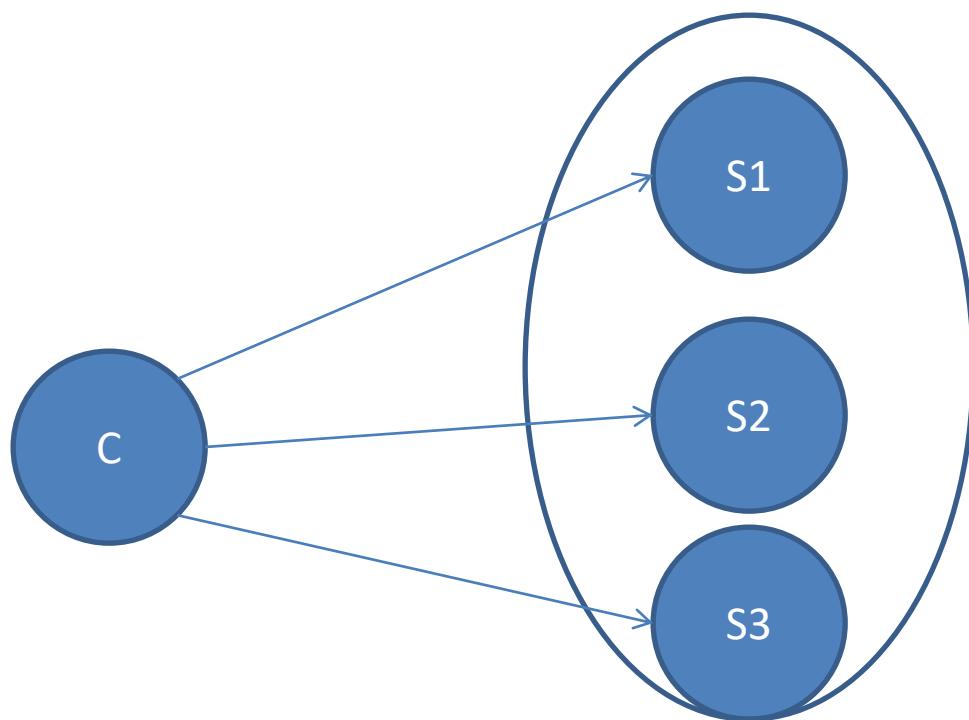
File : 1.14 : 2.16 : 3.19

PI : 1.2.1 : 2.43 : 3.41

# Lazy replication



# File replication using Group



# Update protocols

- Primary Copy Replication
- Voting
- Voting with ghosts

# Modifying Replicas: Voting Protocol

- Updating all replicas using a coordinator works but is not robust (if coordinator is down, no updates can be performed) => Voting: updates (and reads) can be performed if some specified # of servers agree.
- Voting Protocol:
  - A version # (incremented at write) is associated with each file
  - To perform a read, a client has to assemble a read quorum of Nr servers; similarly, a write quorum of Nw servers for a write
  - If  $Nr + Nw > N$ , then any read quorum will contain at least one most recently updated file version
  - For reading, client contacts Nr active servers and chooses the file with largest version #
  - For writing, client contacts Nw active servers asking them to write. Succeeds if they all say yes.

# Voting Protocol with Ghosts

- $N_r$  is usually small (reads are frequent), but  $N_w$  is usually close to  $N$  (want to make sure all replicas are updated). Problem with achieving a write quorum in the presence of server failures
- Voting with ghosts: allows to establish a write quorum when several servers are down by temporarily creating dummy (ghost) servers (at least one must be real)
- Ghost servers are not permitted in a read quorum (they don't have any files)
- When server comes back it must restore its copy first by obtaining a read quorum

# Distributed Synchronization

# Distributed Systems

A distributed system consists of a number of processes.

Each process has a state ( Values of variables)

Each process takes action to change its state, which may be an instruction or a communication action ( send, receive)

- Each process has a common clock and events within a process can be assigned timestamps and thus ordered.
- In DS, the time order of events across different processes are also to be considered.
- The multiple processors do not share a common memory or clock.
- Instead, each processor has its own local memory and communicates with other processes through communication lines.

# The Importance of Synchronization

- Various components of a distributed system must cooperate and exchange information, synchronization is a necessity.
- Constraints, both implicit and explicit, are therefore enforced to ensure synchronization of components.
- Synchronization is required for
  - Fairness
  - Correctness

# *Clock Synchronization*

- Logical Clocks
  - Happened Before relation
- Physical Clocks
  - Centralized
    - Broadcast Based
    - Request Driven
  - Decentralized

# Physical Clocks & Synchronization

- In a DS, each process has its own clock.
- Clock Skew versus Drift
  - Clock Skew = Relative Difference in clock *values* of two processes
  - Clock Drift = Relative Difference in clock *frequencies (rates)* of two processes
- *A non-zero clock drift will cause skew to continuously increase.*

# Implementation of Physical Time Service

- Obtaining accurate value when implementing a physical time service
- Synchronizing the concept of physical time throughout the distributed system.
- Implementation
  - Centralized
  - Distributed

# Centralized Physical service

- Broadcast based
- Request driven

## Problems

- Single point of failure
- Traffic around server increases
- Not scalable

# Distributed Physical service

- Client broadcast its current time at predefined set intervals
- Starts timer
- Collects messages
- Calculates average values and then adjust time values.

# *Process Synchronization*

Techniques to coordinate execution among processes

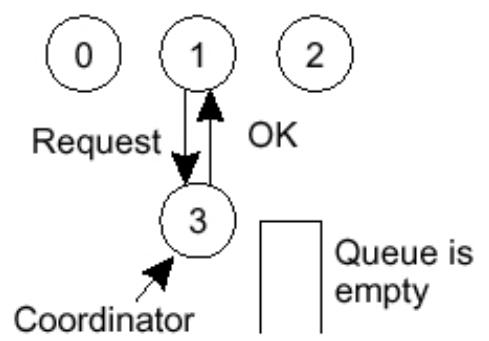
- One process may have to wait for another
- Shared resource (e.g. critical section) may require exclusive access

# *Mutual Exclusion*

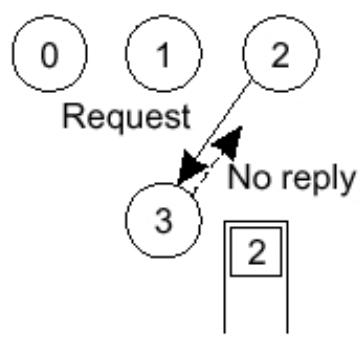
- Centralized
- Distributed
  - Lock-based (aka permission based, non-token based) - to enter the CS a process needs to obtain permission from other processes in the system.
  - Token-based - unique token (privilege) circulated in the system. A process possessing the token can enter CS

# Centralized Algorithm

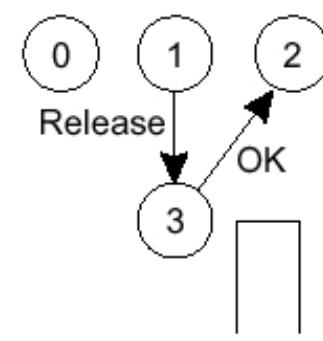
- One process is elected as the coordinator.
- When any process wants to enter a critical section, it sends a request message to the coordinator stating which critical section it wants to access.
- If no other process is currently in that critical section, the coordinator sends back a reply granting permission. When the reply arrives, the requesting process enters the critical section. If another process requests access to the same critical section, it is ignored or blocked until the first process exits the critical section and sends a message to the coordinator stating that it has exited.



(a)



(b)



(c)

# Distributed Algorithm – Permission based

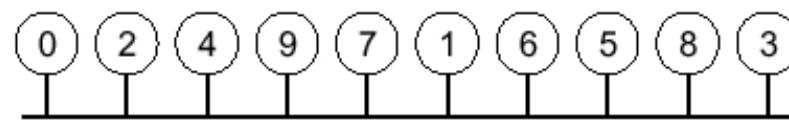
- When a process wants to enter a critical section, it builds a message containing the name of the critical section, its process number, and the current time. It then sends the message to all other processes, as well as to itself.

- When a process receives a request message, the action it takes depends on its state with respect to the critical section named in the message. There are three cases: If the receiver is not in the critical section and does not want to enter it, it sends an OK message to the sender.
- If the receiver is in the critical section, it does not reply. It instead queues the request.
- If the receiver also wants to enter the same critical section, it compares the time stamp in the incoming message with the time stamp in the message it has sent out. The lowest time stamp wins. If its own message has a lower time stamp, it does not reply and queues the request from the sending process.
- When a process has received OK messages from all other processes, it enters the critical section. Upon exiting the critical section, it sends OK messages to all processes in its queue and deletes them all from the queue.

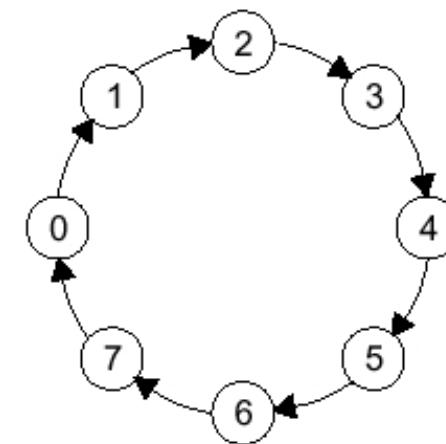
# Token Based Algorithm

- Another approach is to create a logical or physical ring.
- Each process knows the identity of the process succeeding it.
- When the ring is initialized, Process 0 is given a token. The token circulates around the ring in order, from Process k to Process k + 1.
- When a process receives the token from its neighbor, it checks to see if it is attempting to enter a critical section. If so, the process enters the critical section and does its work, keeping the token the whole time.

- After the process exits the critical section, it passes the token to the next process in the ring. It is not permitted to enter a second critical section using the same token.
- If a process is handed a token and is not interested in entering a critical section, it passes the token to the next process.



(a)



(b)