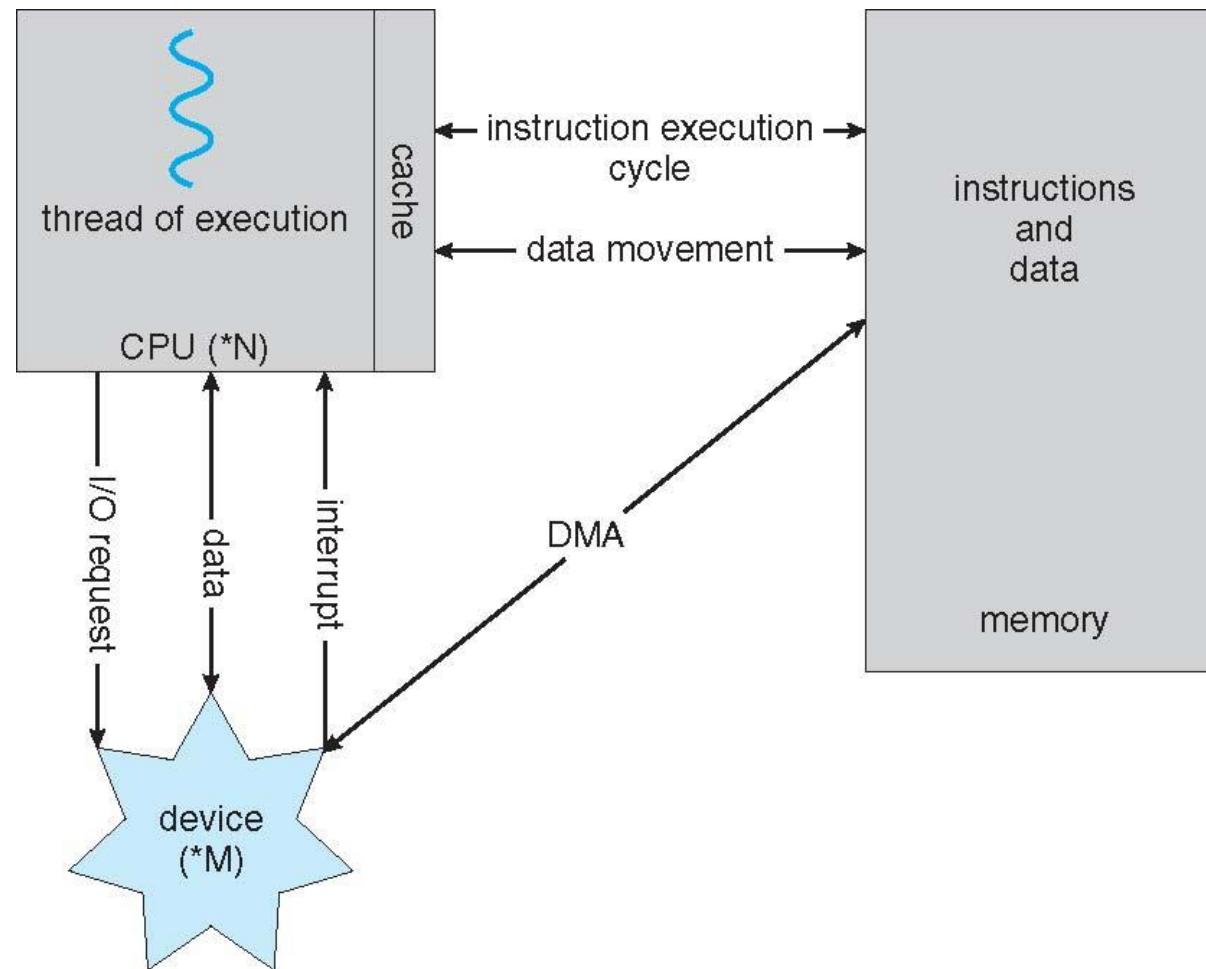


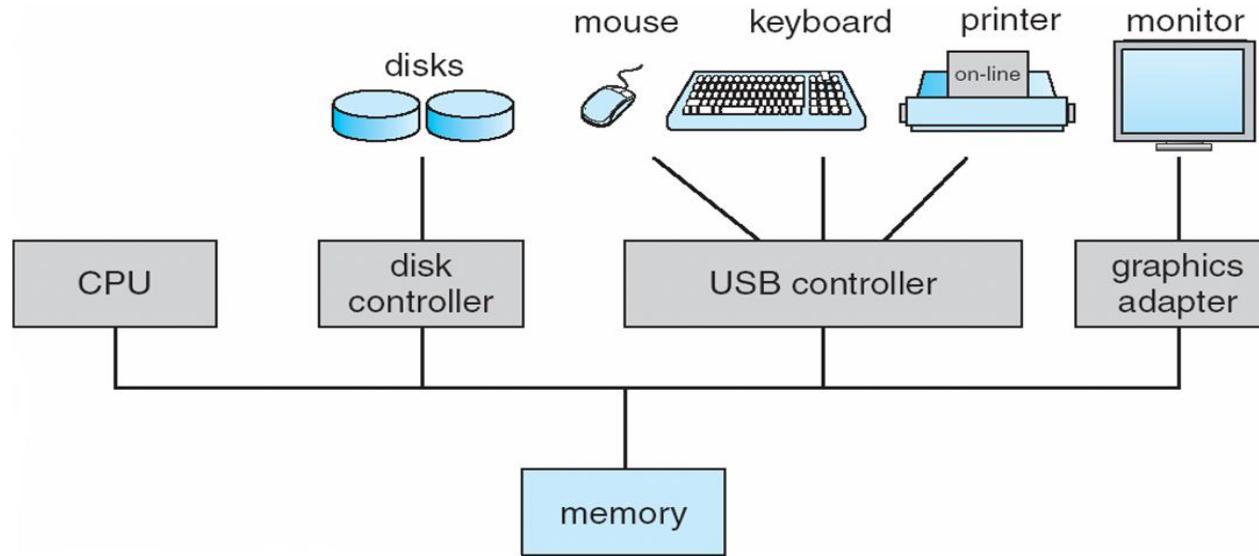
OPERATING SYSTEMS

- 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

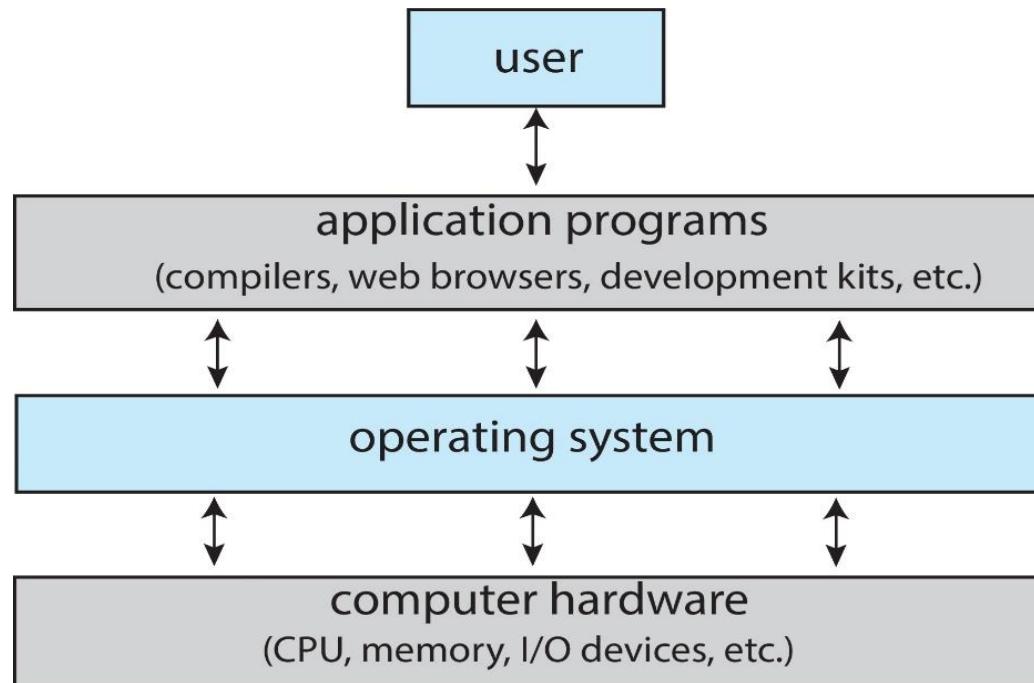
- It provides basic computing resources
 - CPU, memory, I/O devices
- It may be composed of two or more levels.
 - The lowest level consists of the basic resources.
 - The next level is the micro architecture level where physical resources are grouped together as functional units. The basic level operations 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:

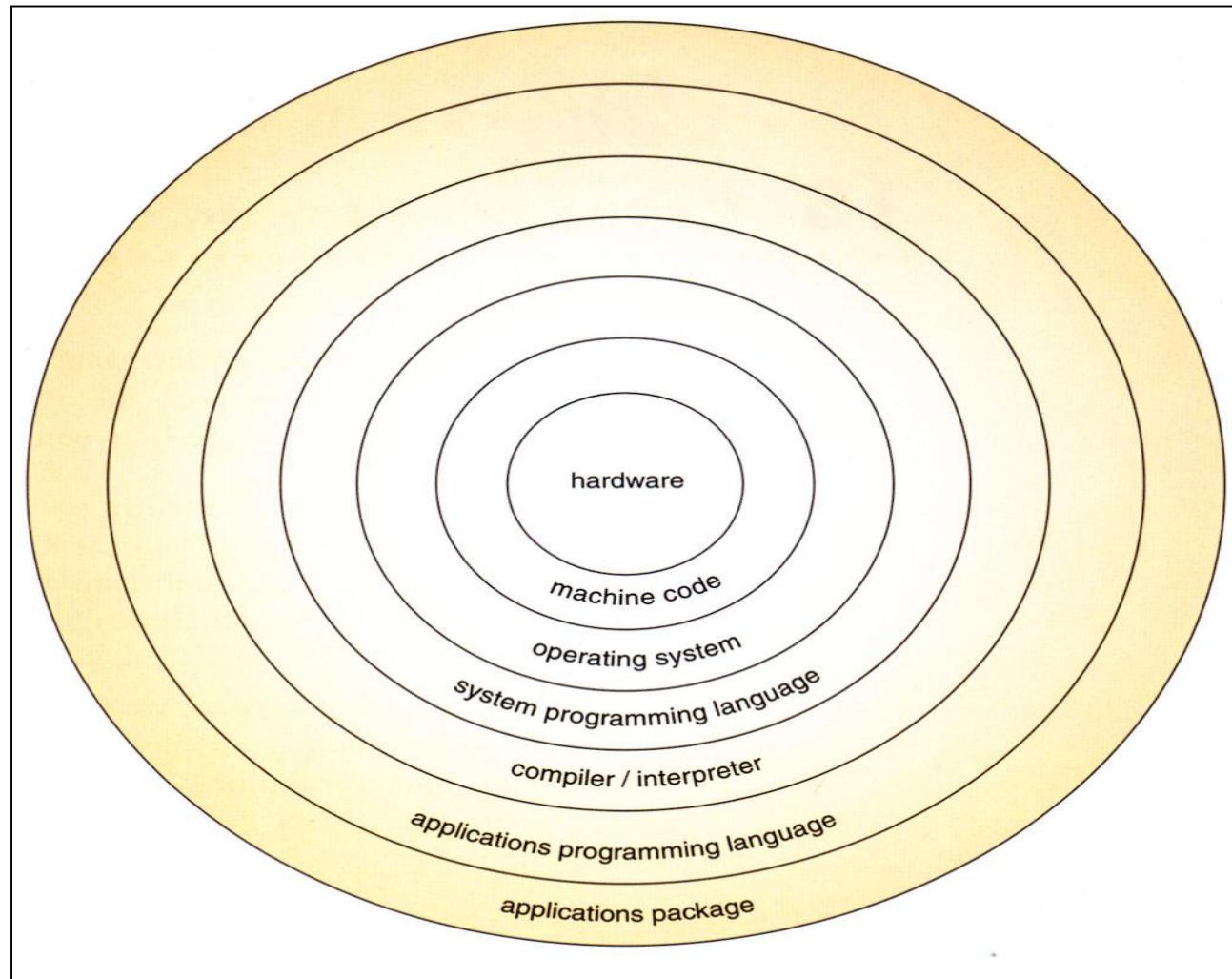
- Operating System: It hides the underlying complex hardware. It controls and coordinates use of hardware among various applications and users.
- Utilities and Application Software: define the ways of in which the system resources are used to solve the computing problems of the users. The system programs 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)

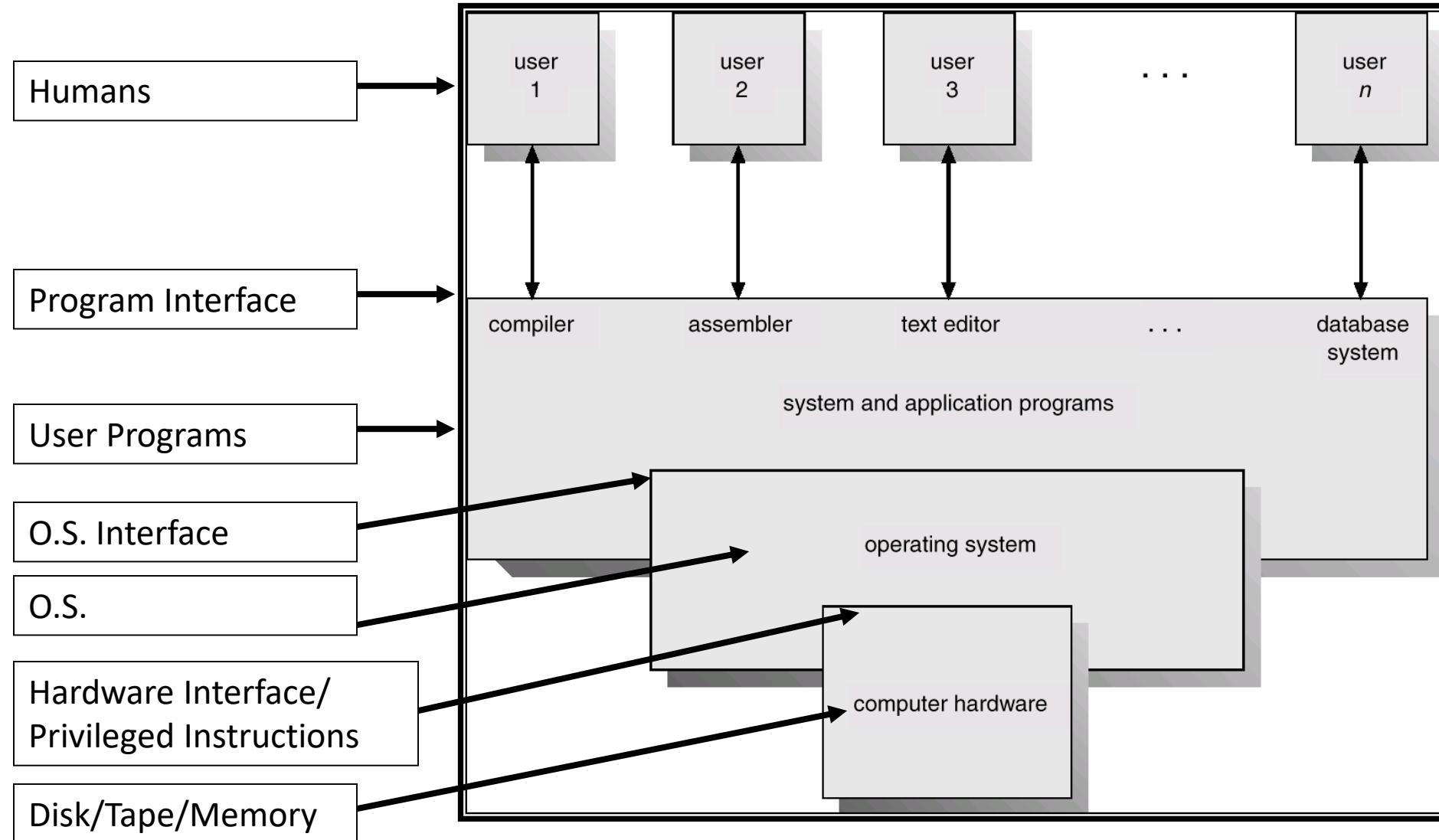
- small program that allows a specific input or output device to communicate with the rest of the computer system

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

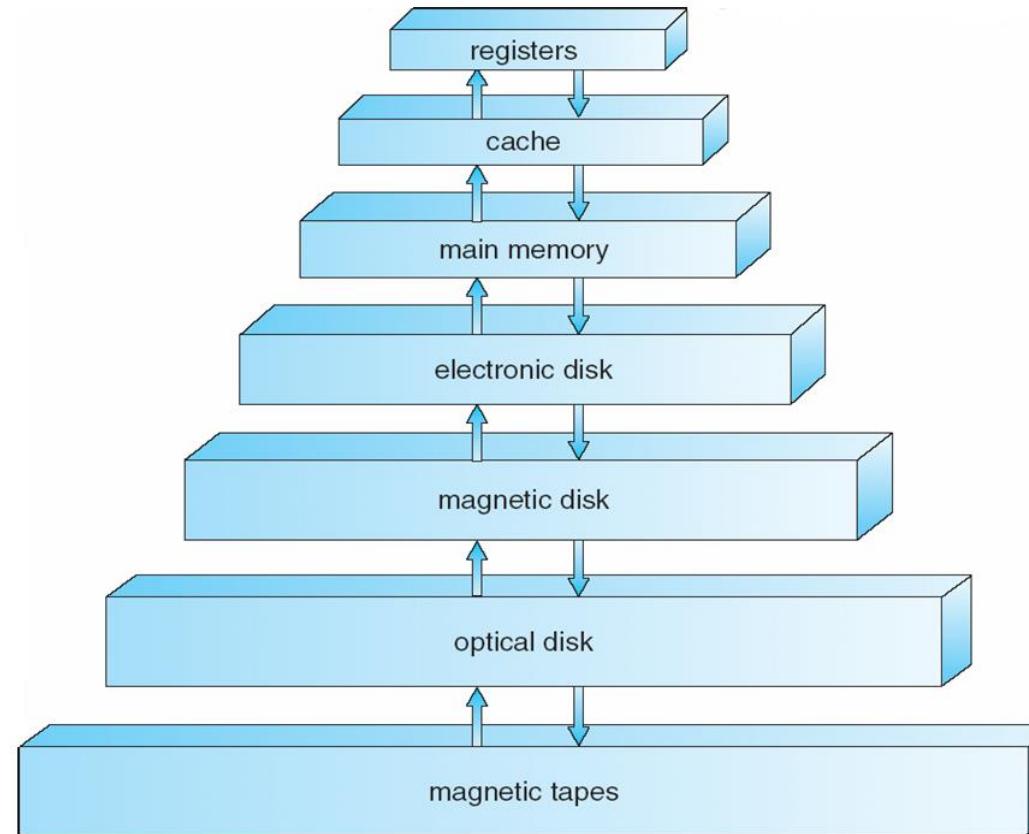
Storage Hierarchy

- 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

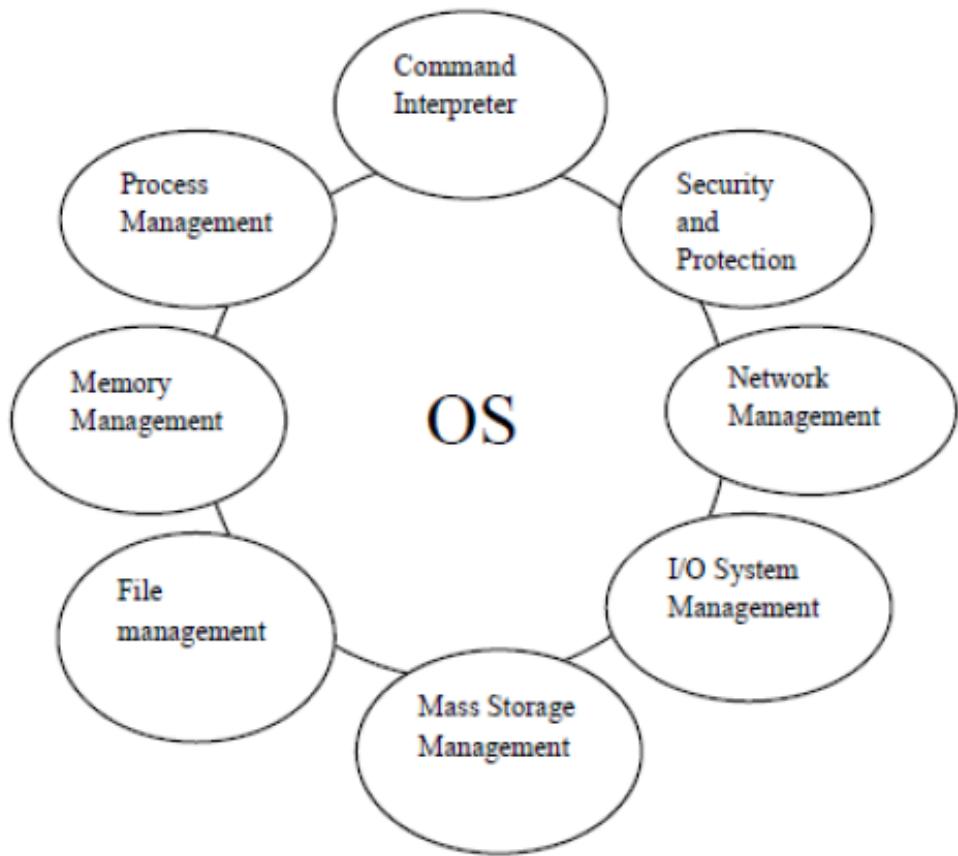
- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
 - If it is, information used directly from the cache (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>			
	Hardware	Example OS Services	User Abstraction
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

OS Service	Hardware Support
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..):

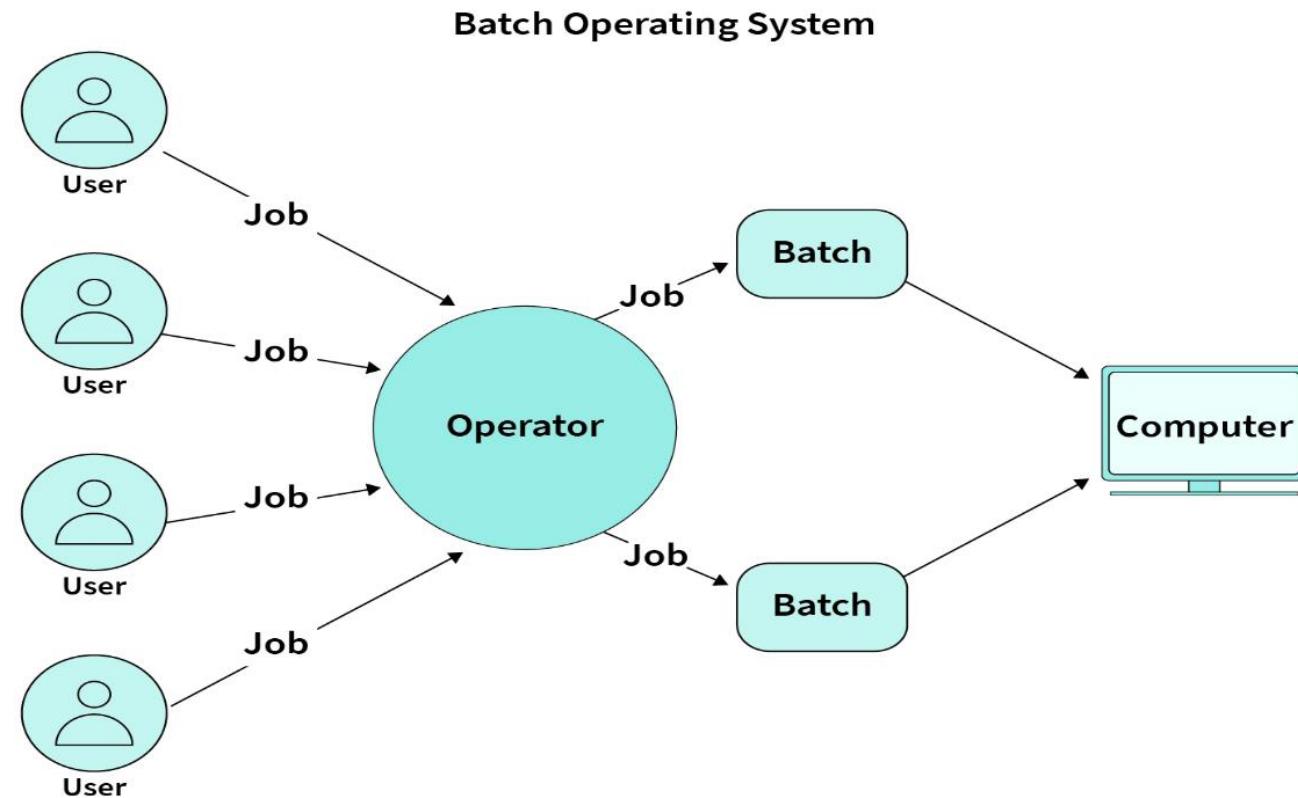
Major Phases	Technical Innovations	Operating Systems
Open Shop	The idea of OS Programmers write programs and submit tape/cards to operator. Operator feeds cards, collects output from printer.	IBM 701 open shop (1954)
Batch Processing	Tape batching, First-in, first-out scheduling.	BKS system (1961)
Multi-programming	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.

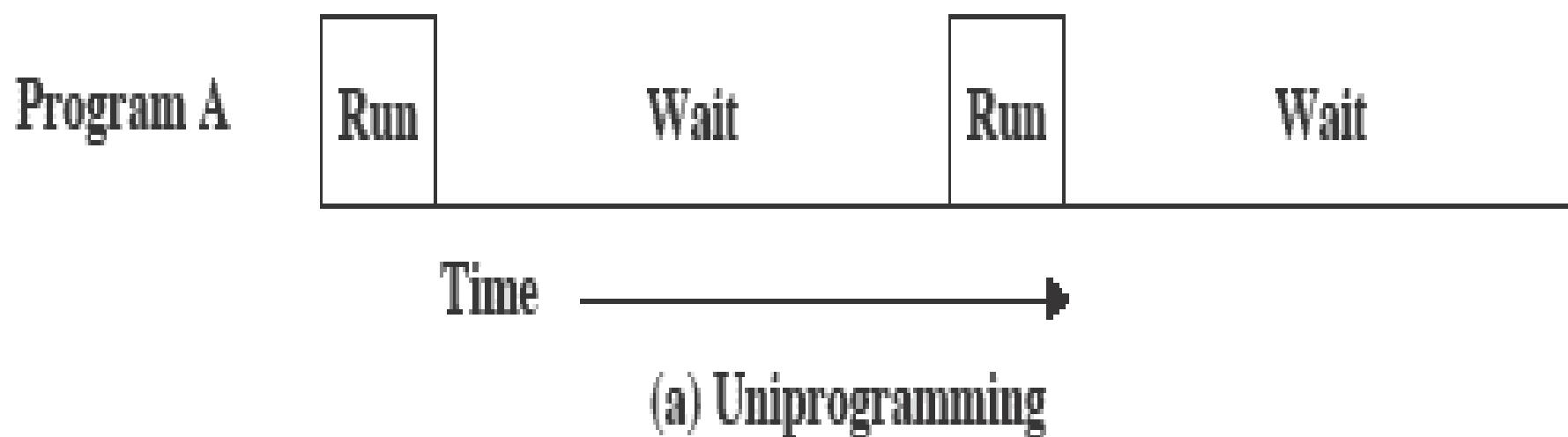
- use interrupt driven I/O.
- use magnetic disk to cache input tape.
- Monitor now schedules jobs. . .

Advantages of Simple Batch Systems

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

Uniprogramming

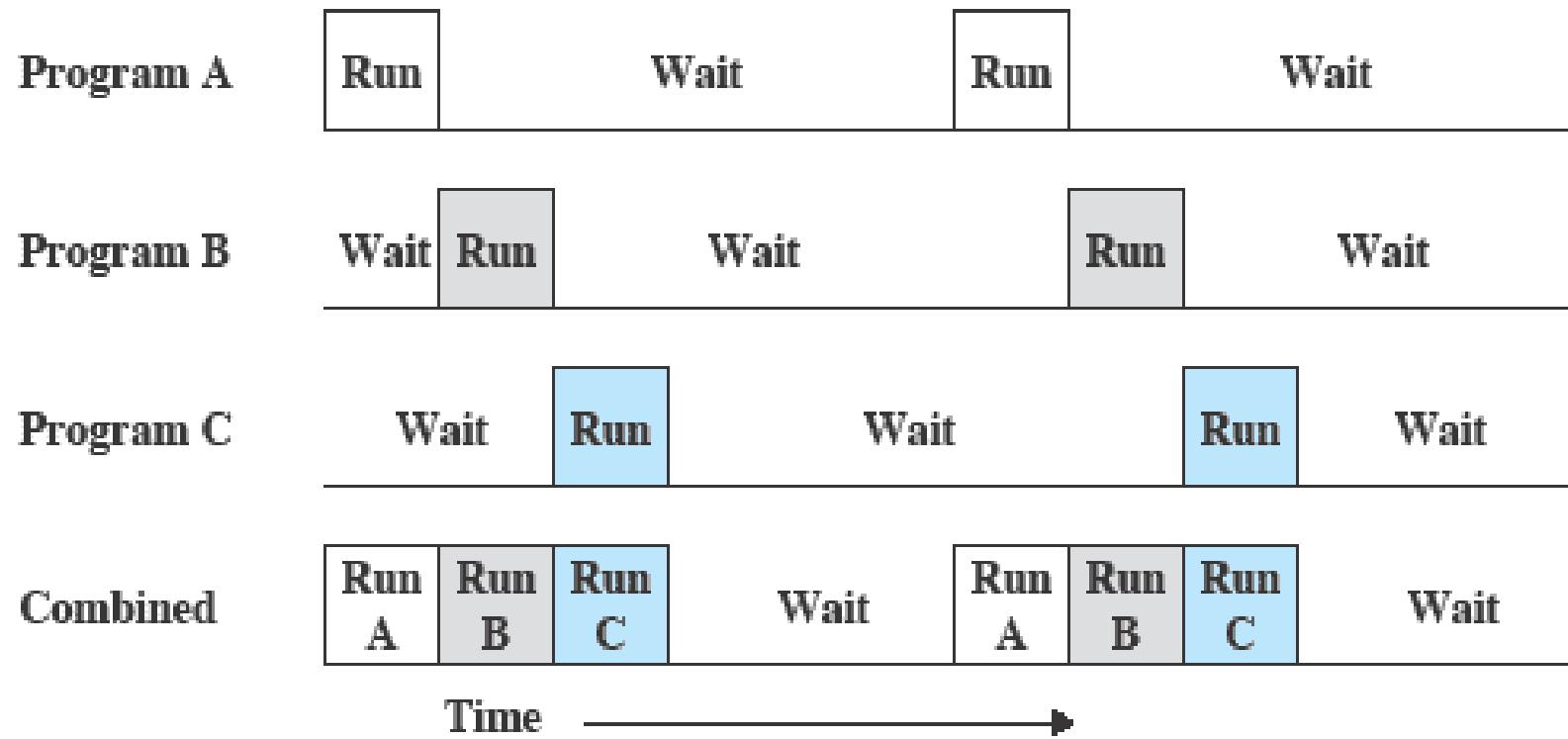
- Processor must wait for I/O instruction to complete before preceding



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).

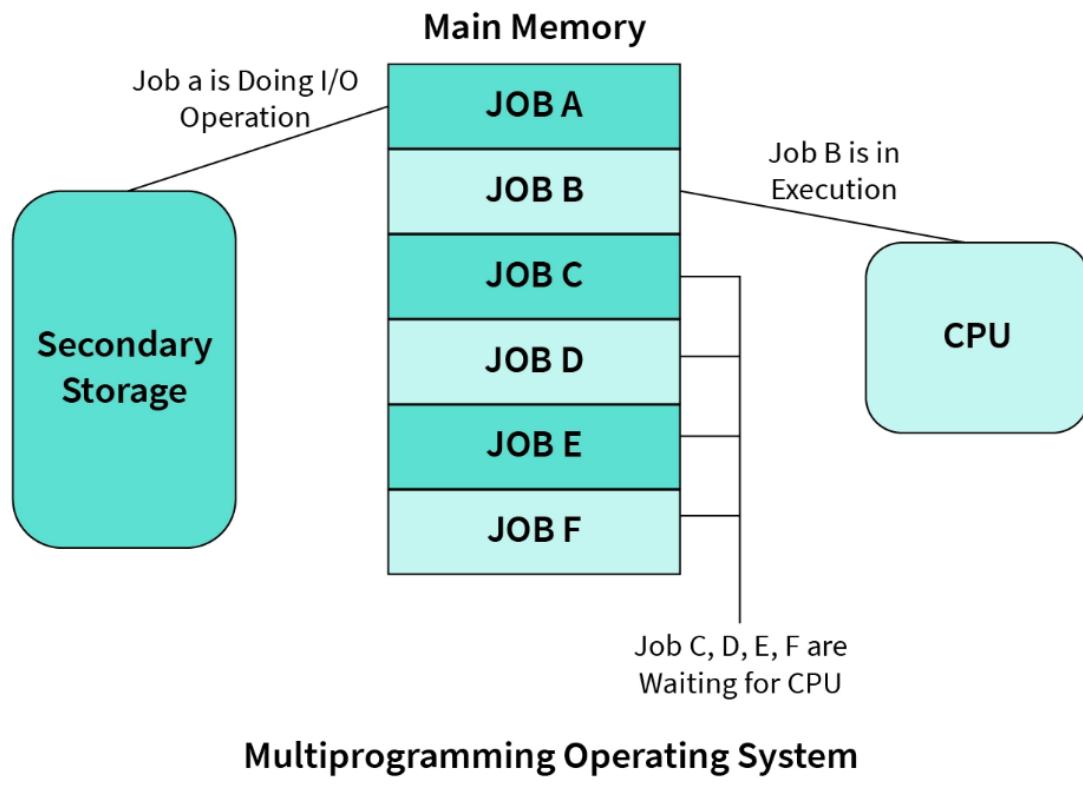
Multiprogramming



- 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.



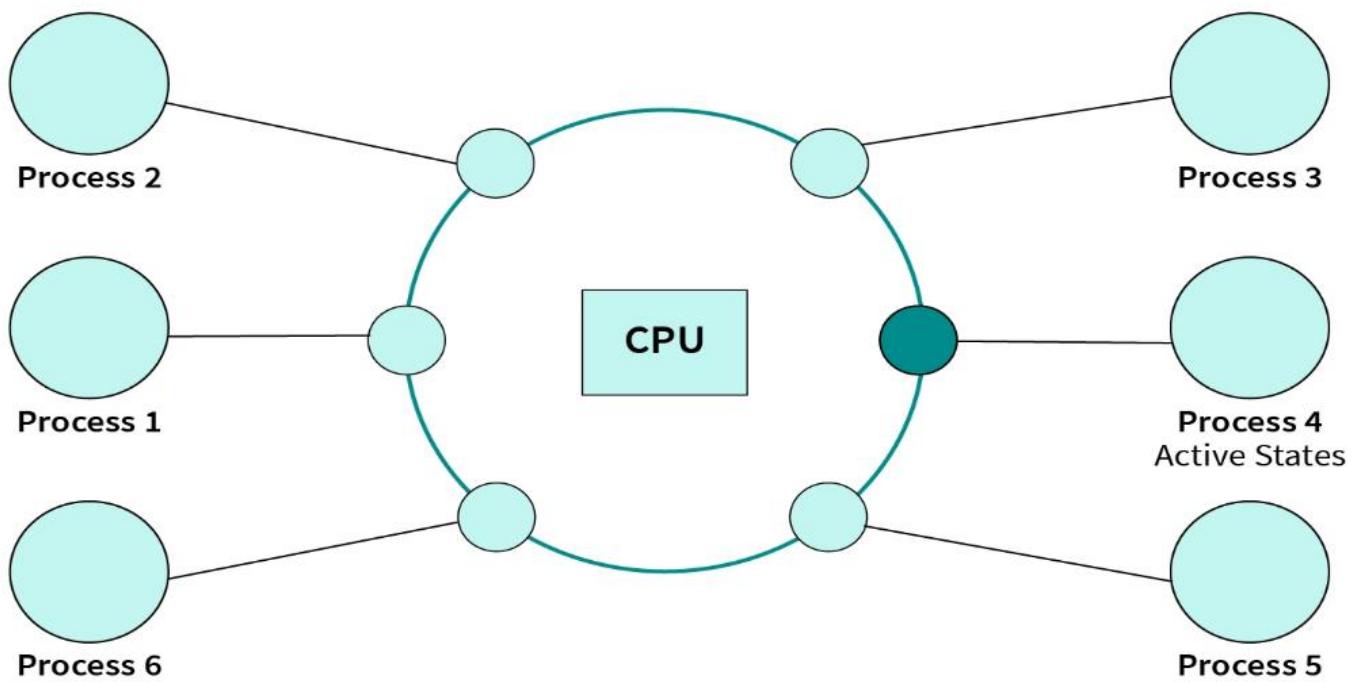
Advantages:

- 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.

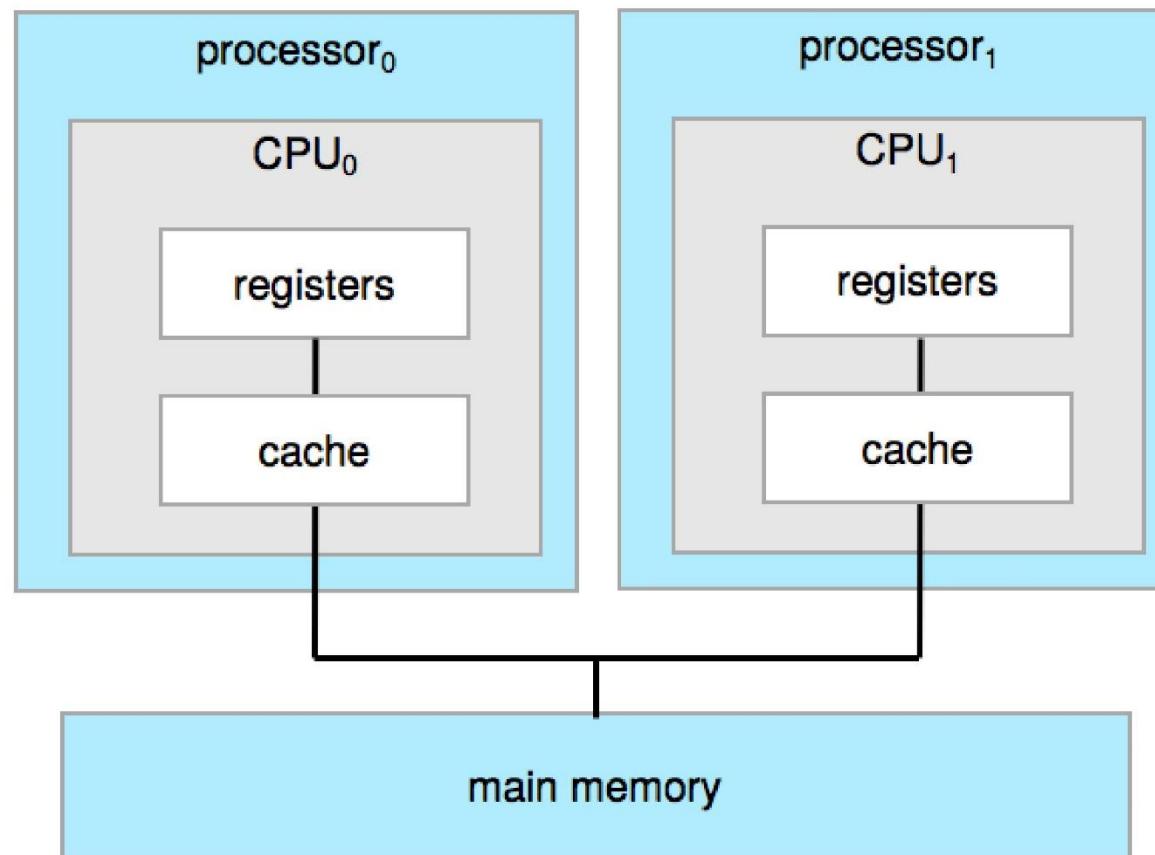
Advantages of Multiprocessor Systems

- 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

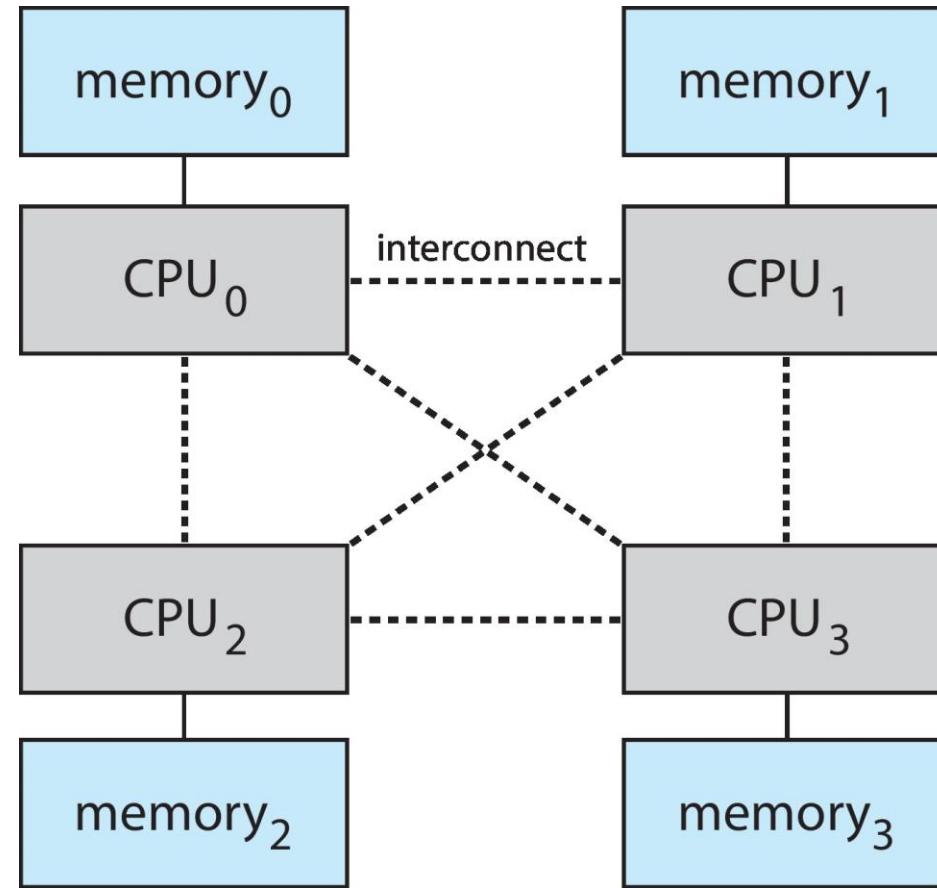
Two types:

- 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

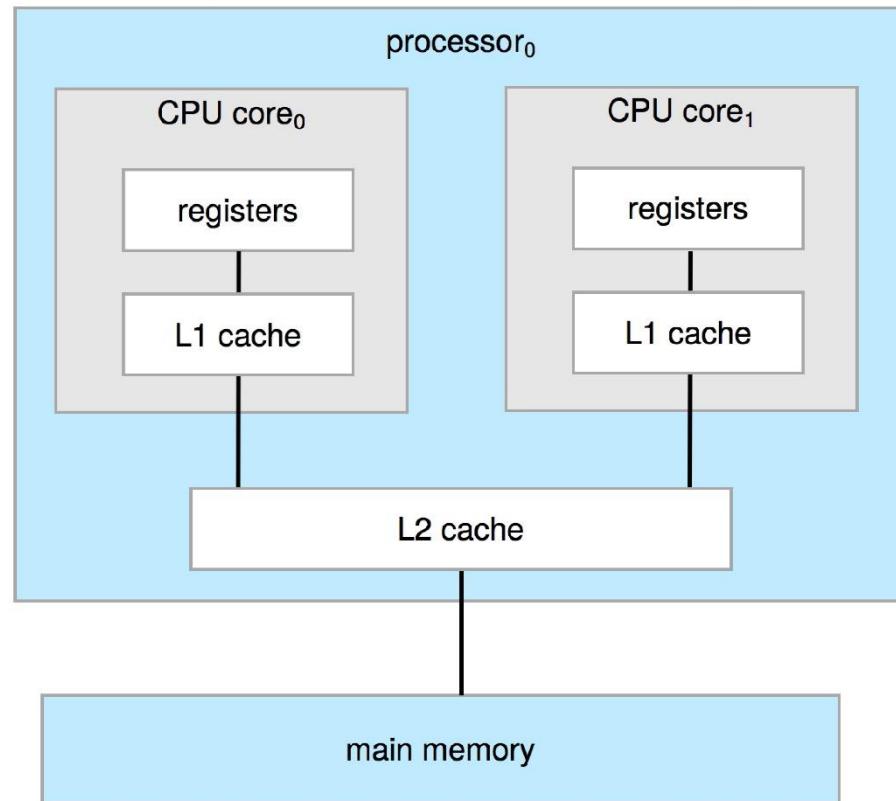


A Dual-Core Design

- 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.

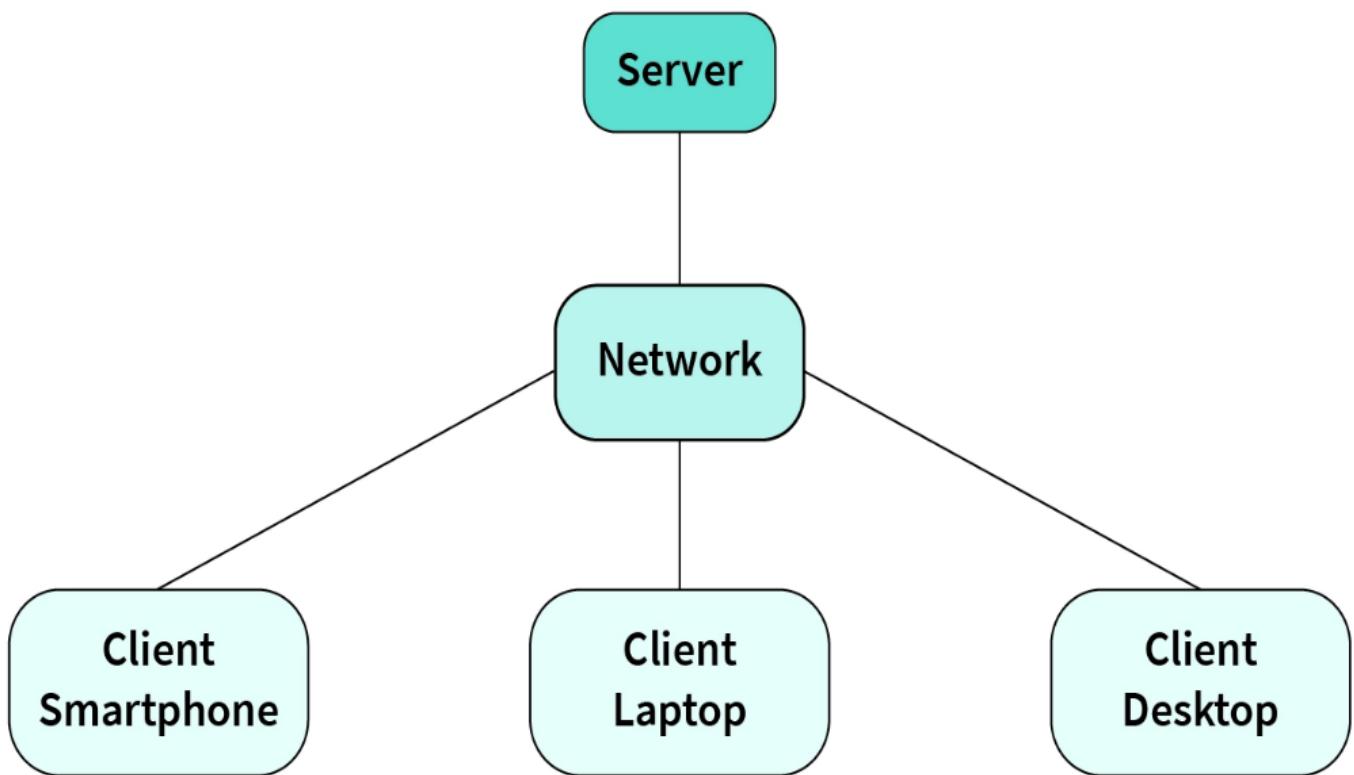


Blade servers

- 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

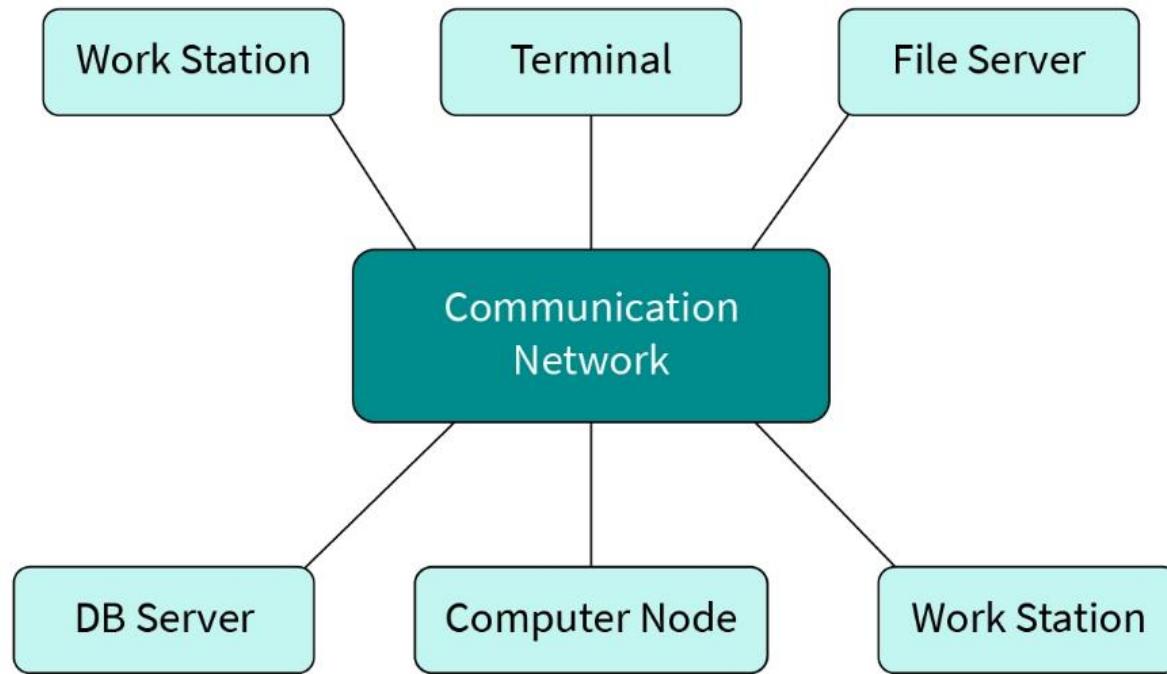
- PC operating systems were neither **multiuser** nor **multitasking**.
- However, the goals of these operating systems have changed with time; instead of maximizing CPU and peripheral utilization, the systems opt for maximizing user convenience and responsiveness. These systems are called **Desktop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

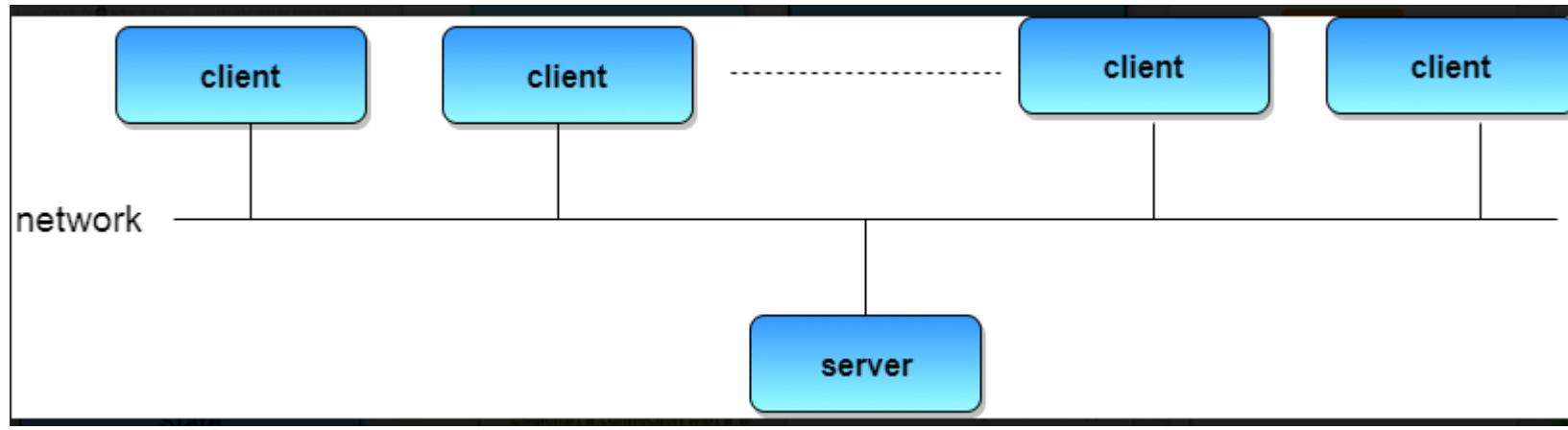


Types of Distributed Operating Systems

- 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.



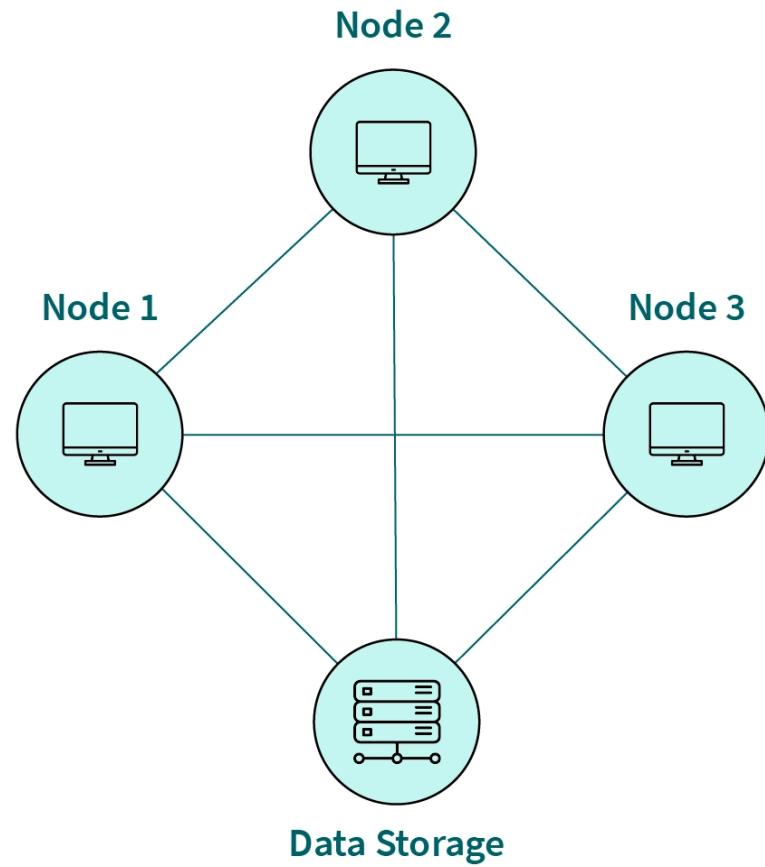
Peer-to-Peer Systems

- 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

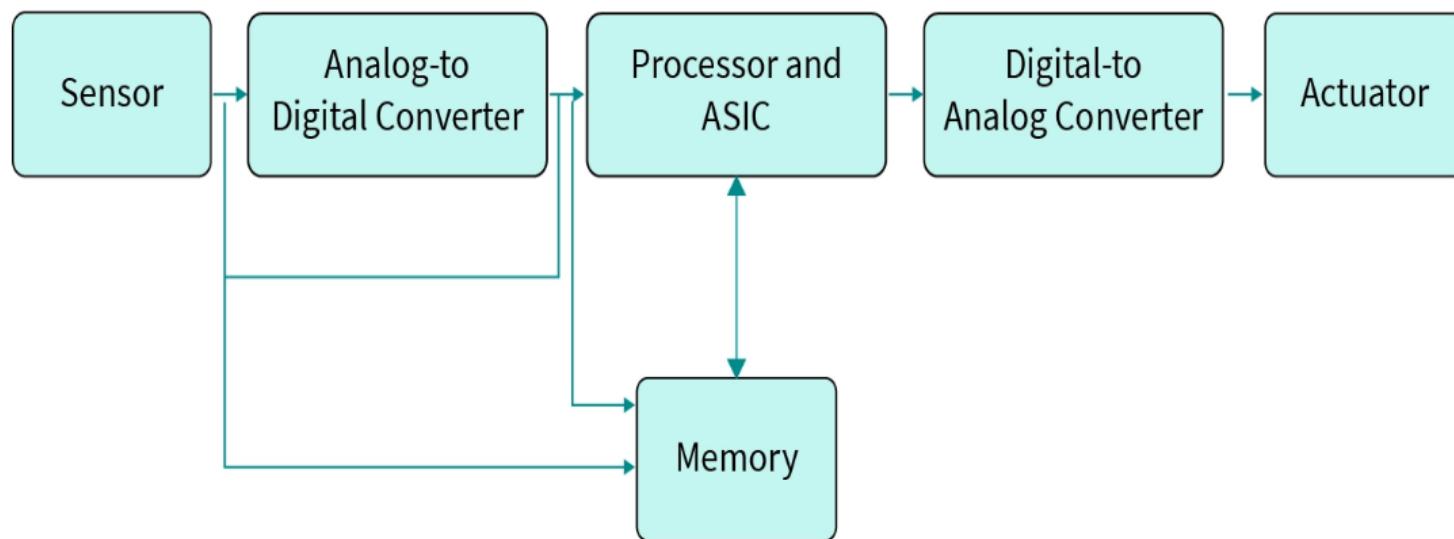
- Handheld systems include **Personal Digital Assistants(PDAs)**, such as Palm-Pilots or Cellular Telephones with connectivity to a network such as the Internet.
- They are usually of limited size due to which most handheld devices have a small amount of memory, include slow processors, and feature small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

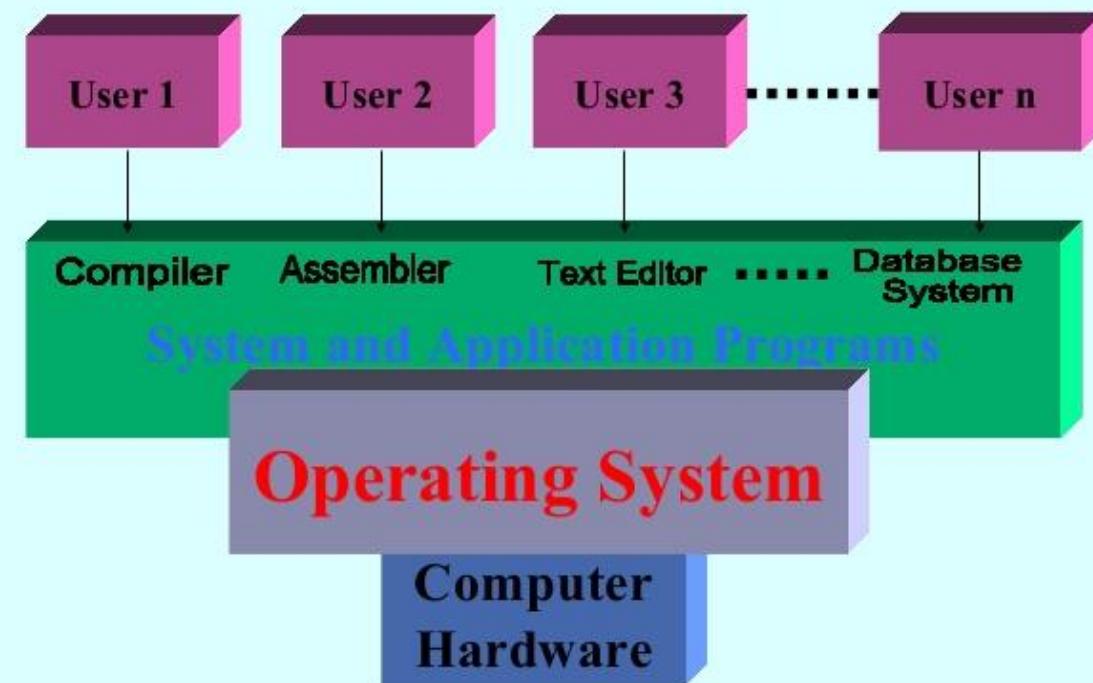


Mobile Operating System

- 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

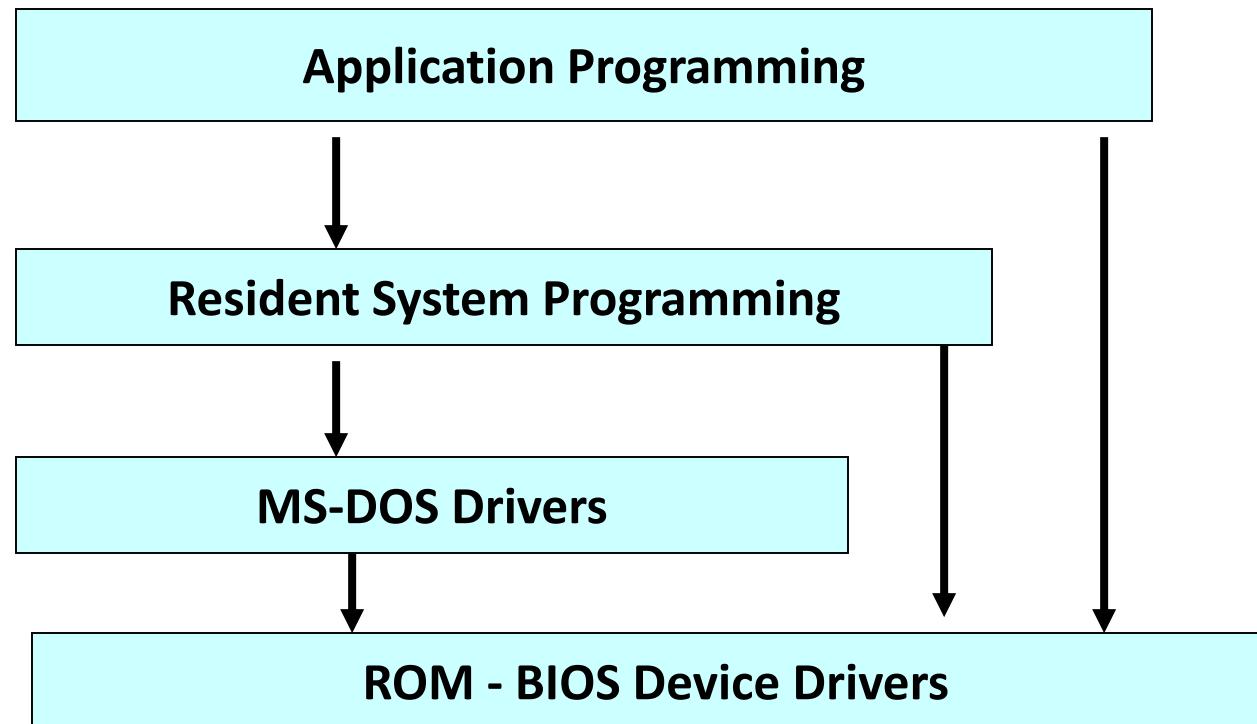
- The OS is a concurrent, real-time program.
- Concurrency arises naturally in an OS when it supports concurrent applications, and because it must interact directly with the hardware.
- Hardware interactions also impose timing constraints

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.

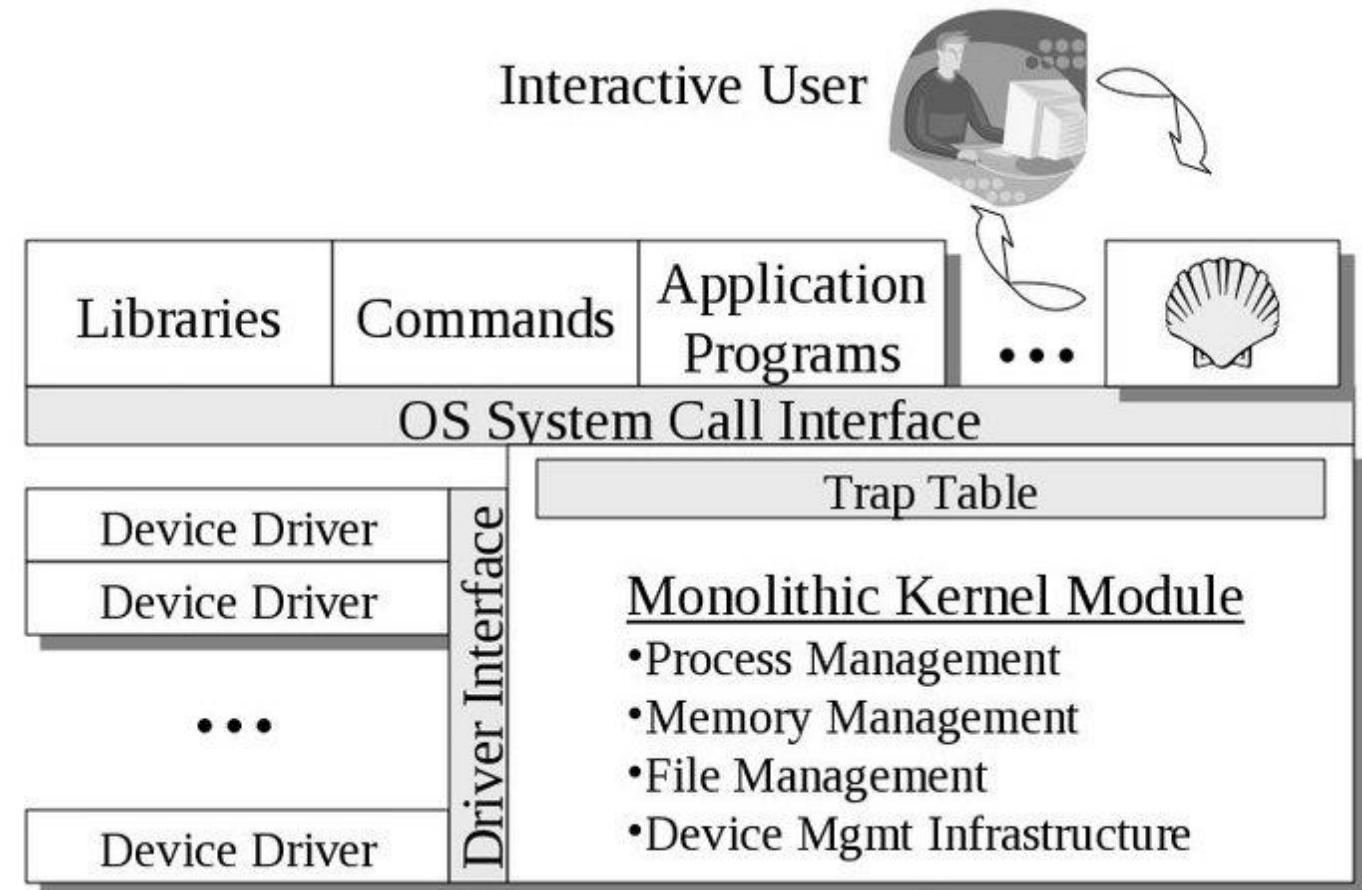
e.g. OS/360, VMS and Linux.

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.

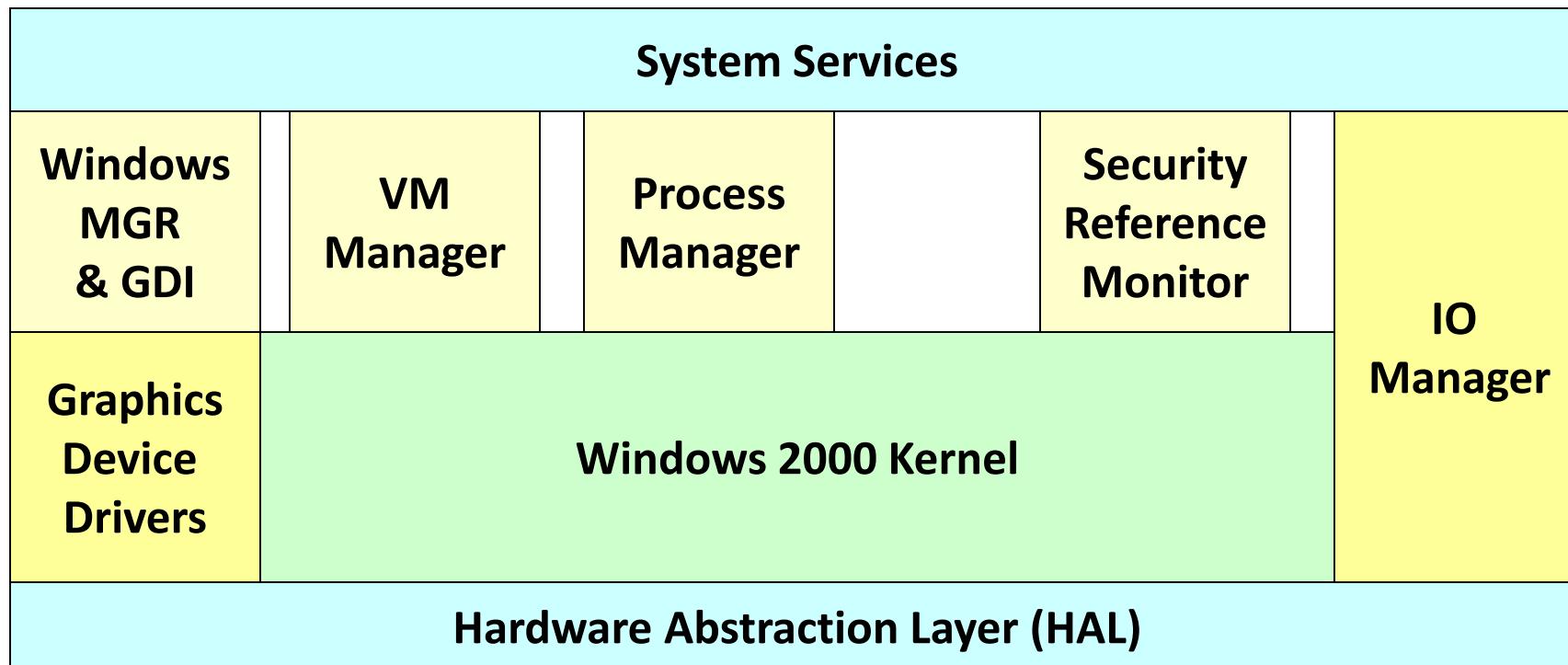
Hierarchical OS Model

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

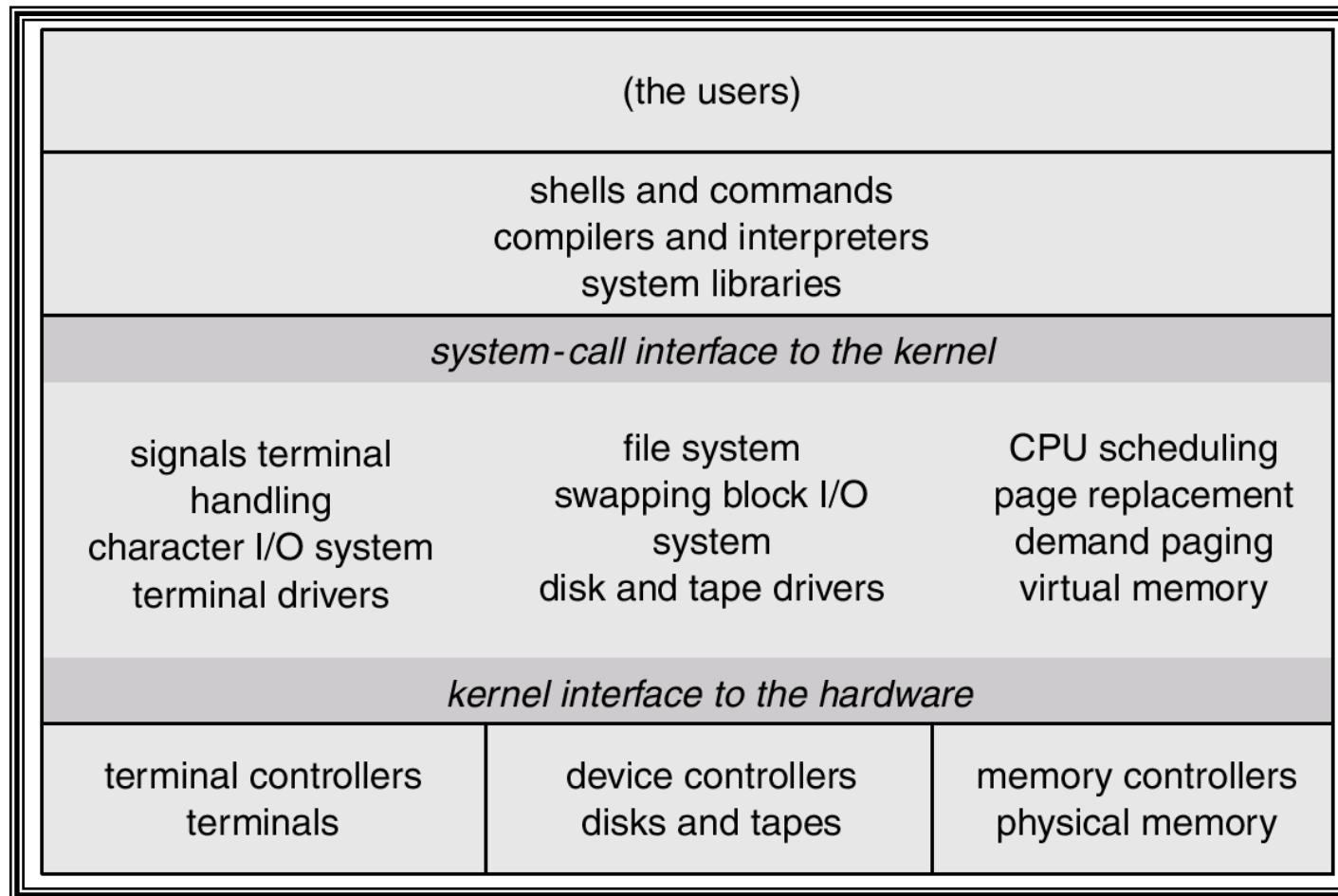
Example - Windows 2000

MGR – ManaGeR

GDI – Graphical Data Interface



Unix



Microkernels

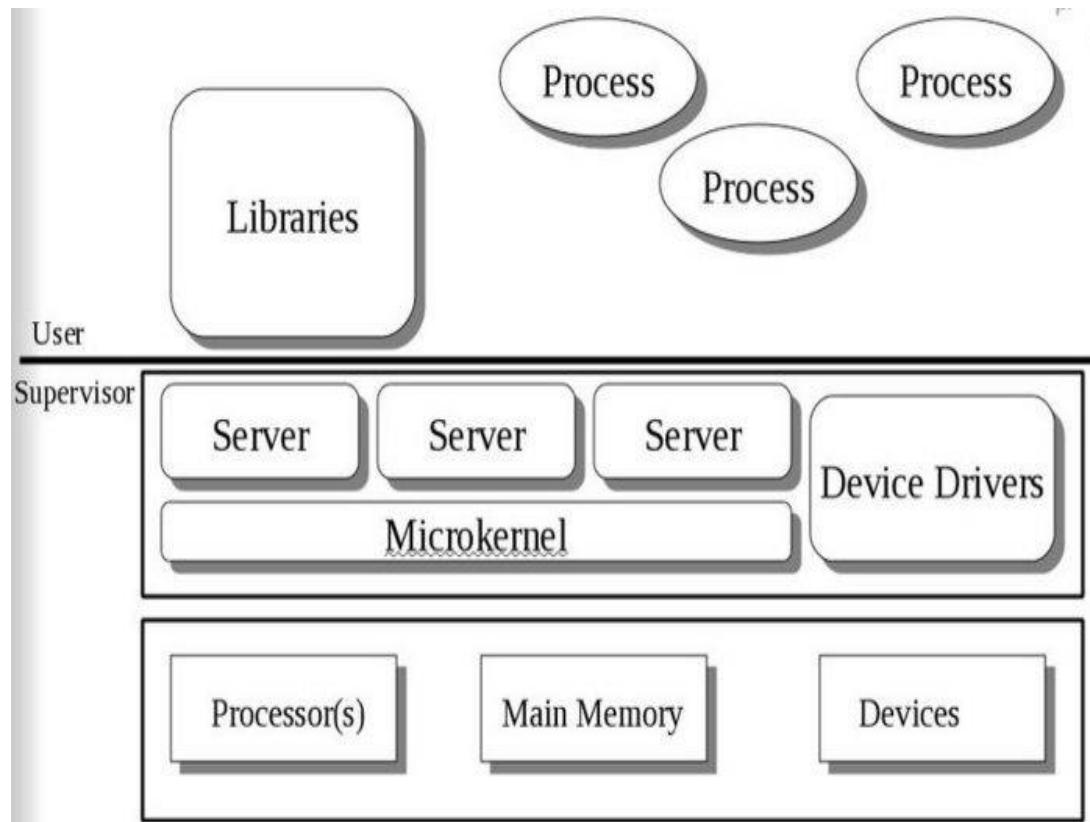
This structures the operating system by removing all nonessential portions of the kernel and implementing them as system and user level programs.

- 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.

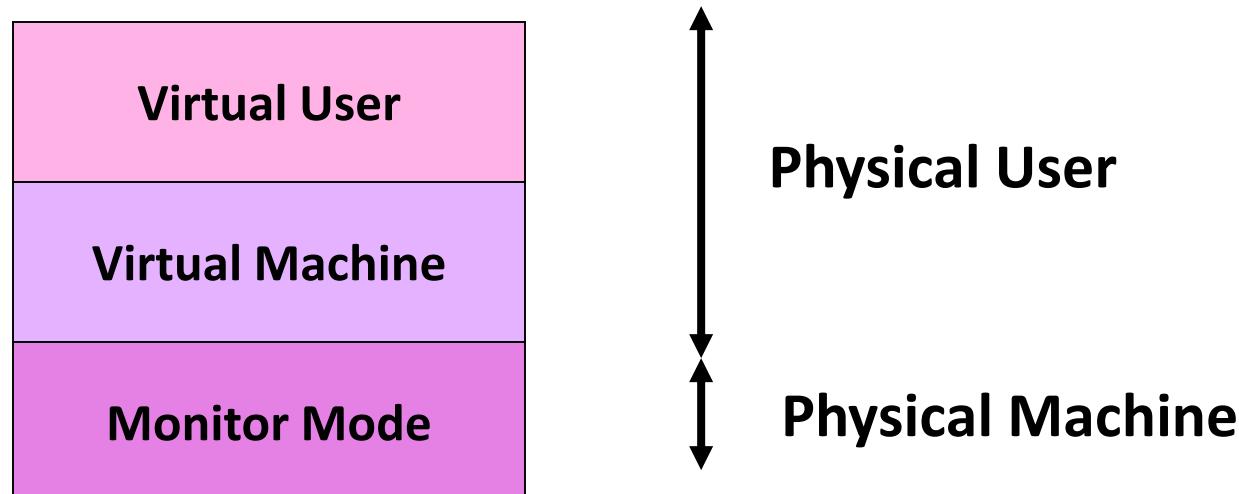
Main *disadvantage* is poor performance due to increased system overhead from message passing.



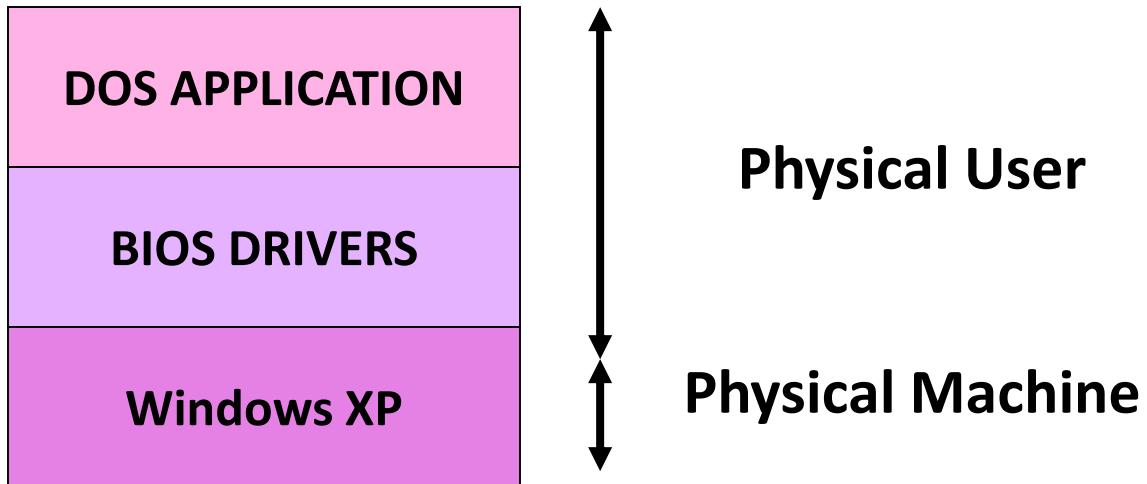
Virtual Machine

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

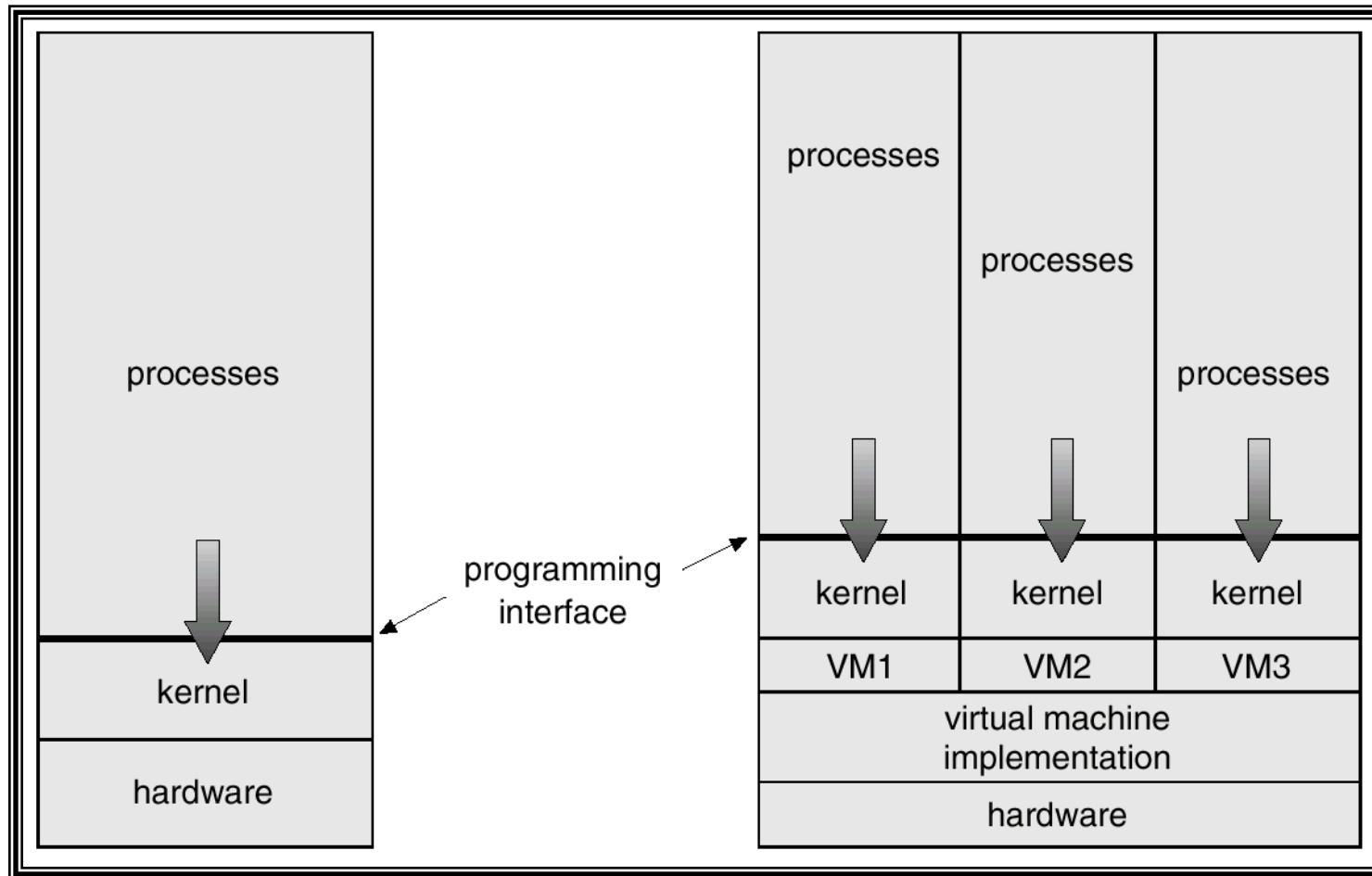
- The resources of the physical machine are shared. Virtual devices are sliced out of the physical ones. Virtual disks are subsets of physical ones.
- Useful for running different OS simultaneously on the same machine.
- Protection is excellent, but no sharing possible.
- Virtual privileged instructions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



Example of MS-DOS on top of Windows XP.

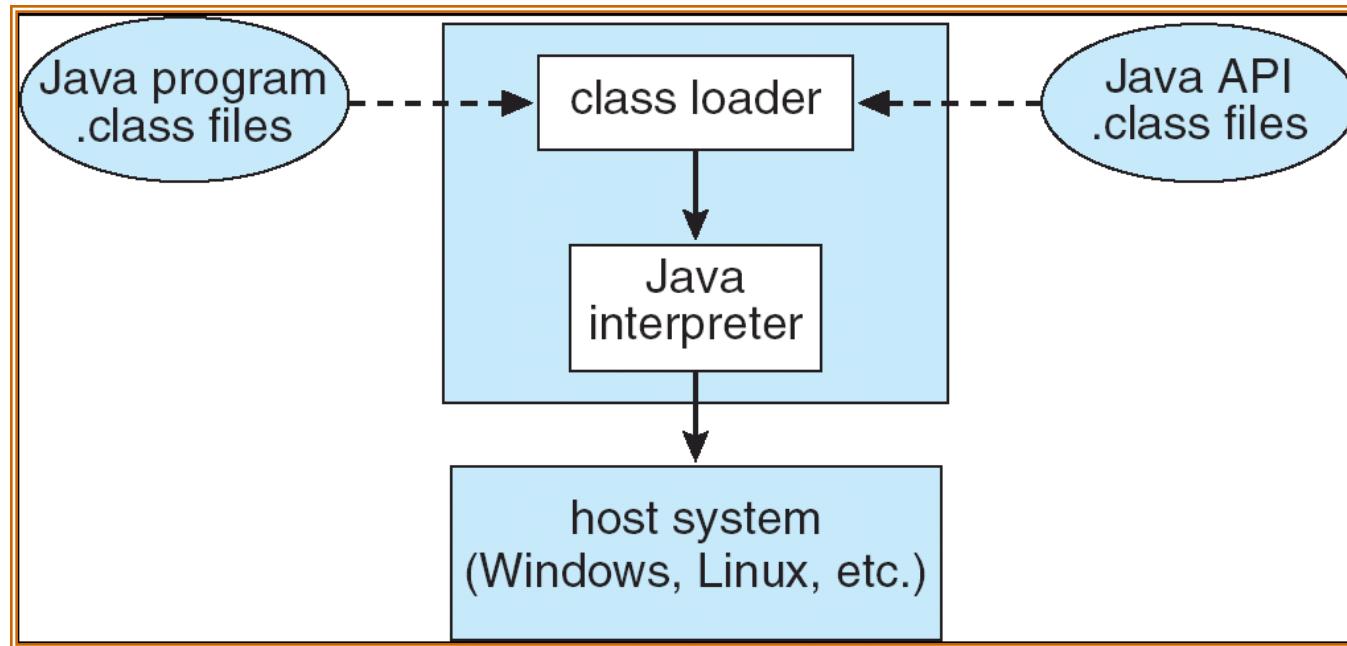


Virtual Machine



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

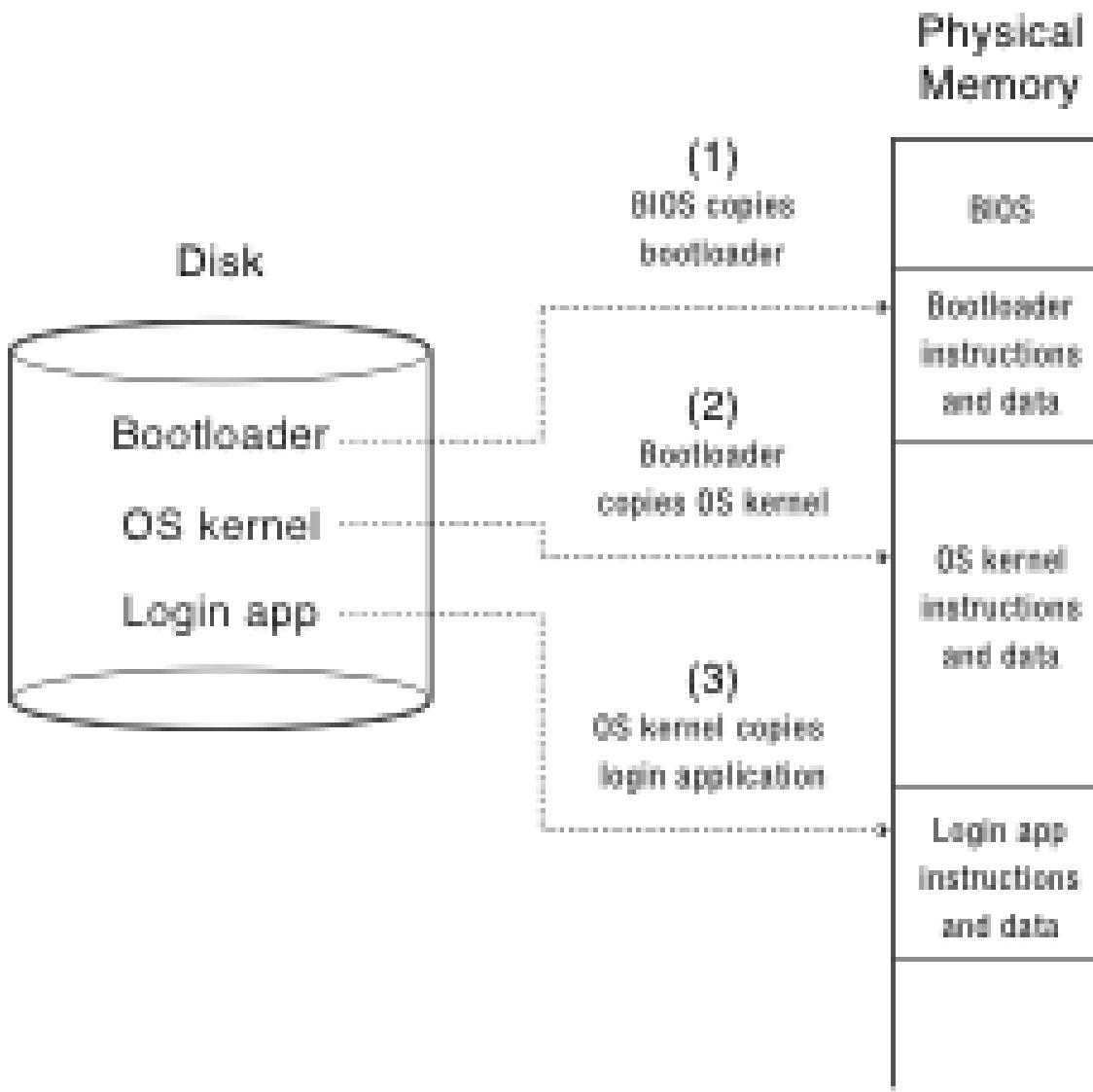
Initial Kernel Program

- 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

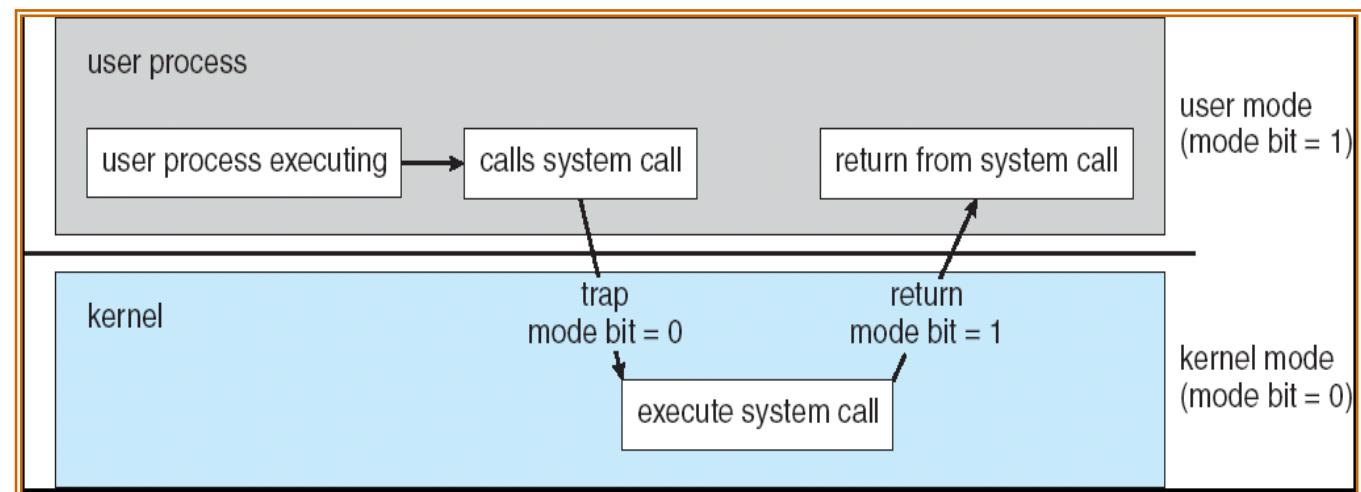
- Application programs included with the OS
- Highly trusted programs
- Perform useful work that most users need
 - listing and deleting files, configuring system
 - ls, rm, Windows Explorer and Control Panel
 - may include compilers and text editors
- Not part of the OS
 - run in user space
- Very useful

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

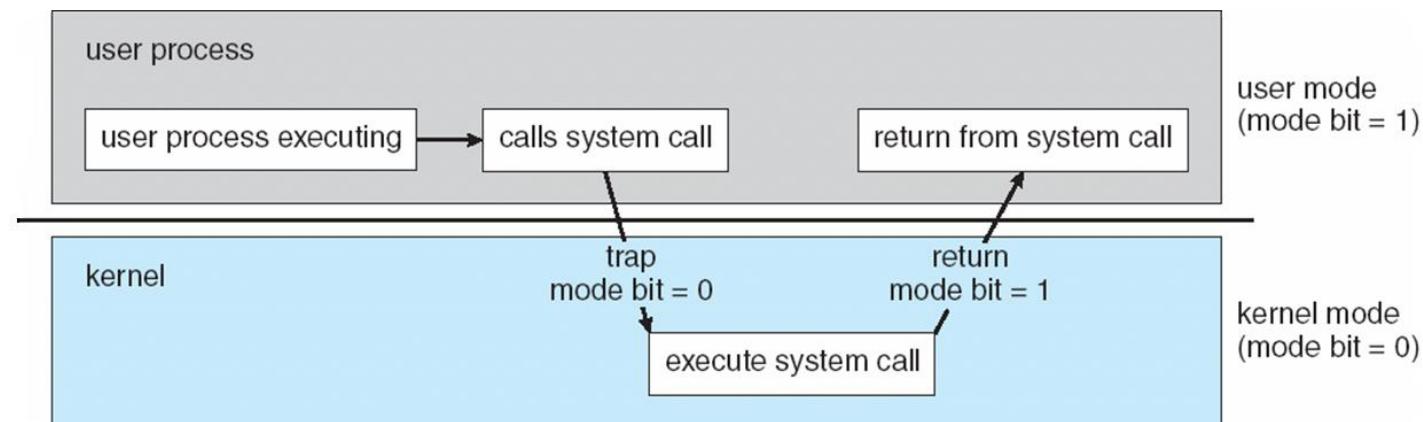
Dual Mode Operation

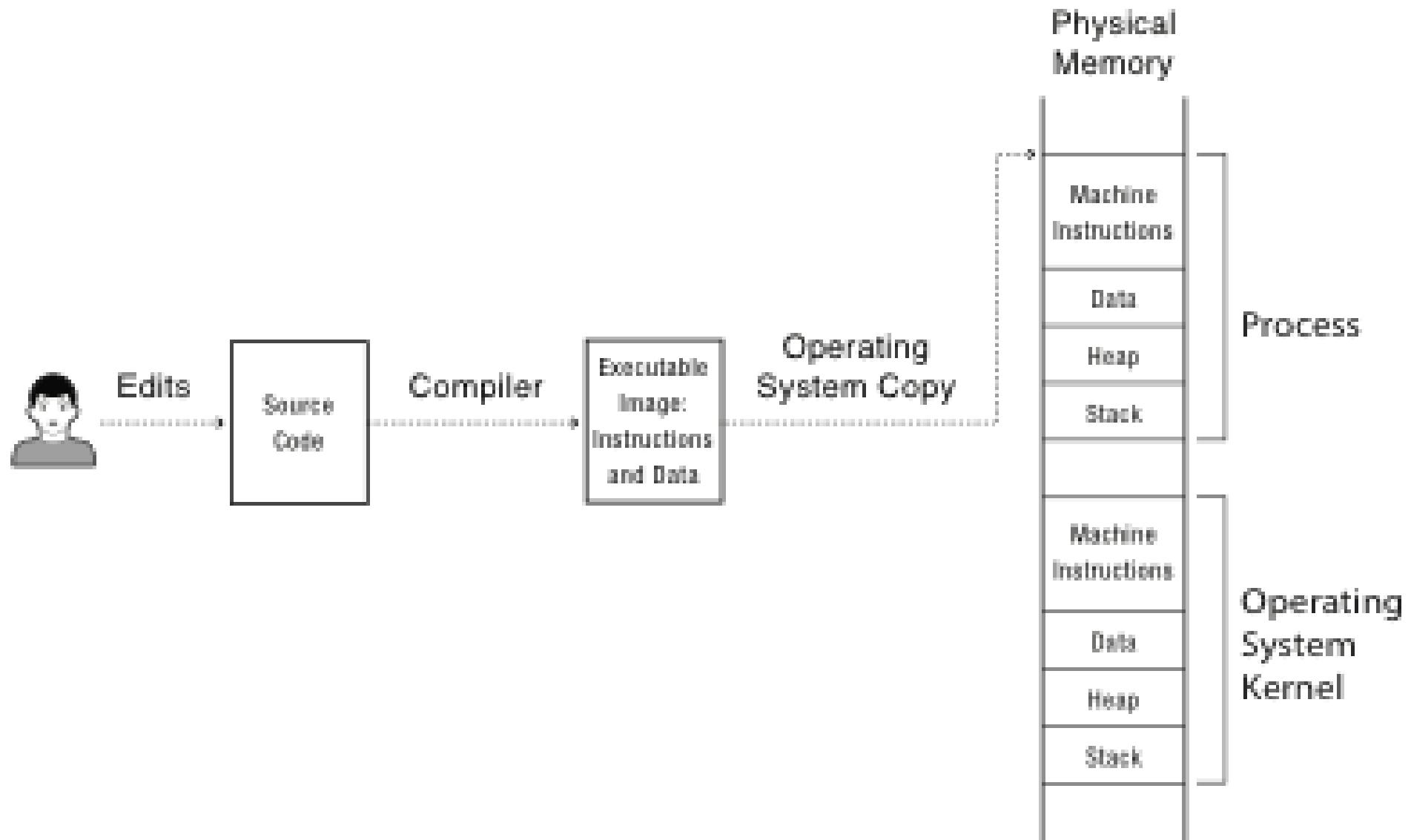
- Some instructions/ops prohibited in user mode:
 - Example: cannot modify page tables in user mode
 - Attempt to modify \Rightarrow Exception generated
- Transitions from user mode to kernel mode:
 - System Calls, Interrupts, Other exceptions



Transition from User to Kernel Mode

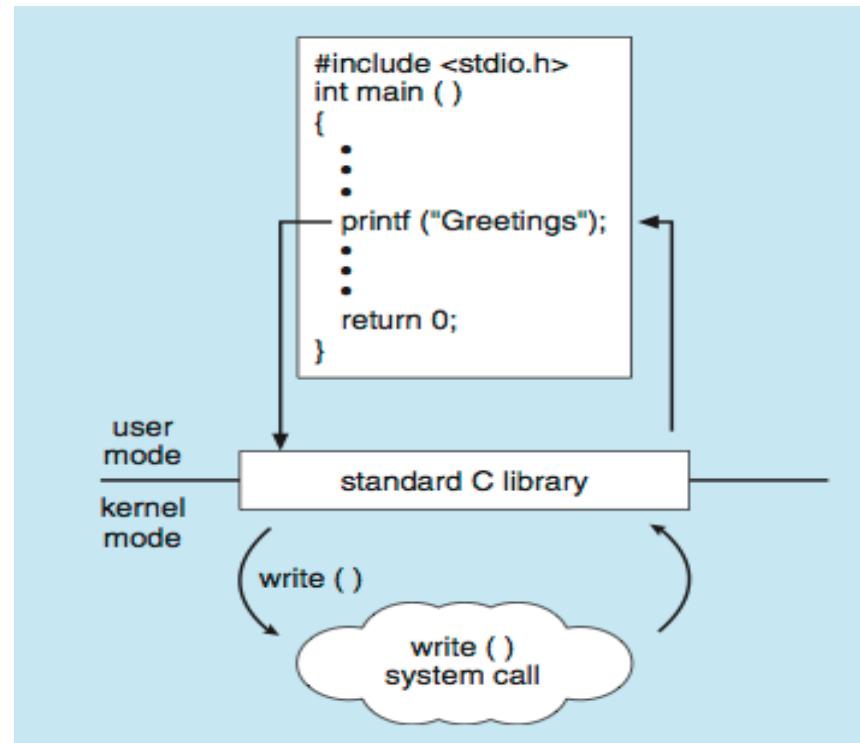
- Timer to prevent infinite loop / process hogging resources
 - Set interrupt after specific period
 - Operating system decrements counter
 - When counter zero generate an interrupt
 - Set up before scheduling process to regain control or terminate program that exceeds allotted time





Standard C Library Example

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



System Calls

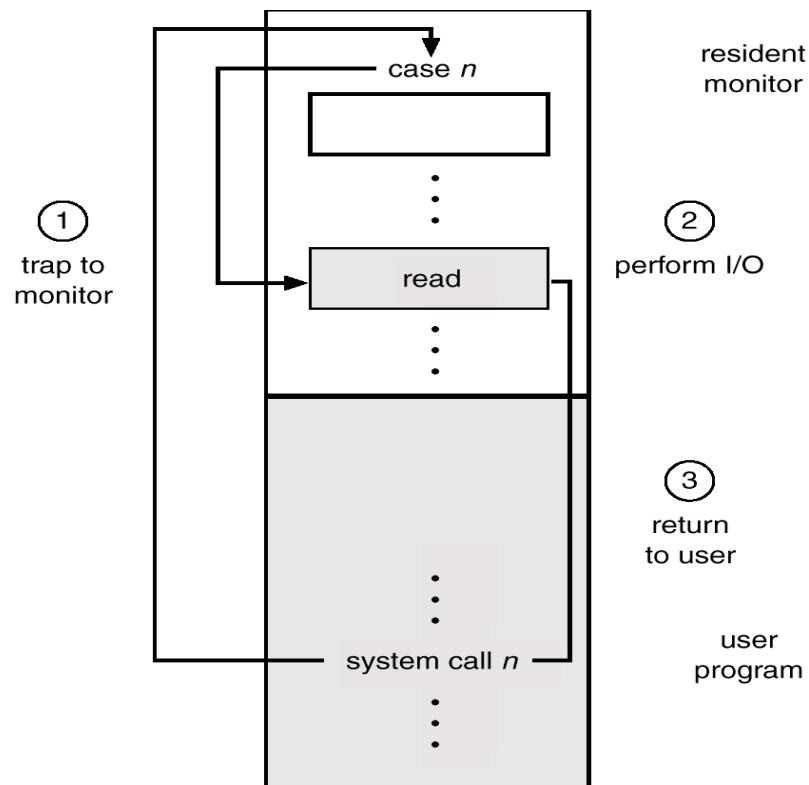
- System calls provide the interface between a running program and the operating system.
 - Generally available as assembly-language instructions.
 - Languages have been defined to replace assembly language for systems programming; allow system calls to be made directly (e.g., C, C++)
- Three general 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

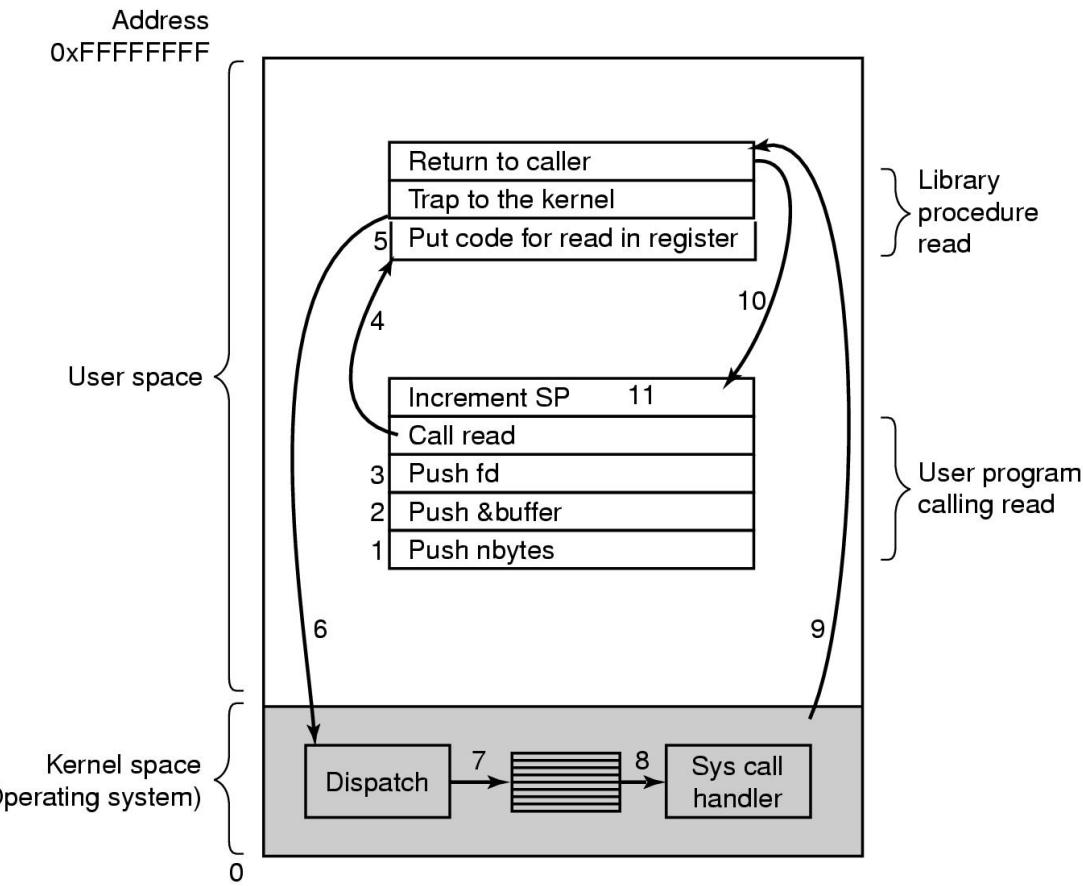
- A System Call is the main way a user program interacts with the Operating System.



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

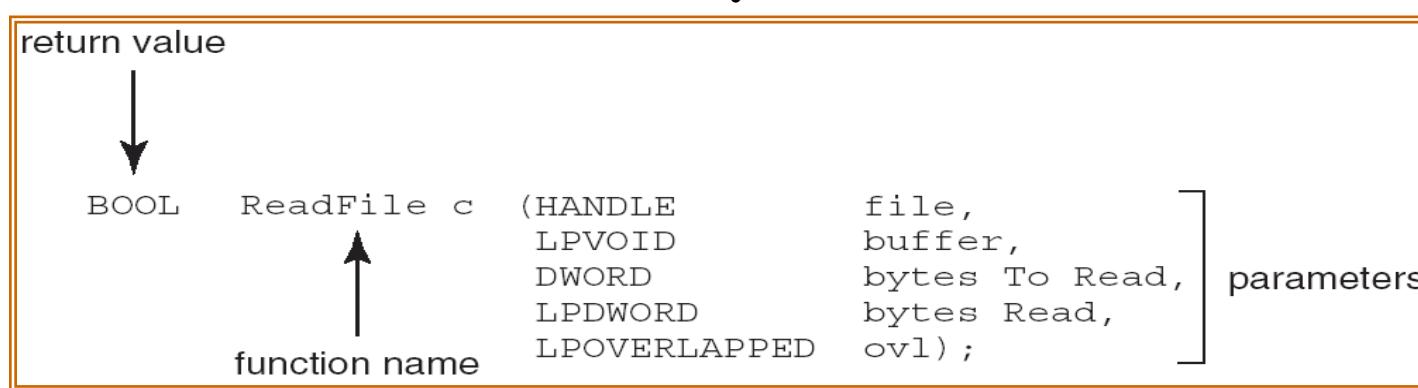
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `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`.

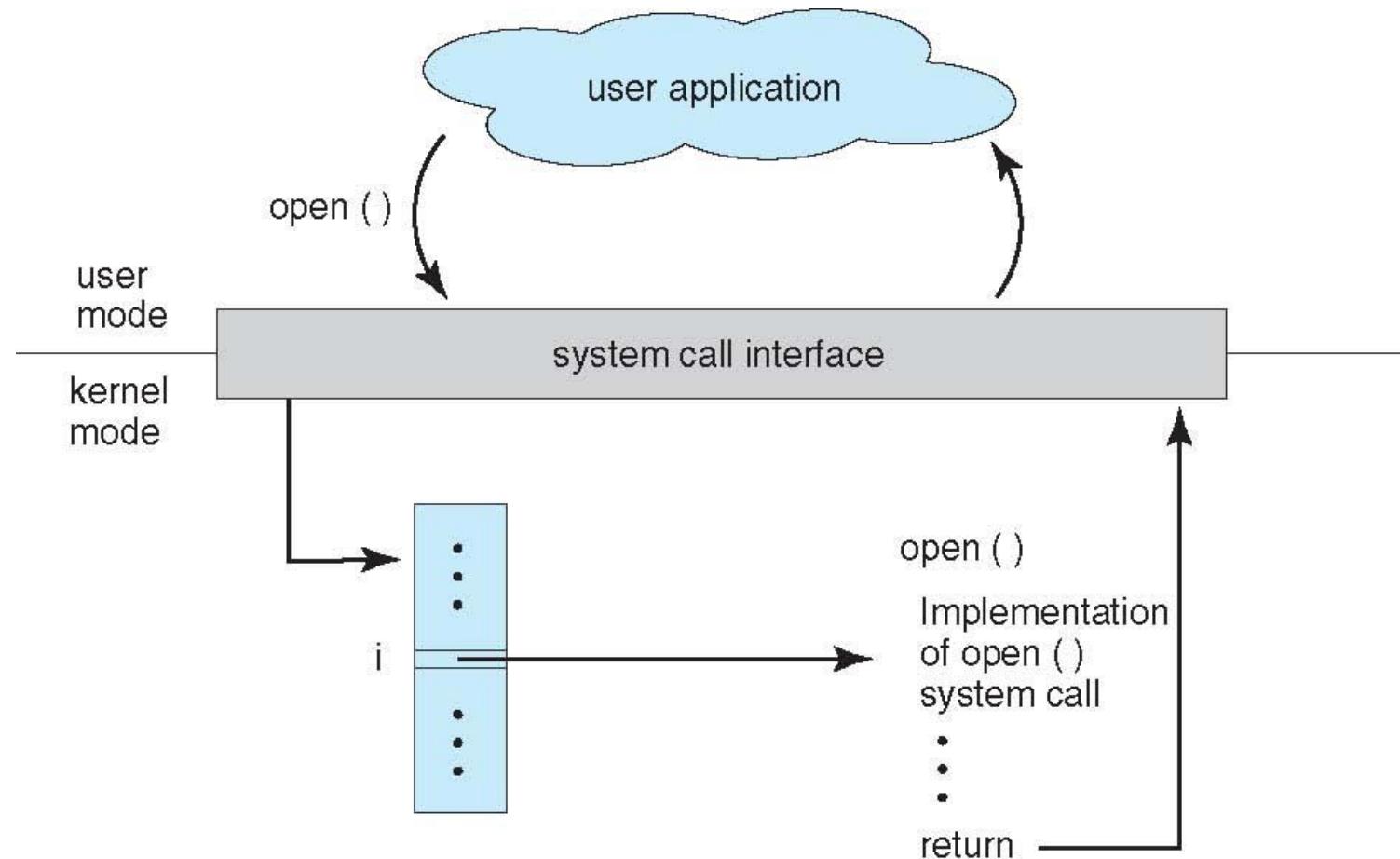
Example of Windows API

- 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

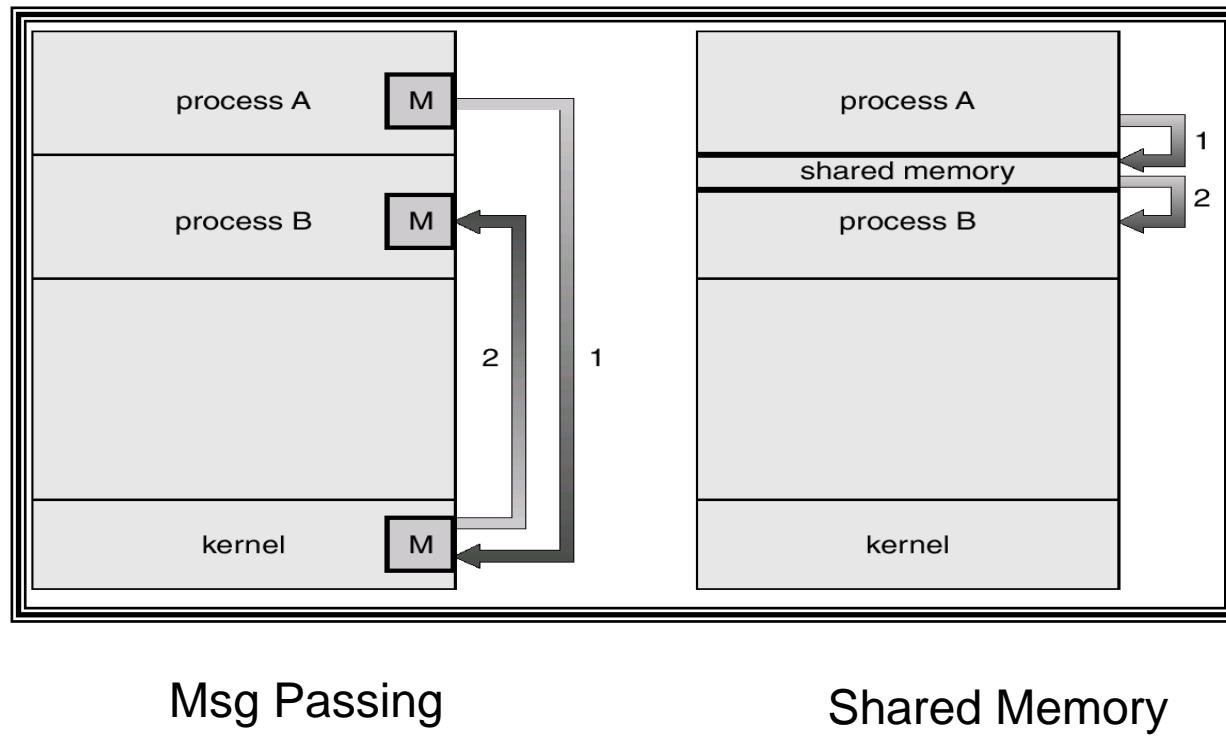
API – System Call – OS Relationship



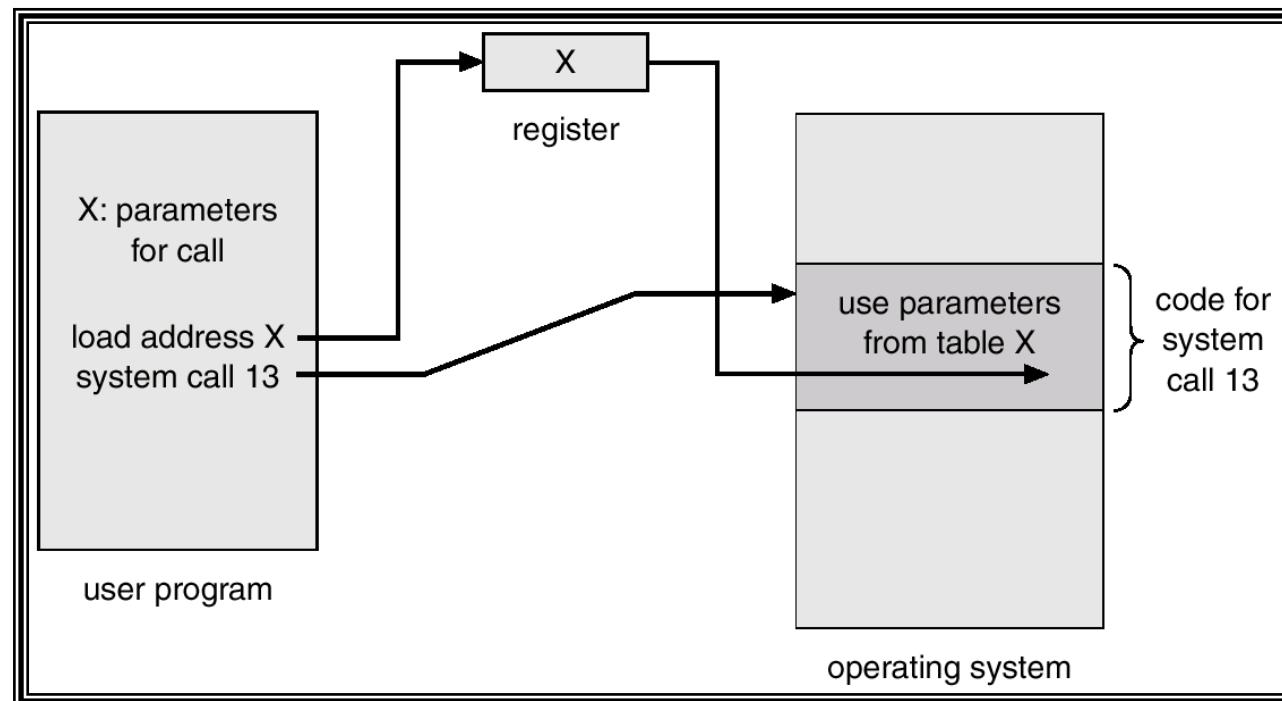
System Call Parameter Passing

- 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

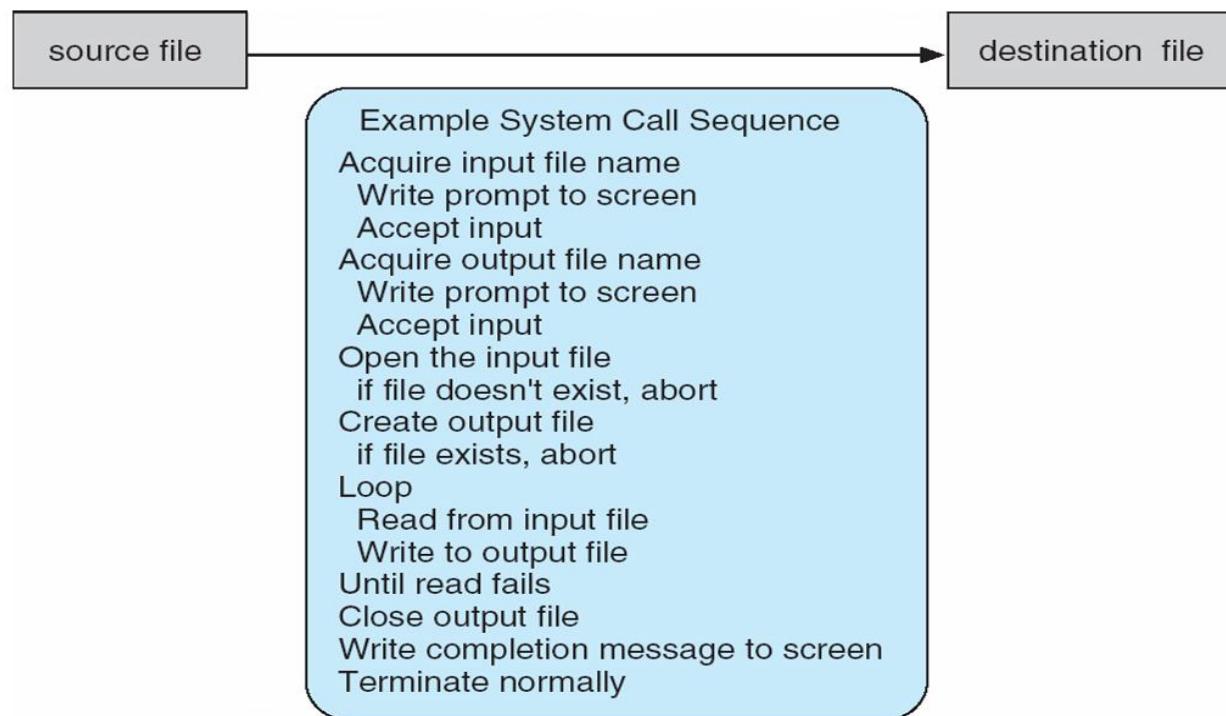


Passing of Parameters As A Table



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,

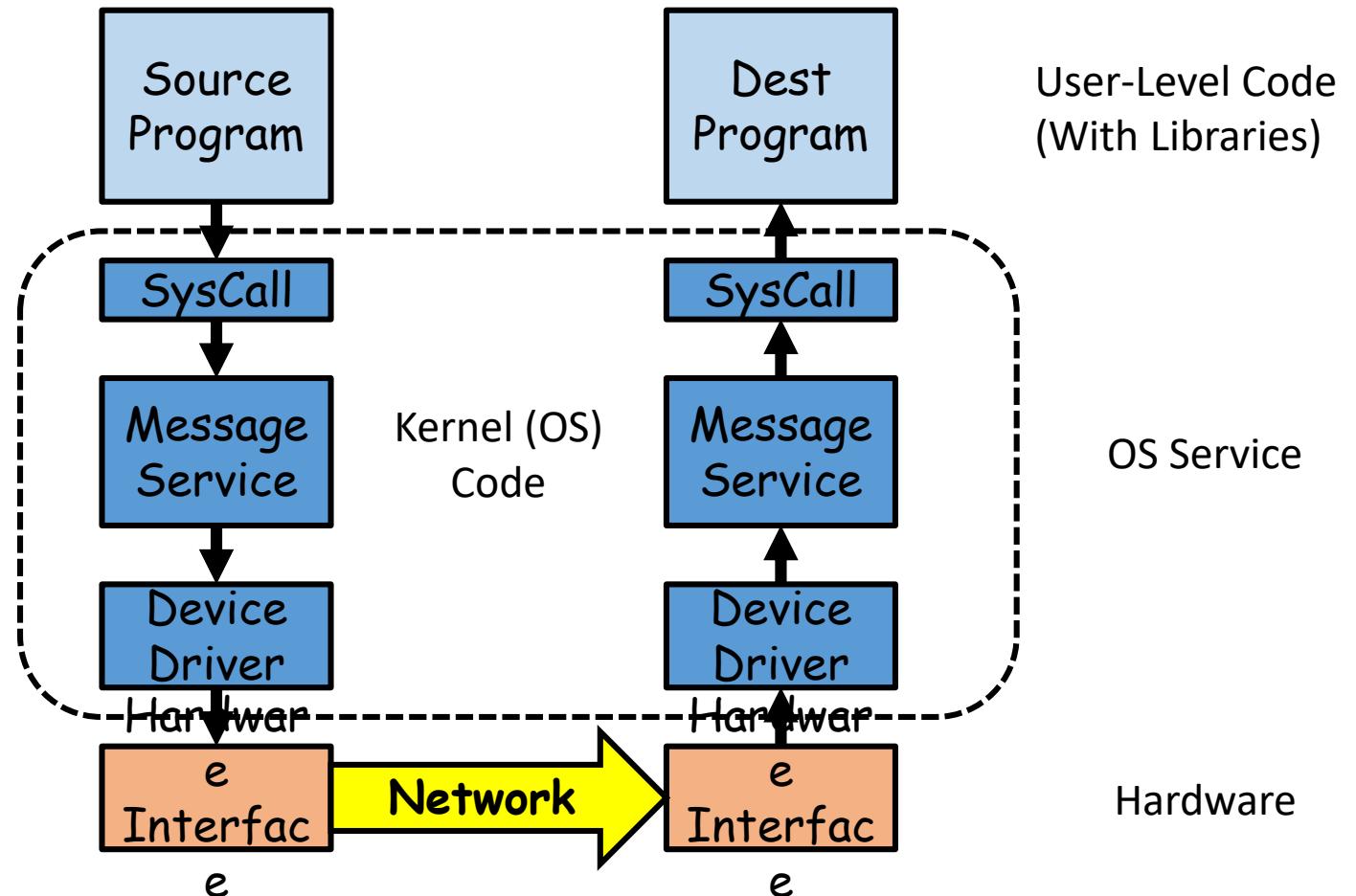
Handle interrupts

- Prevent Denial Of Service (DOS)

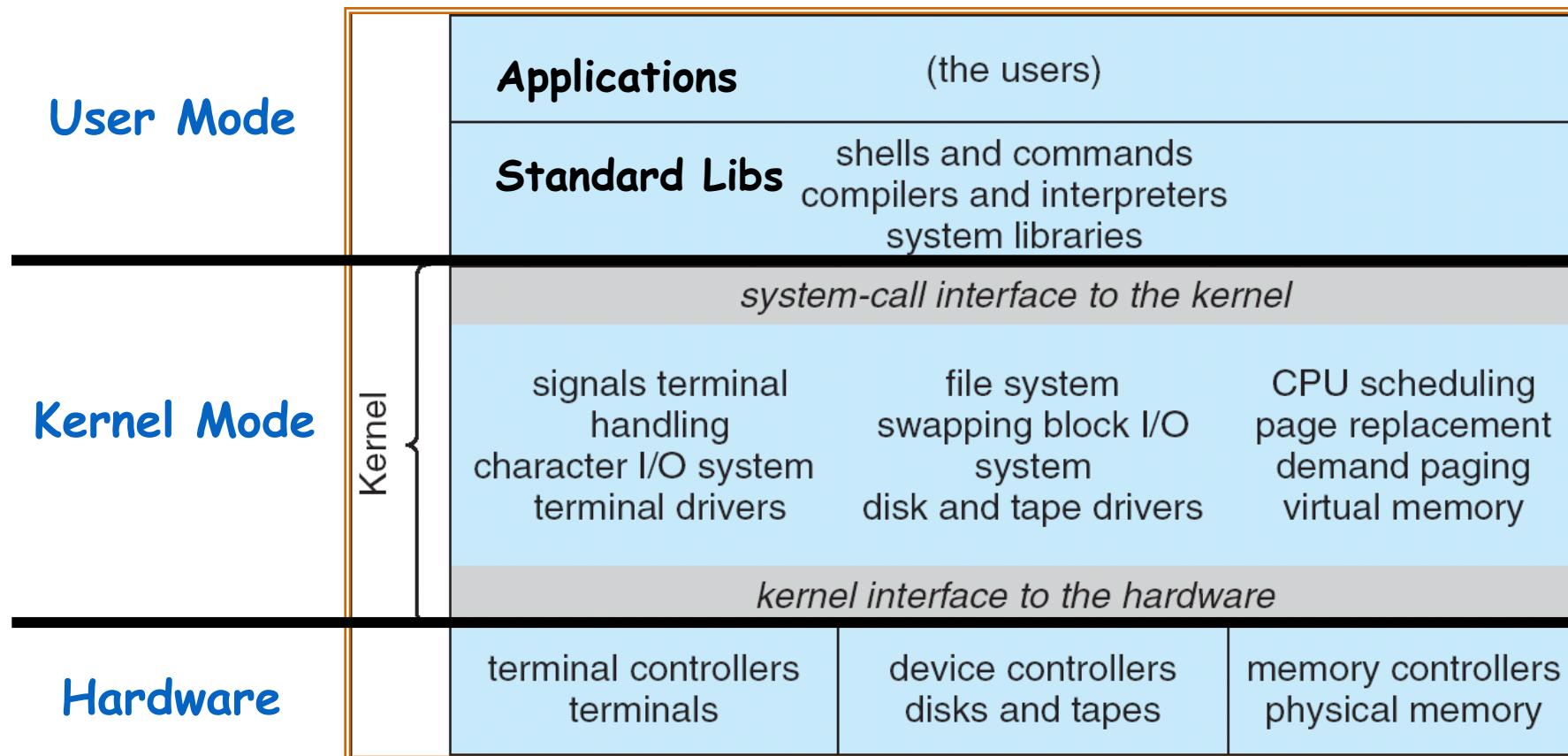
and/or Deadlock

Traditional Approach:

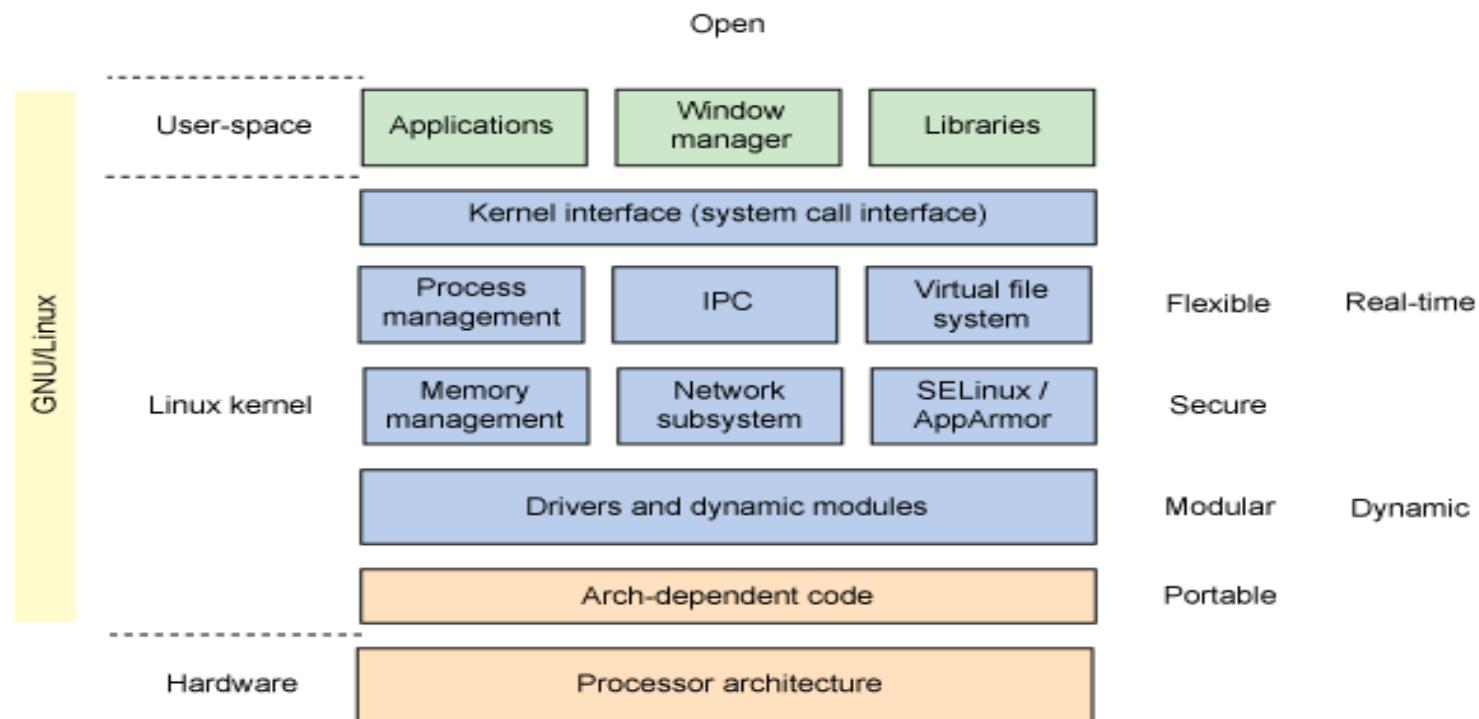
Use a system call + OS Service



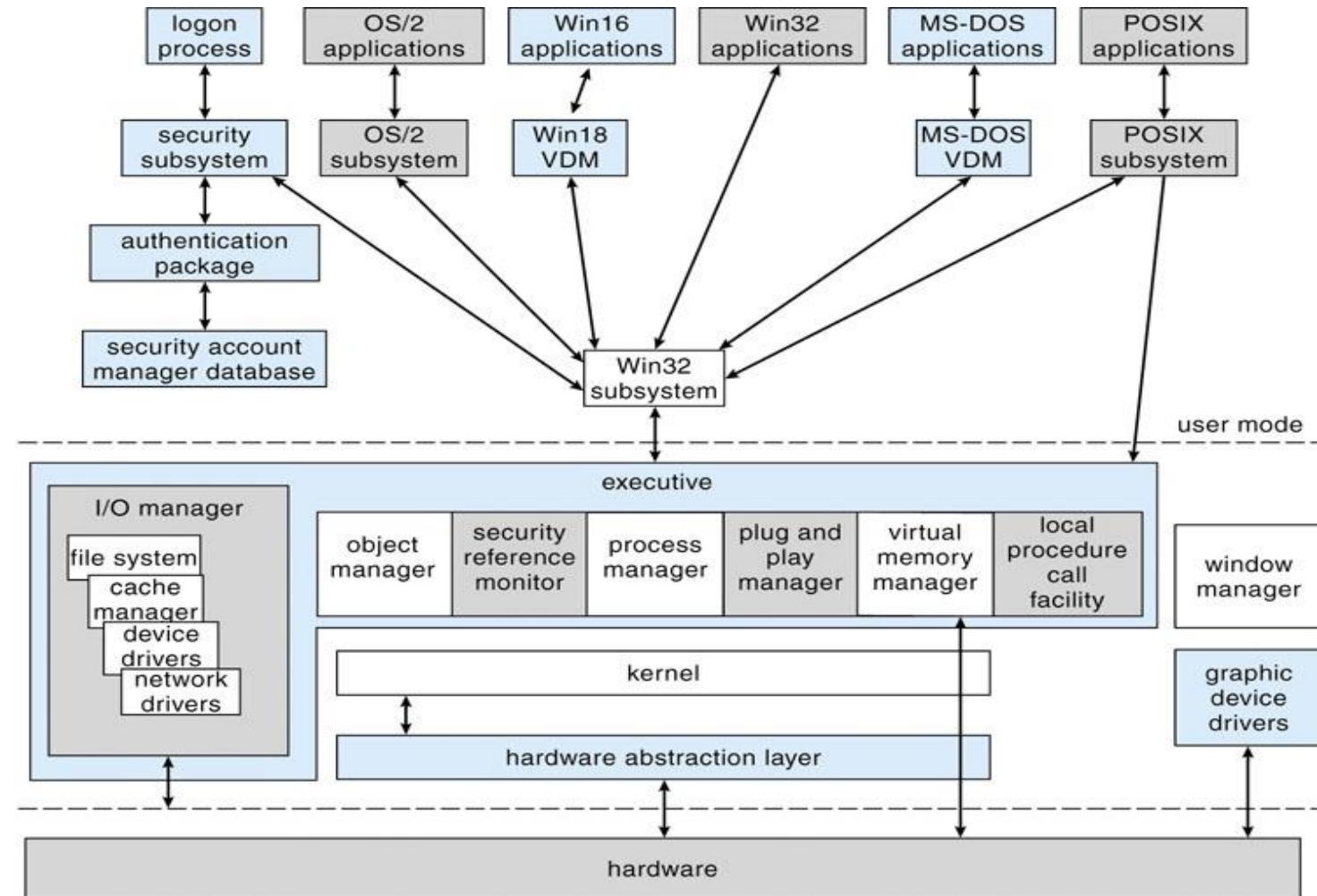
UNIX System Structure



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()

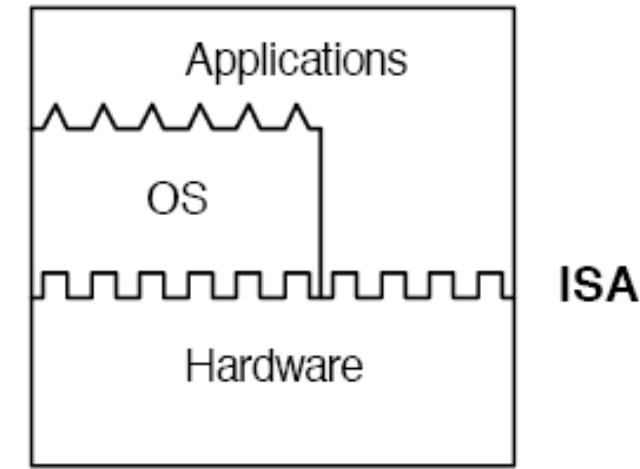
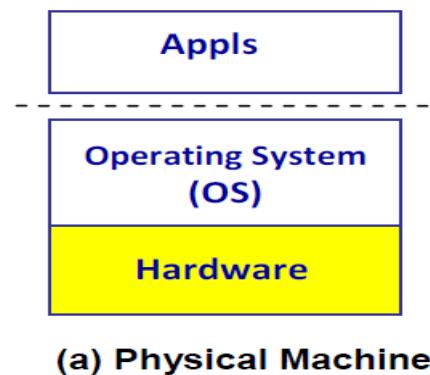
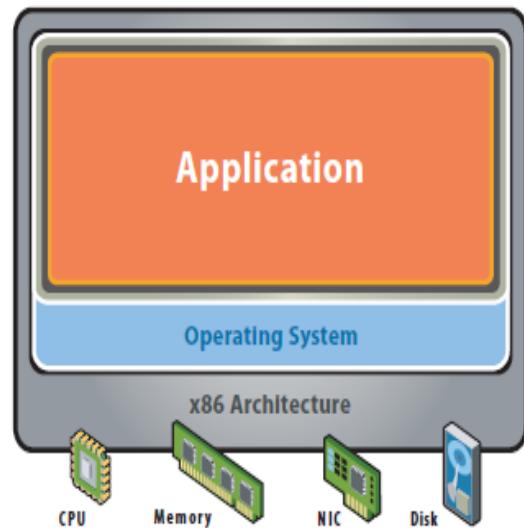
UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	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)



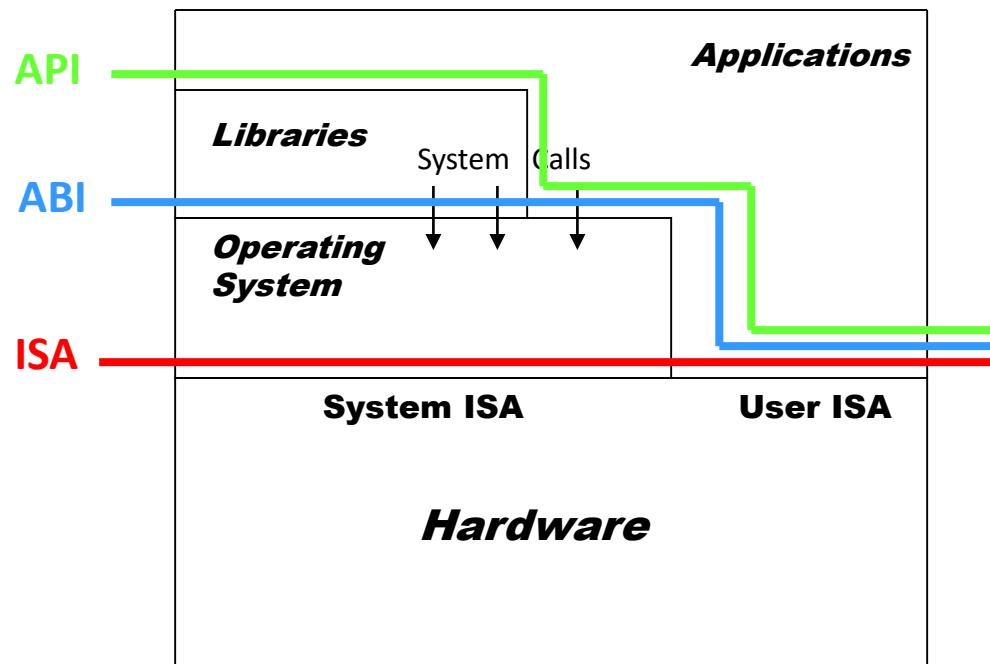
- Single OS image per machine
- Software and hardware tightly coupled
- Running multiple applications on same machine
- Underutilized resources
- Inflexible and costly infrastructure

- 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

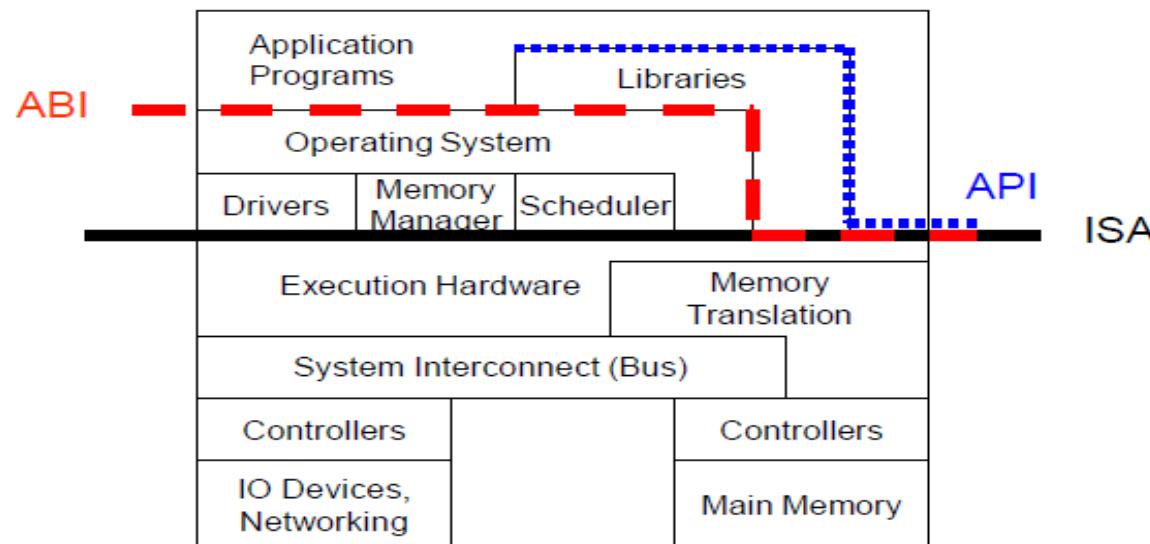
Architecture: formal specification of a system's interface and the logical behavior of its visible resources.



- **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.

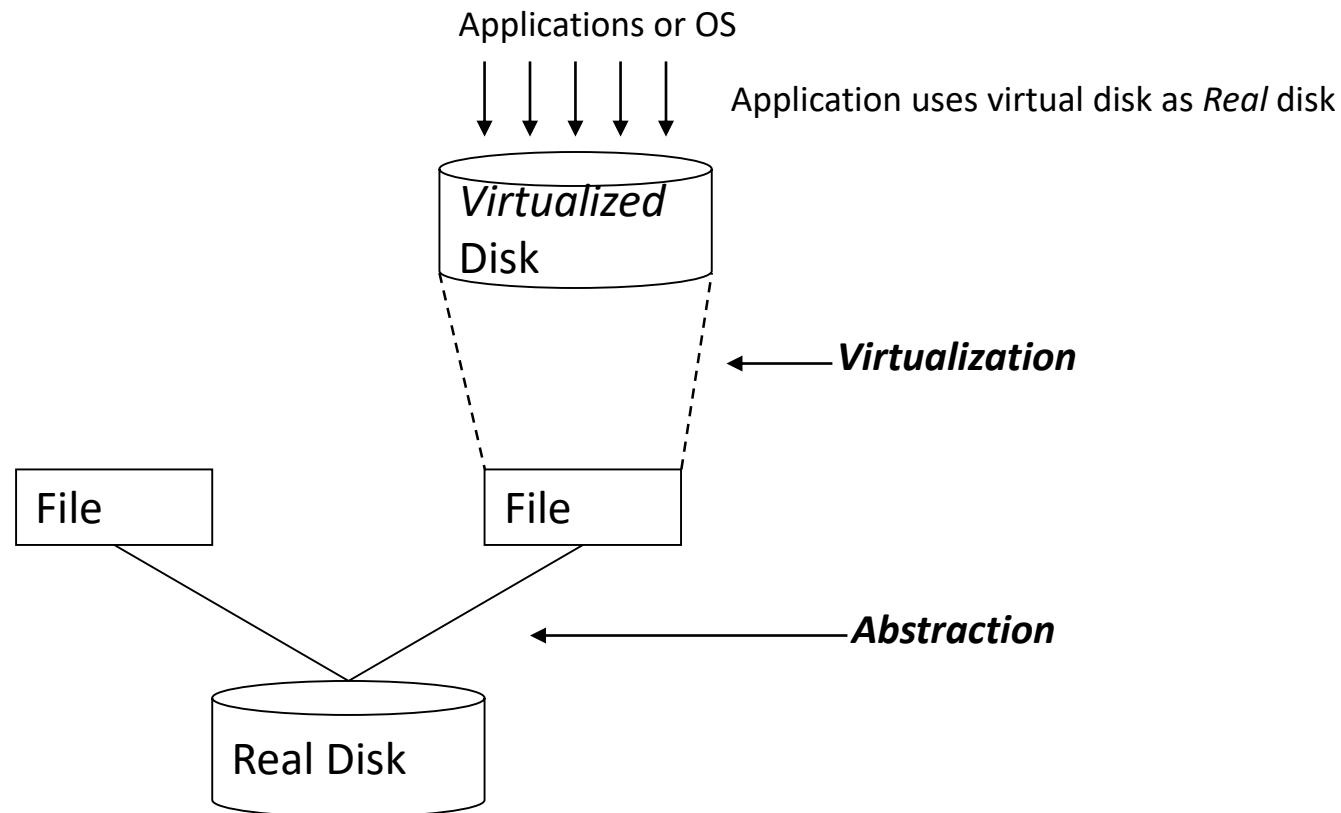
What is virtualization?

- **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.

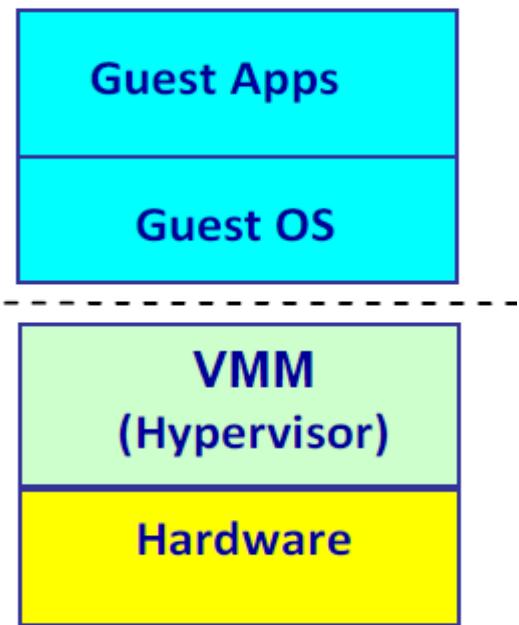
- The resources of the physical computer are shared to create the virtual machines.
 - CPU scheduling can create the appearance that users have their own processor.
 - Spooling (simultaneous peripheral operations online) and a file system can provide virtual card readers and virtual line printers.
 -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

VM types

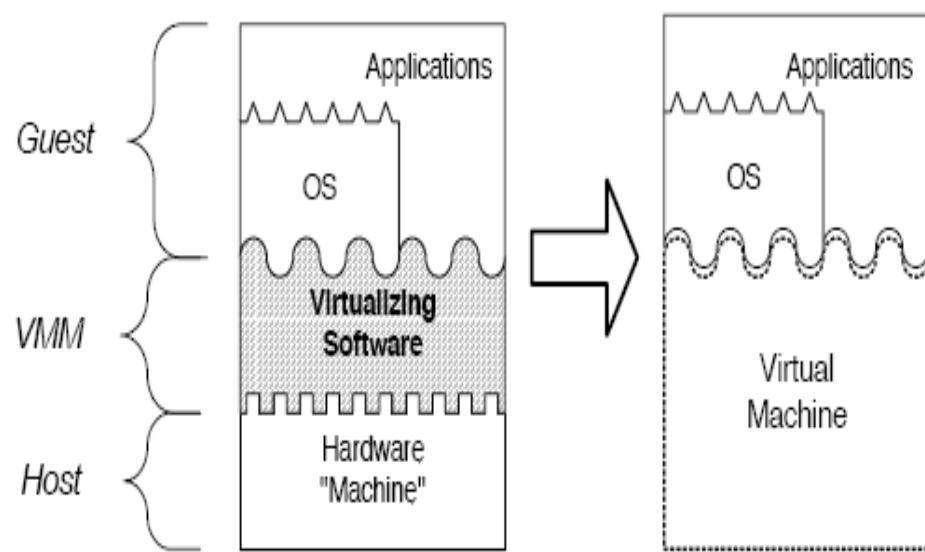
- Native VM
- Hosted VM
- Dual mode VM

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

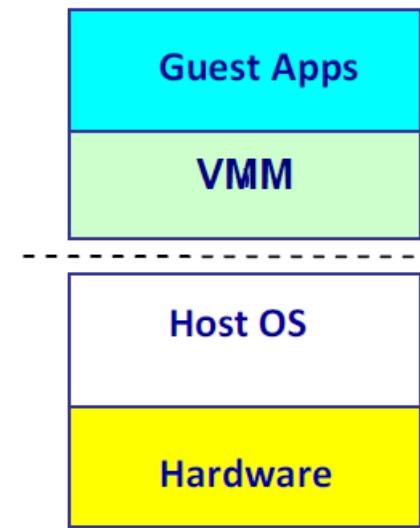


- Figure b shows a native VM installed with the use a VMM called a *hypervisor at the privileged mode*.
- For example, the hardware has a x-86 architecture running the Windows system. The guest OS could be a Linux system and the hypervisor is the XEN system developed at Cambridge University.
- This hypervisor 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

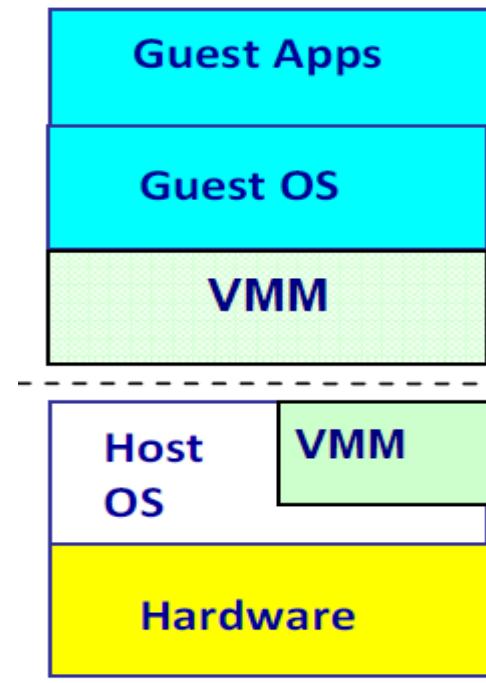
Another architecture is the host VM shown in Fig. (c). Here the VMM runs with a non-privileged mode. The host OS need not be modified.



(c) Hosted VM

Dual Mode VM

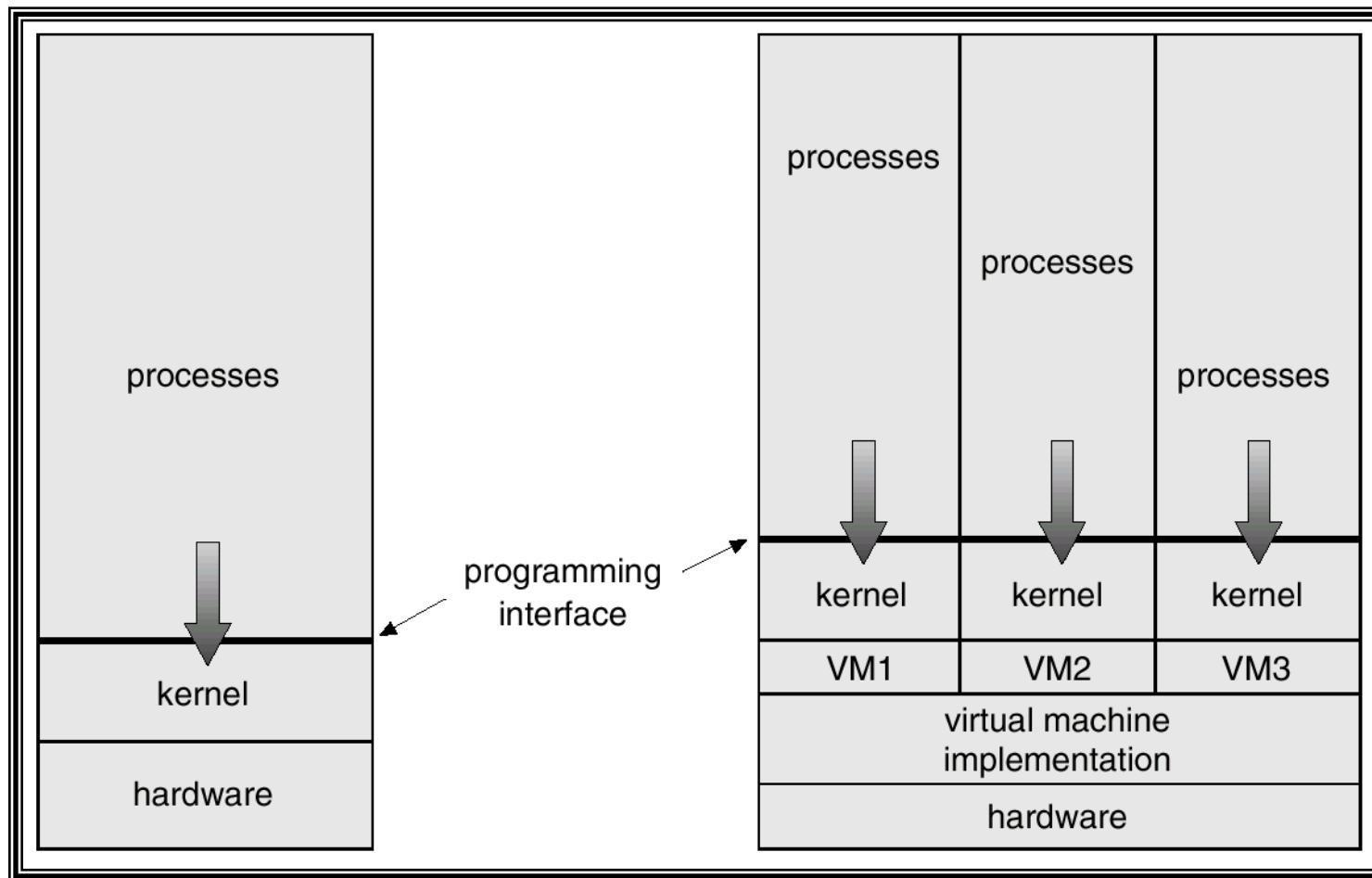
The VM can be also implemented with a dual mode as shown in Fig. (d).



(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

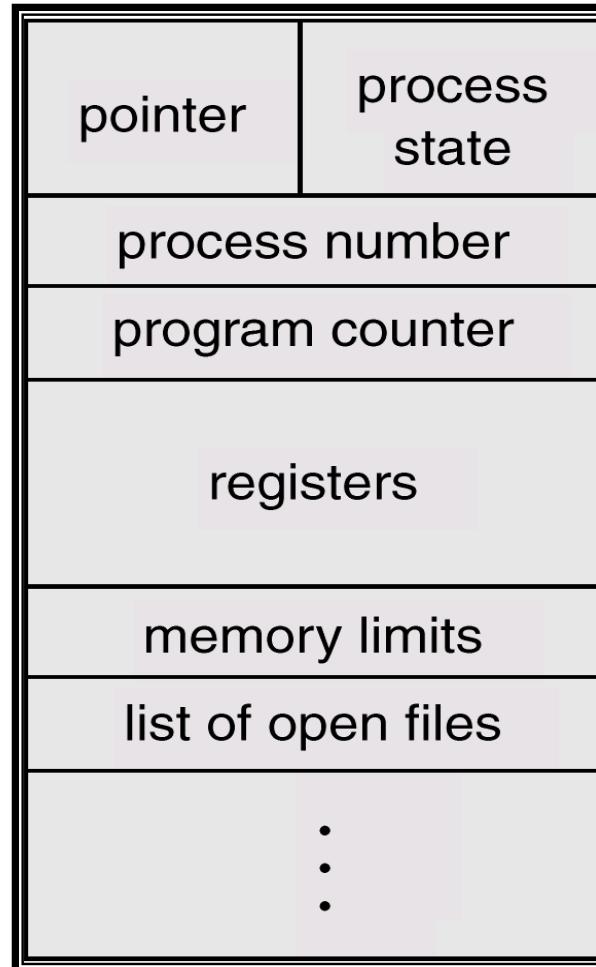
Process Control Block (PCB)

- 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.

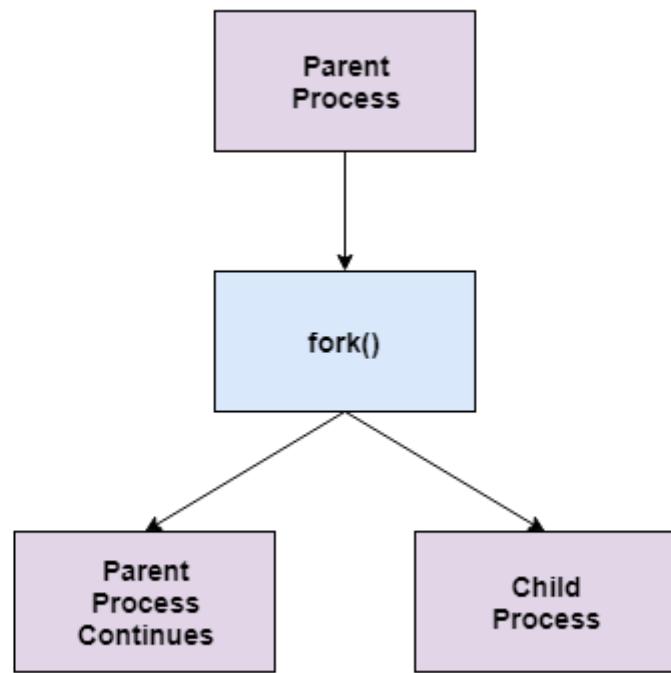
- In Operating System, the fork() system call is used by a process to create another process. The process that used the fork() system call is the parent process and process consequently created is known as the child process.

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.

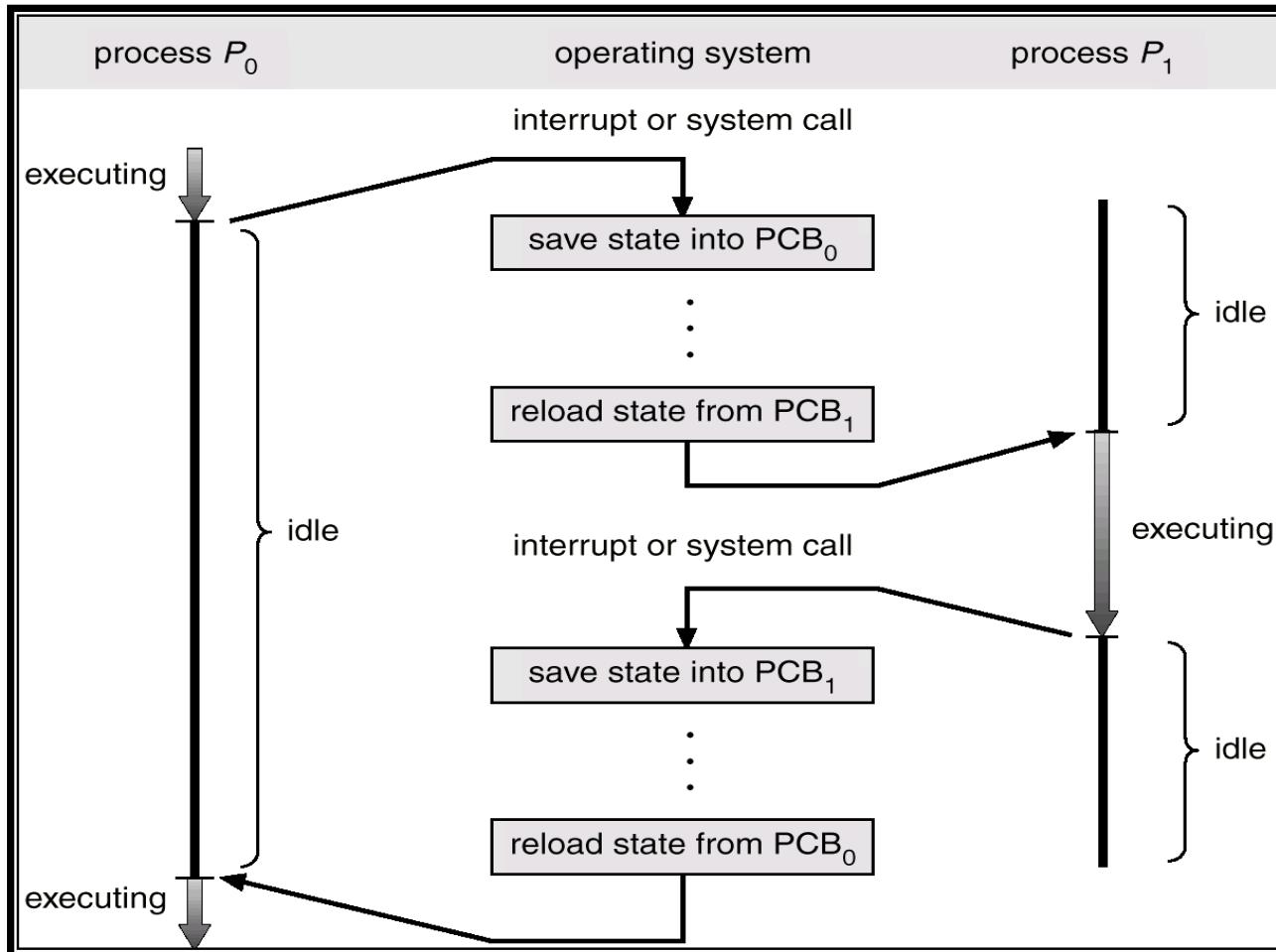
Static and Dynamic Process:

- 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

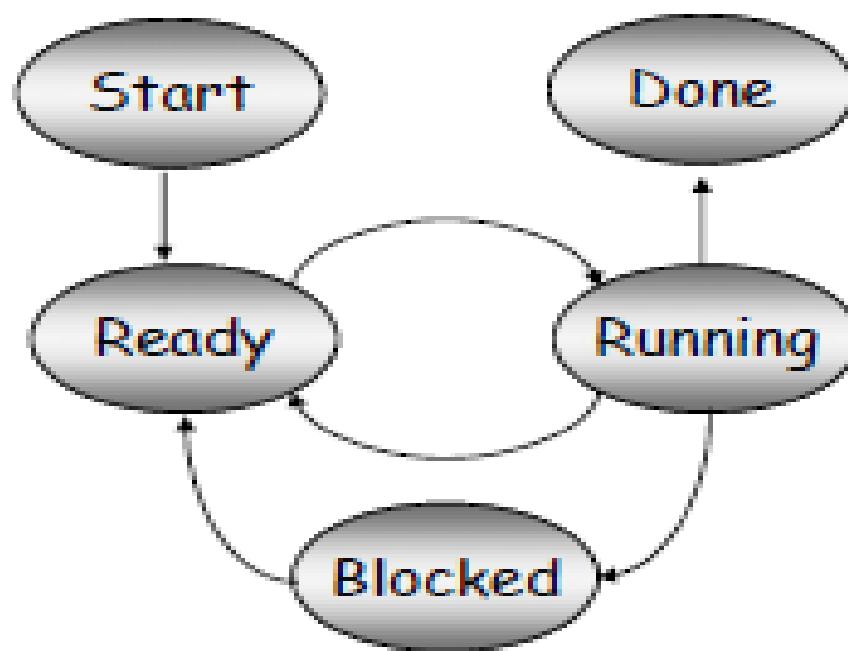


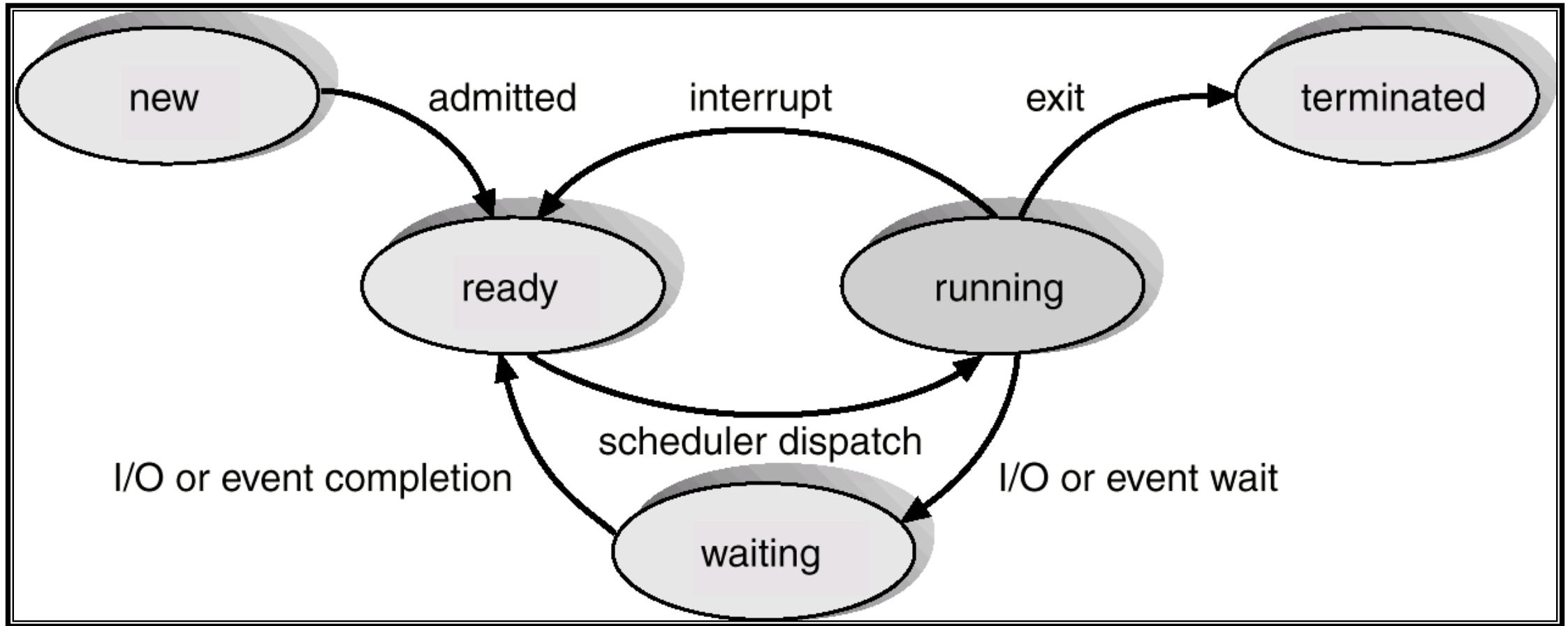
Life Cycle of a Process

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.

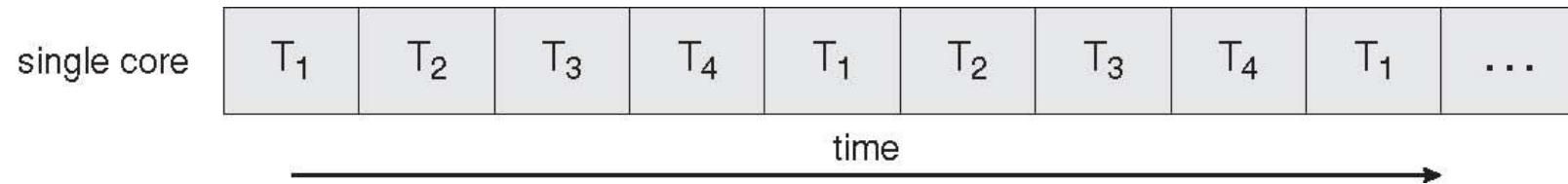
Thread Concepts

- 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.

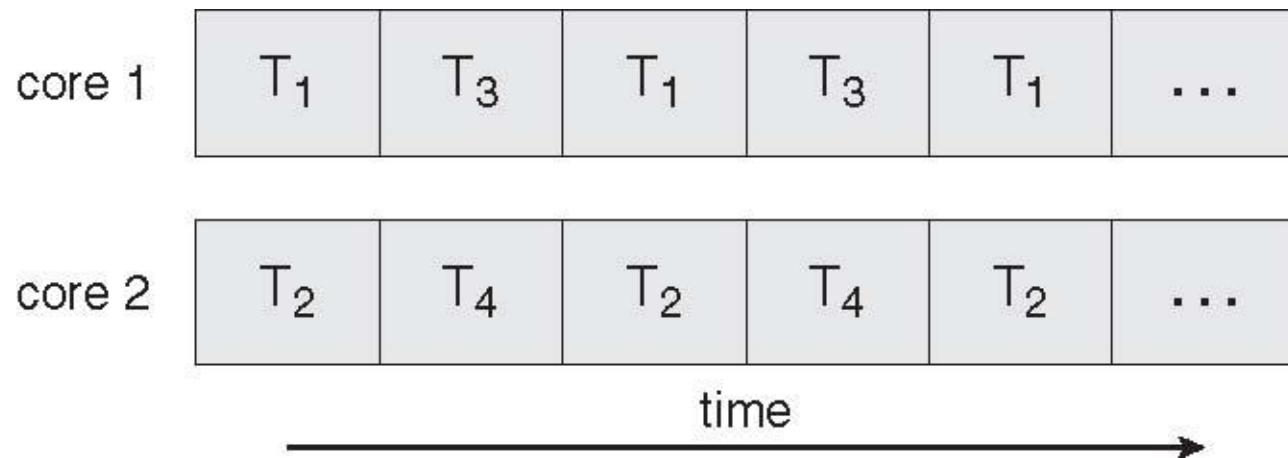
Multicore Programming

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

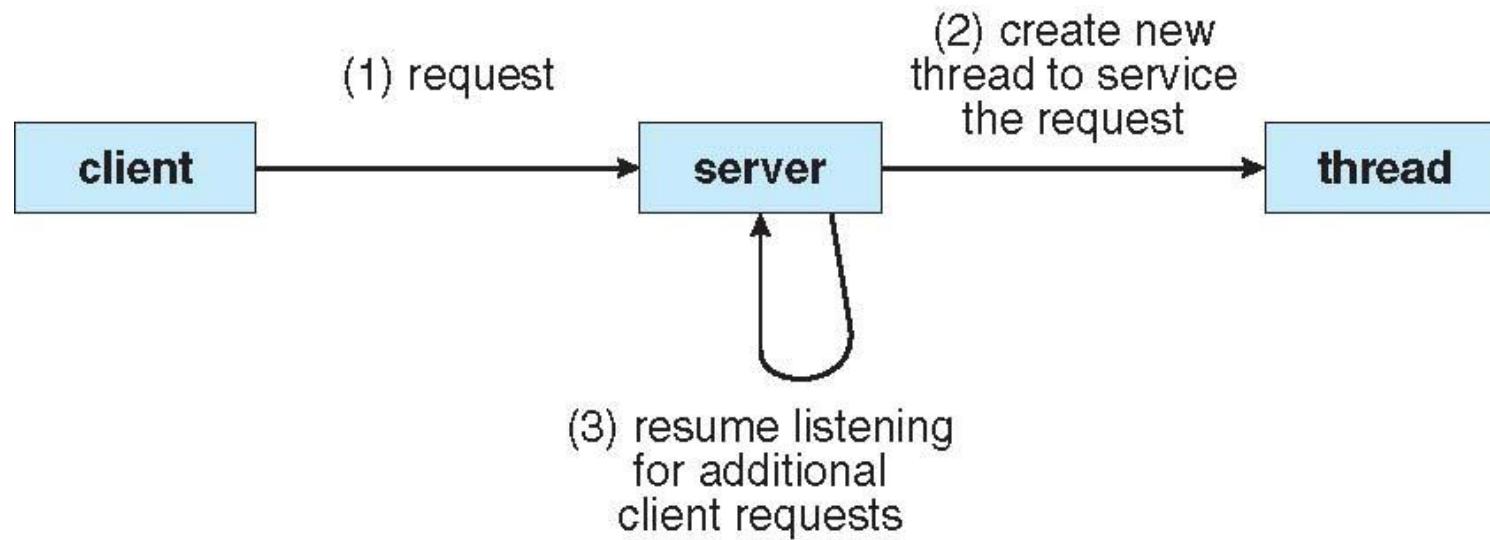
Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System

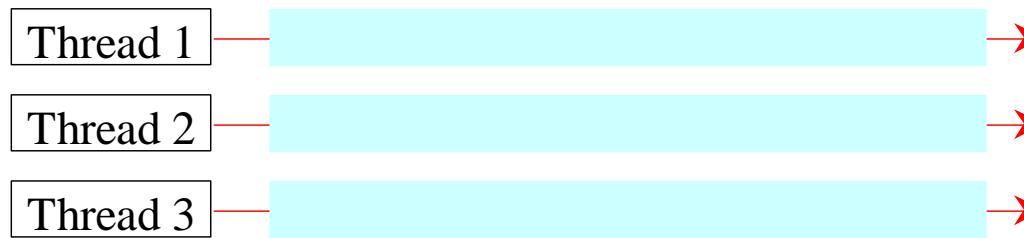


Multithreaded Server Architecture

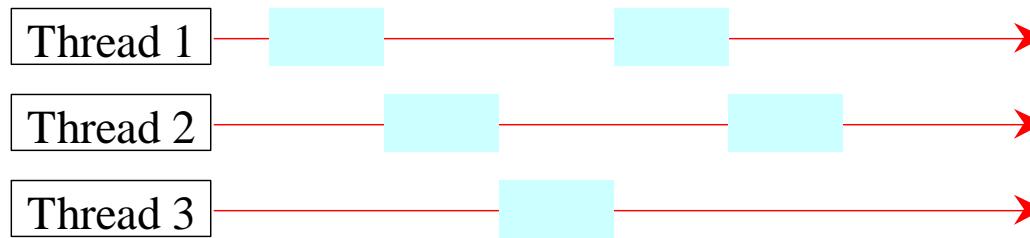


Threads Concept

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

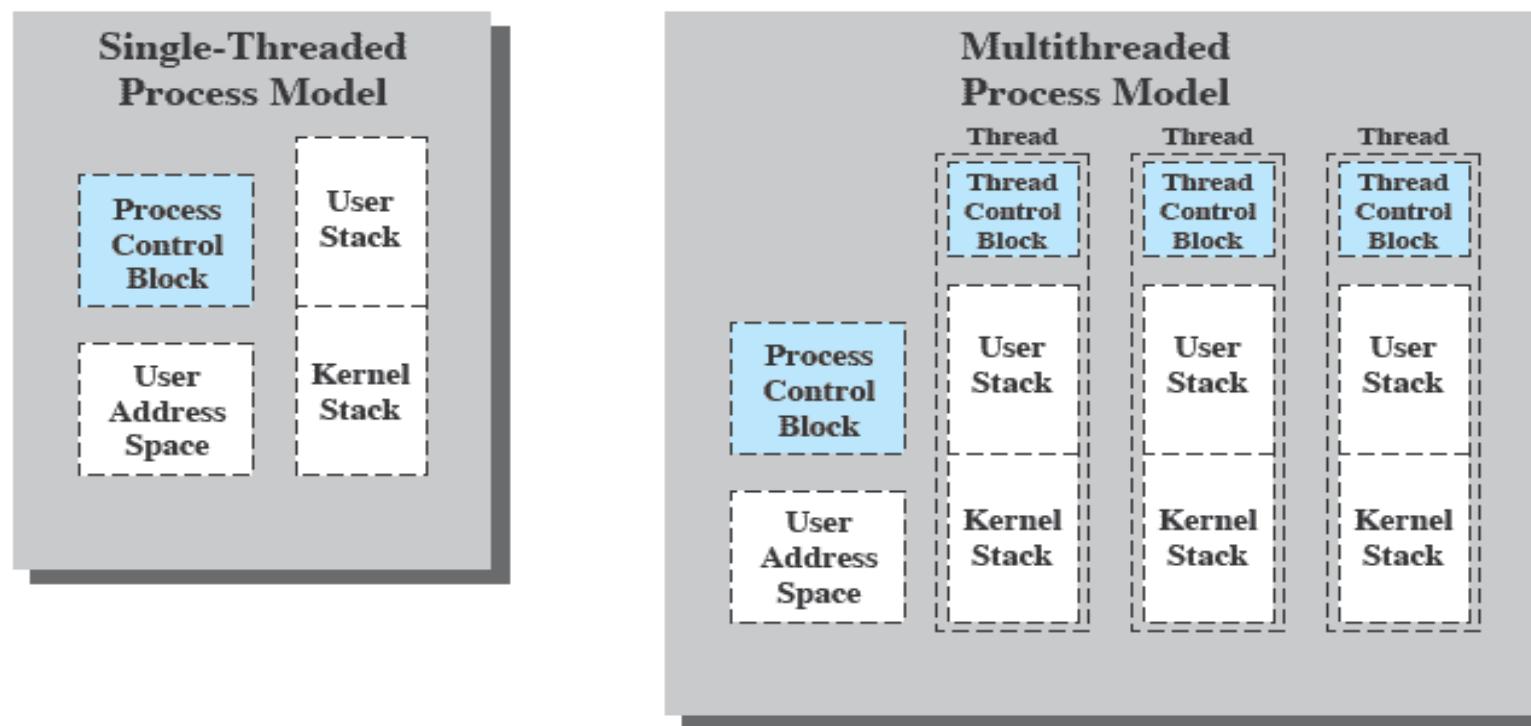
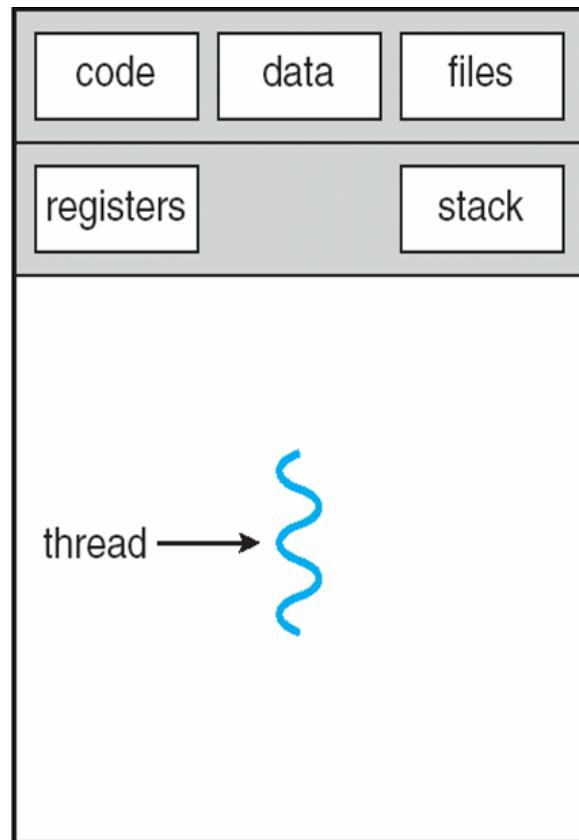
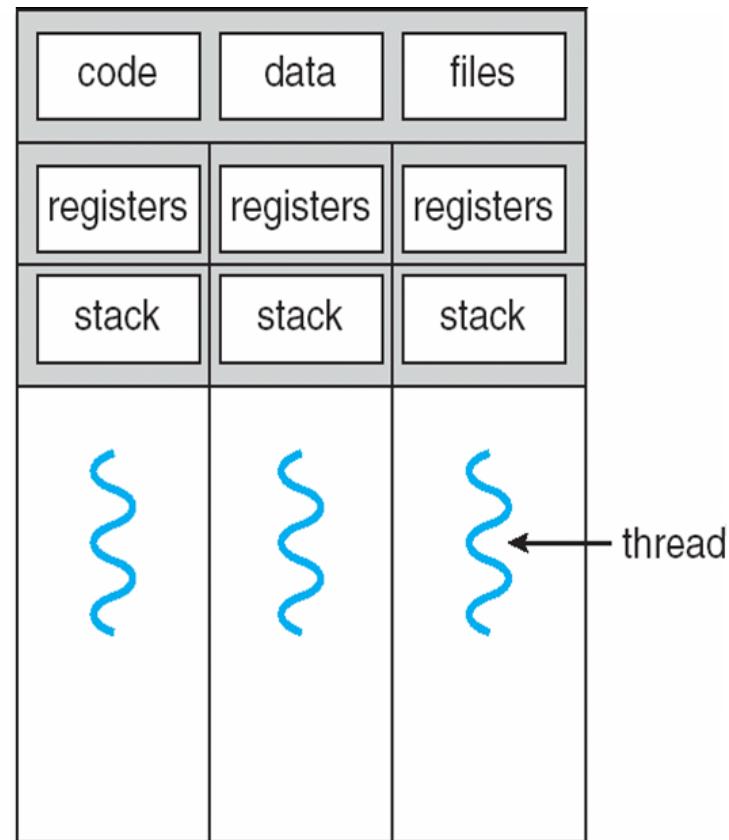


Figure 4.2 Single Threaded and Multithreaded Process Models

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)

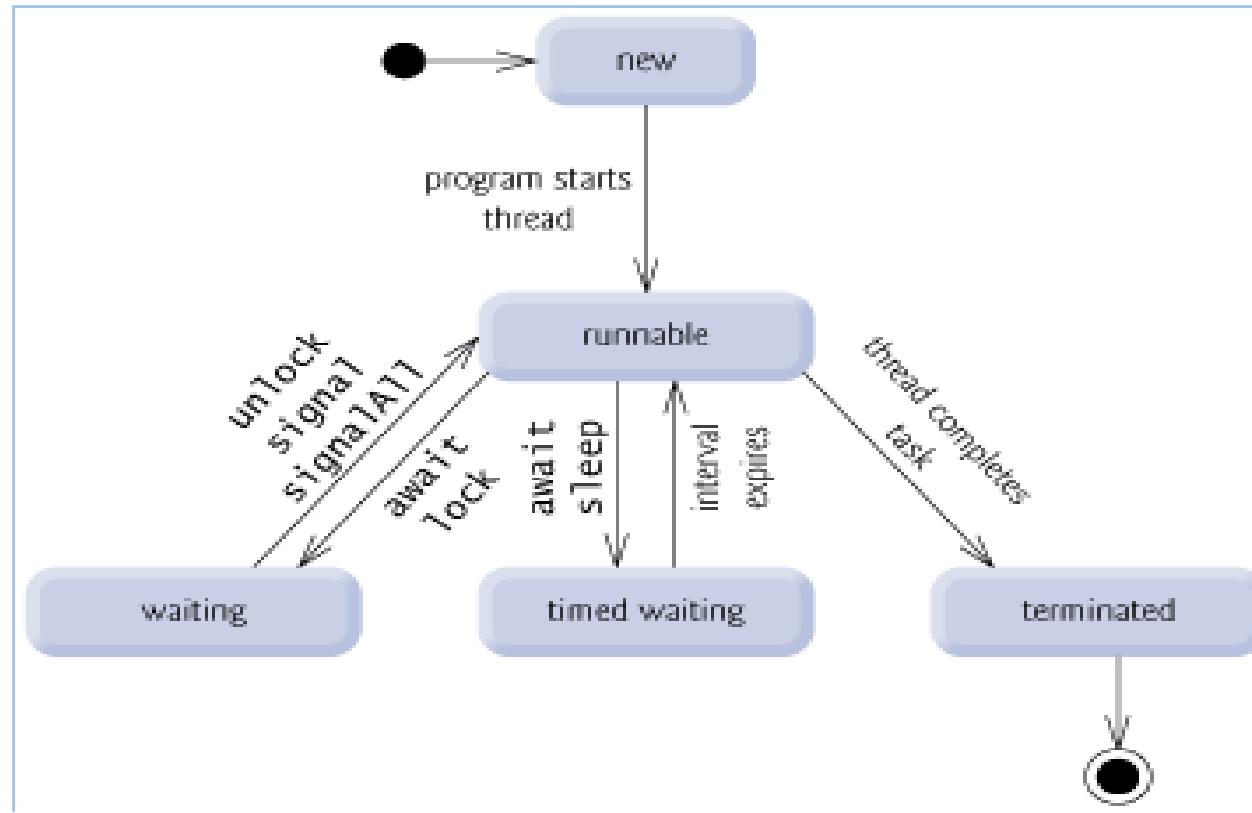
Thread Execution States

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

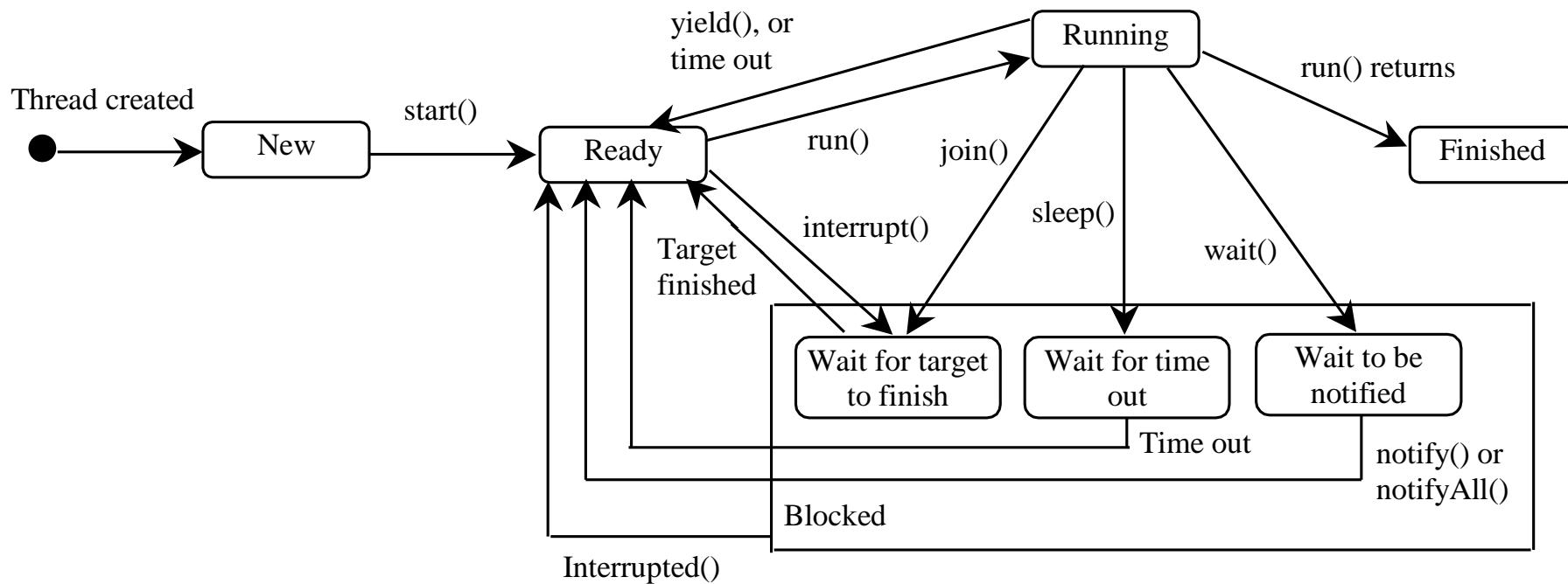
The operations needed to change state are:

- 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

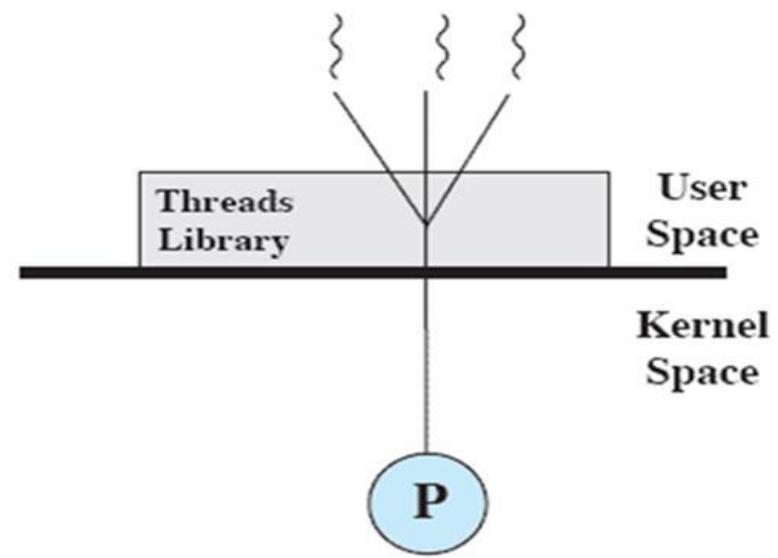
Kernel directly supports multiple threads of control in a process. Benefits are scheduling/synchronization coordination, less overhead than process, suitable for parallel application. The drawbacks are more expensive than user-level threads and more overhead.

- 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)

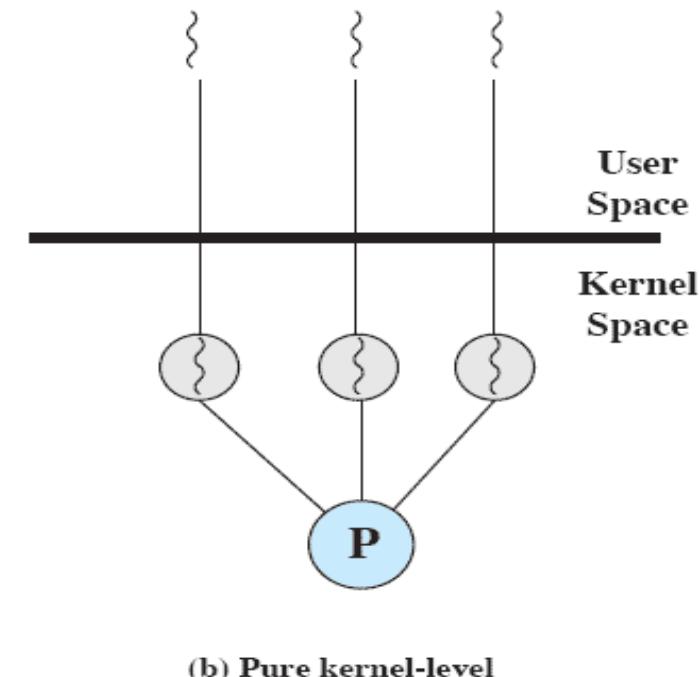
- Thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level

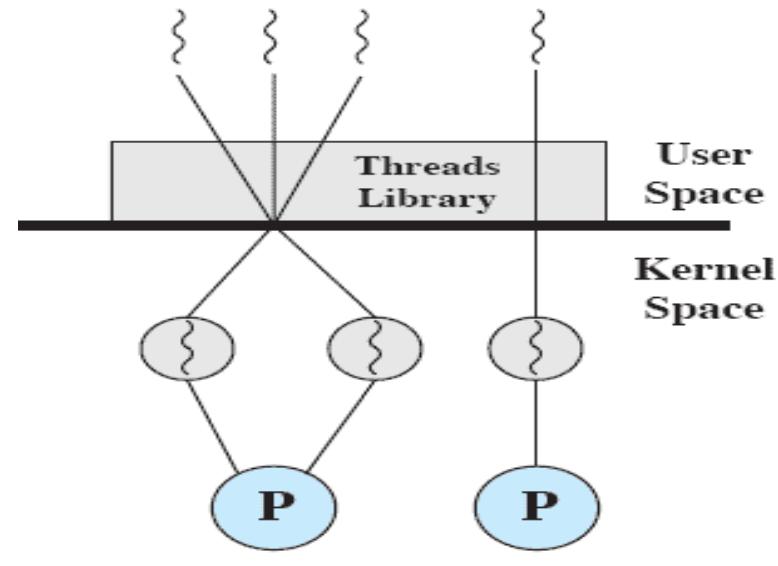
Kernel-Level Threads (KLTs)

- 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

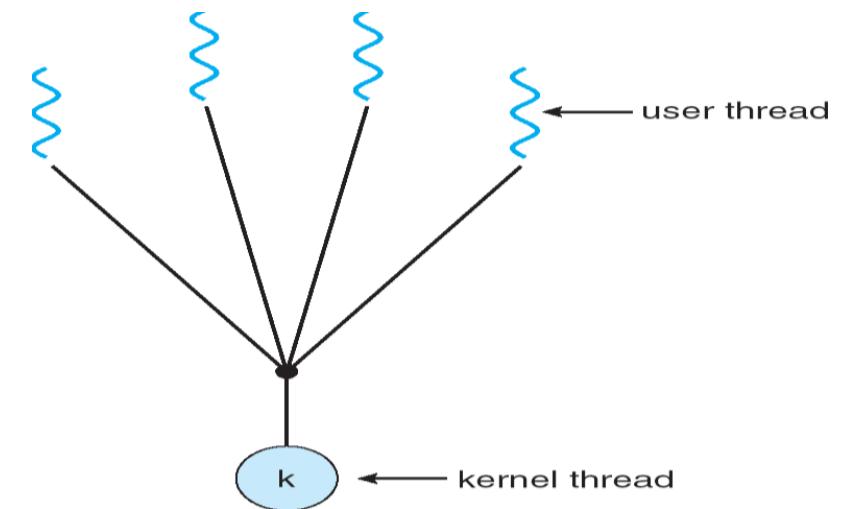
Kernels are generally multi-threaded.

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.

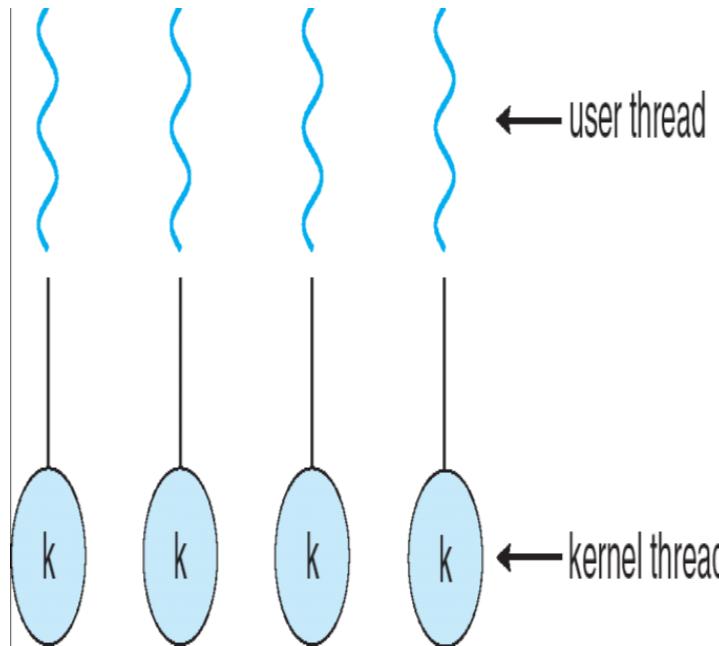
Many-to-One

- 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

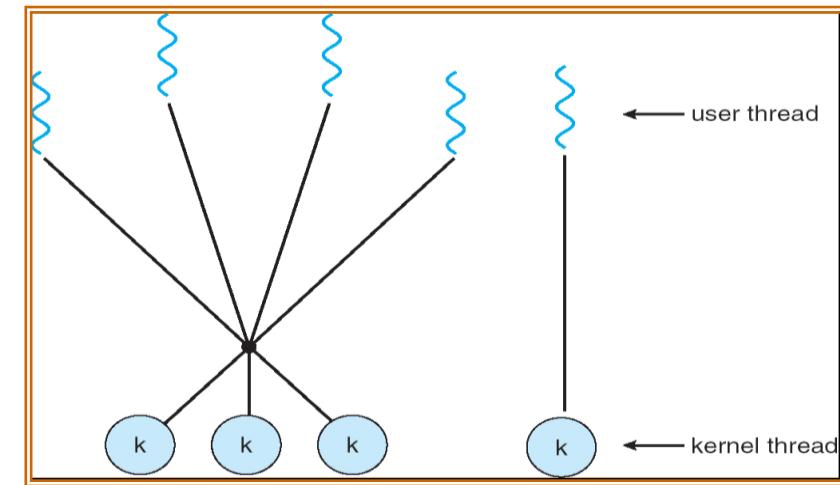
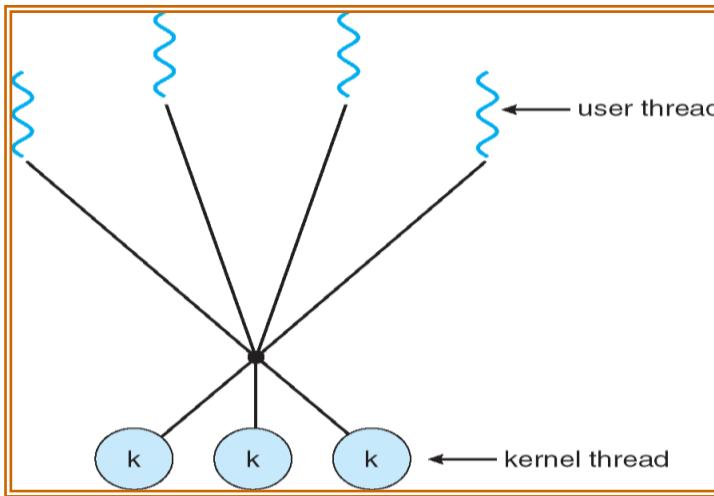


- Each **user-level thread** maps to **kernel thread**
 - Creating a **user-level thread** **creates a kernel thread**
 - *This is a drawback to this model; creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may **burden** the performance of 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
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	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
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	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

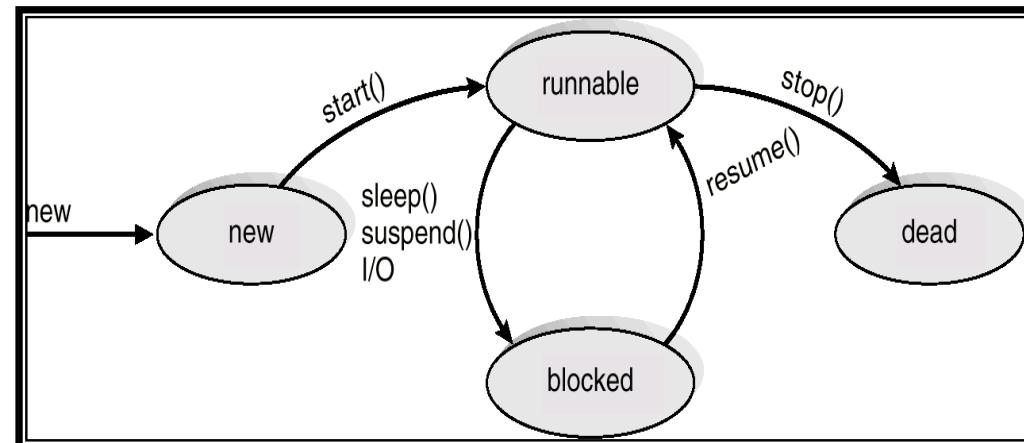
Various Implementations

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.

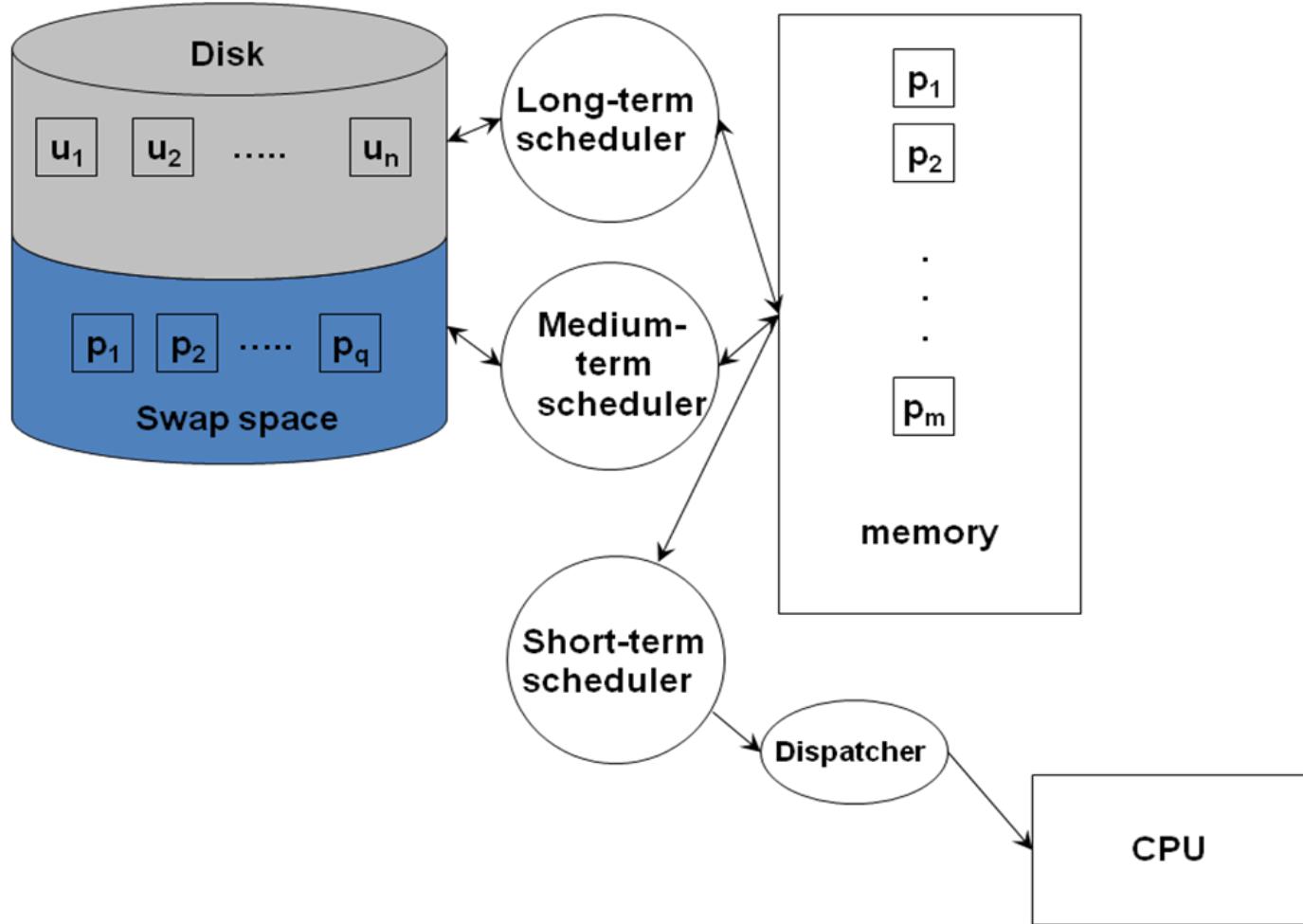
When is Scheduling needed?

- 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

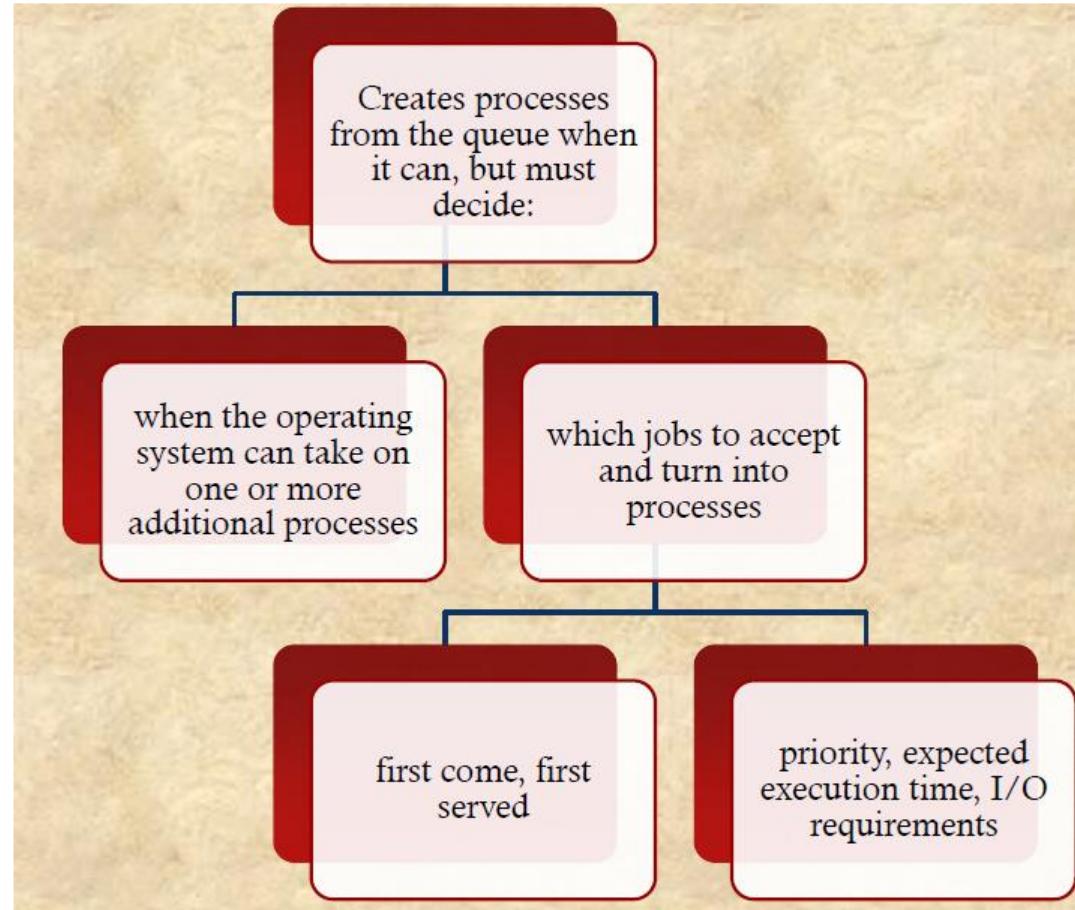
Medium-term scheduling The decision to add to the number of processes that are partially or fully in main memory

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

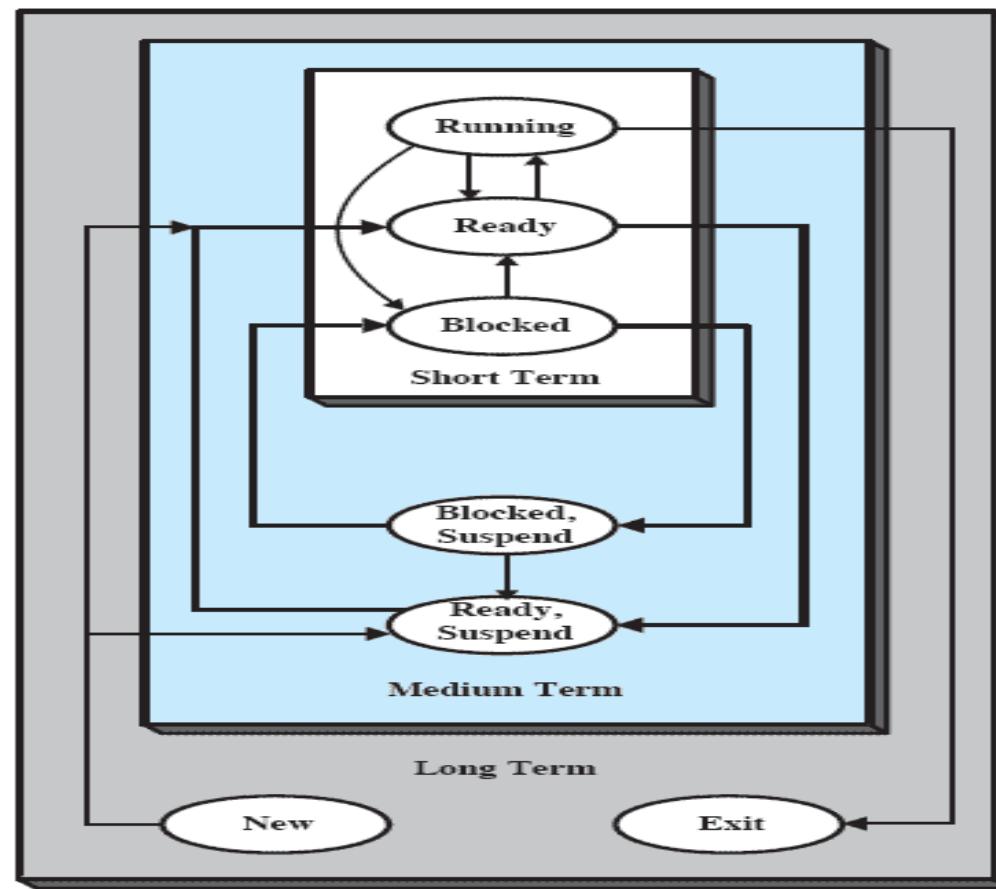
- Main objective is to allocate processor time to optimize certain aspects of system behavior
- A set of criteria is needed to evaluate the scheduling policy
- User-oriented criteria
 - relate to the behavior of the system as perceived by the individual user or process (such as response time in an interactive 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

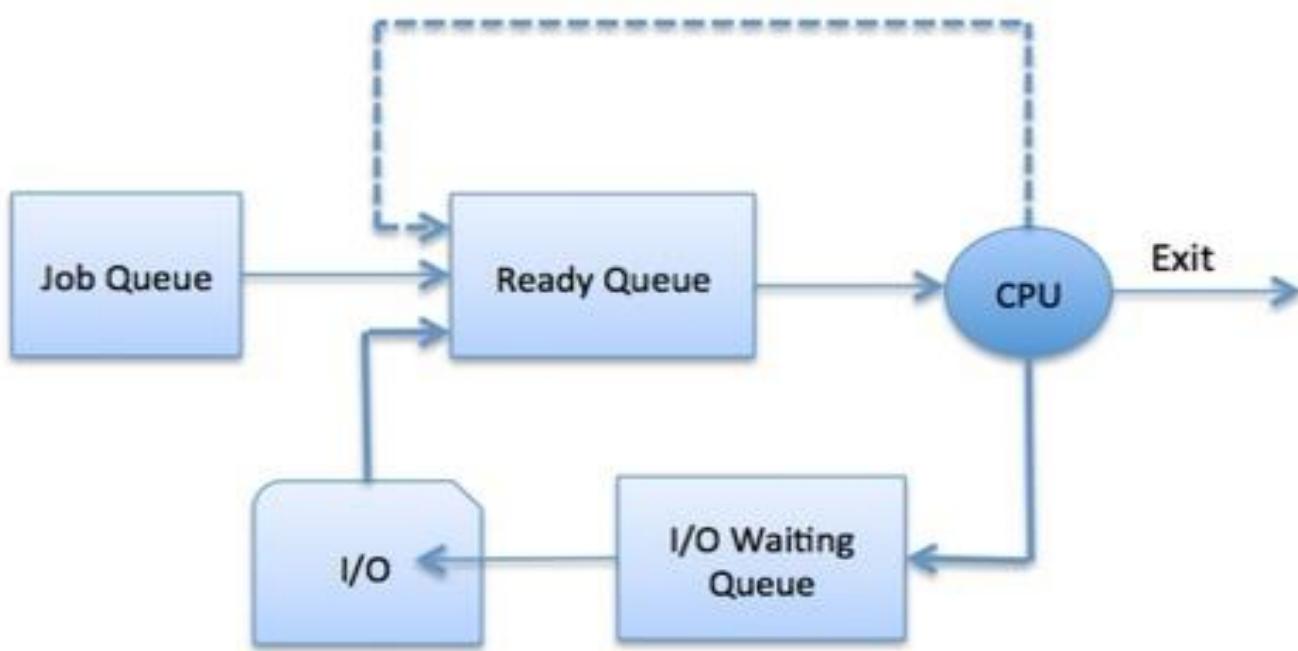


Process Scheduling Queues

- 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.



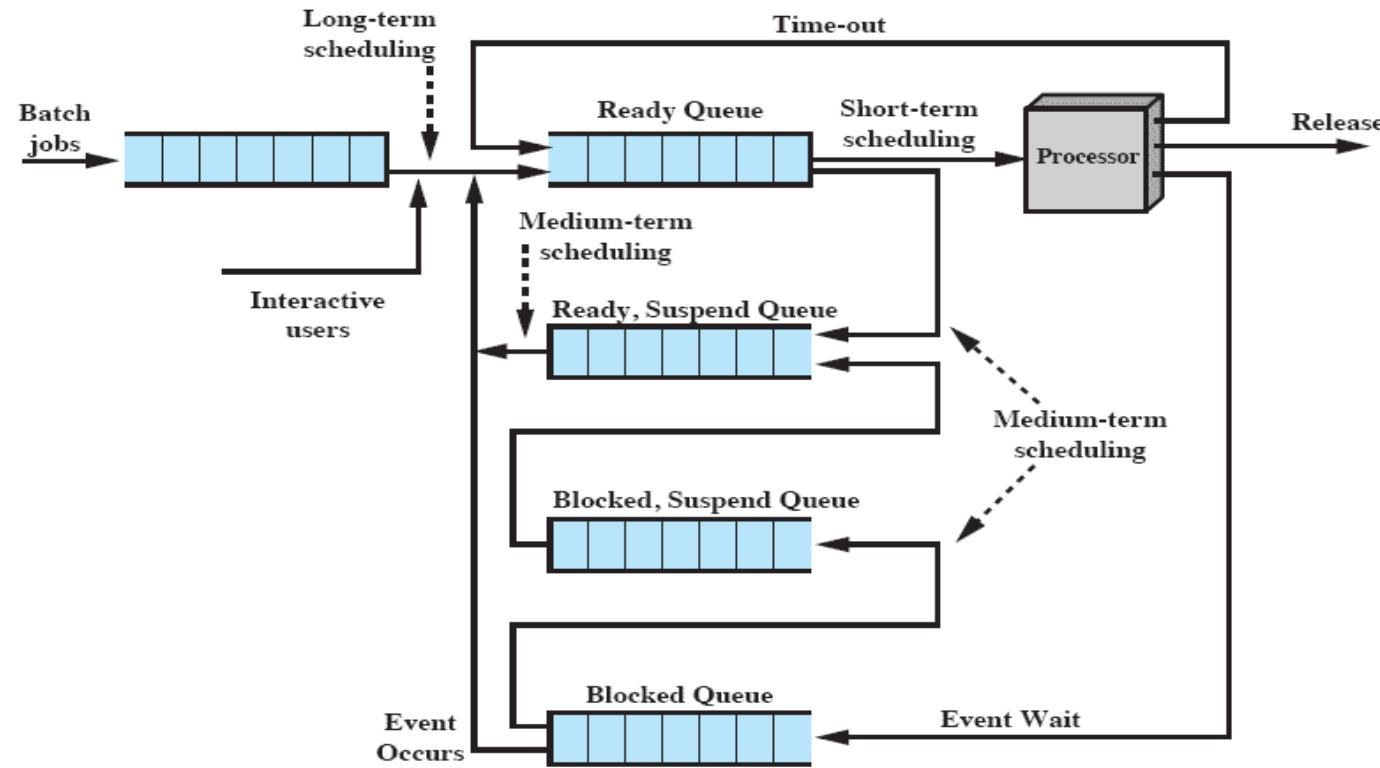


Figure 9.3 Queuing Diagram for Scheduling

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

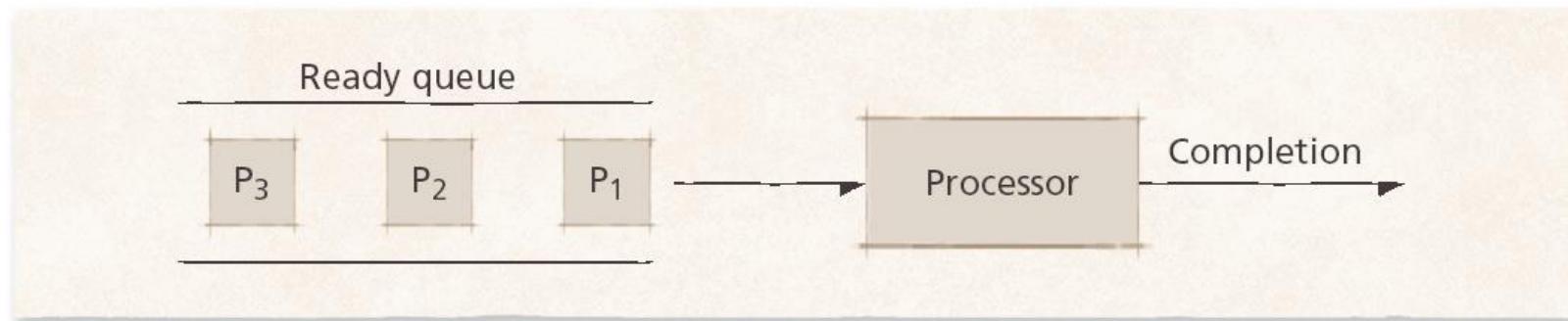
Scheduling Algorithms

- 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

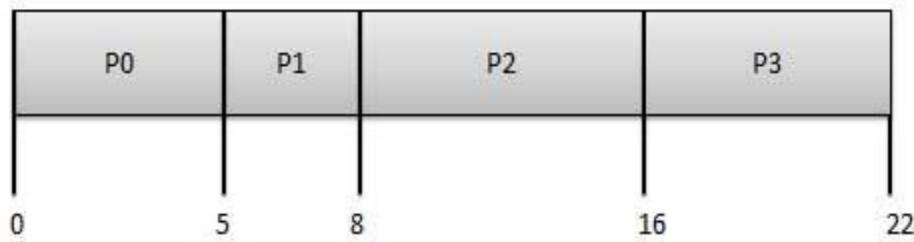
- Simplest scheme: Jobs are executed on first come, first serve basis
- Processes dispatched according to arrival time
- It is a non-preemptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue
- Poor in performance as average wait time is high
- Rarely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

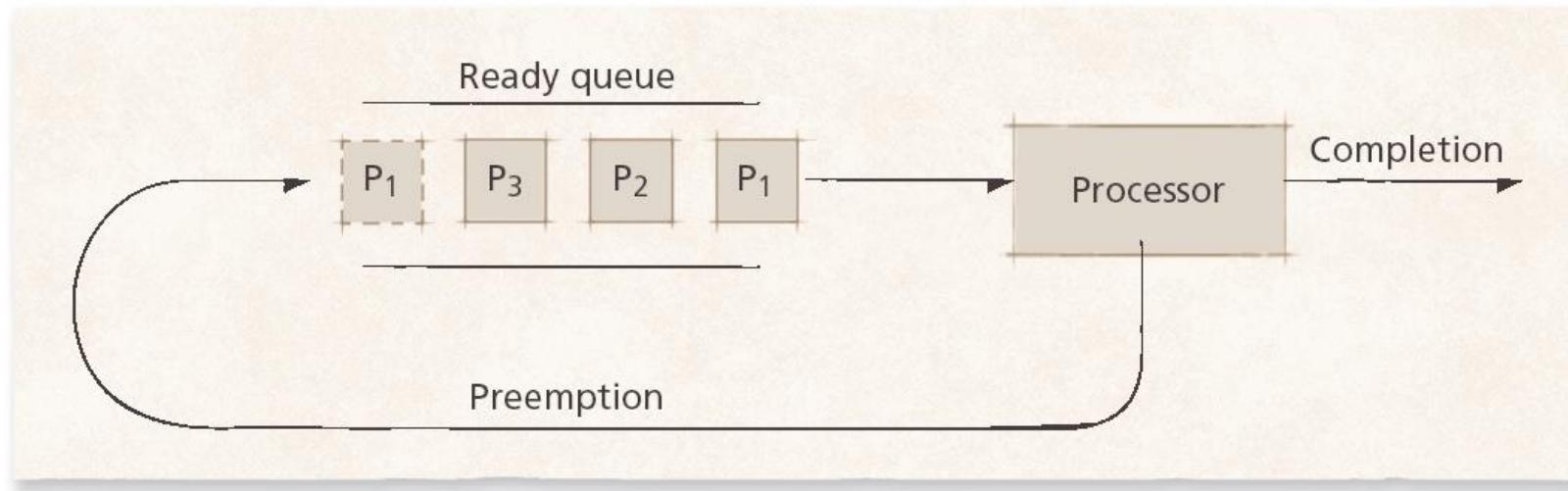
Process	Wait Time
P0	$0 - 0 = 0$
P1	$5 - 1 = 4$
P2	$8 - 2 = 6$
P3	$16 - 3 = 13$

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

Round-Robin (RR) Scheduling

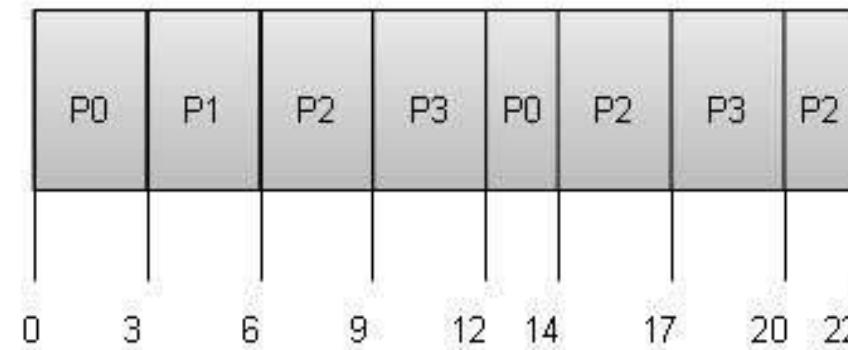
- Round-robin scheduling
 - Based on FIFO
 - Processes run only for a limited amount of time called a time slice or quantum
 - Preemptible
 - Requires the system to maintain several processes in memory to minimize overhead
 - Context switching is used to save states of preempted processes.
 - Often used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

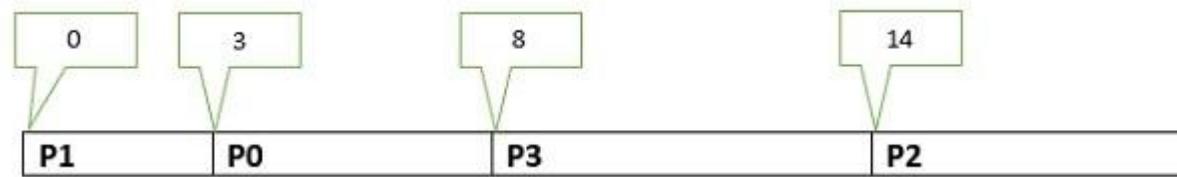
- Scheduler selects process with smallest time to finish
 - Lower average wait time than FIFO
 - Reduces the number of waiting processes
 - Potentially large variance in wait times
 - Non-preemptive
 - Results in slow response times to arriving interactive requests
 - Best approach to minimize waiting 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

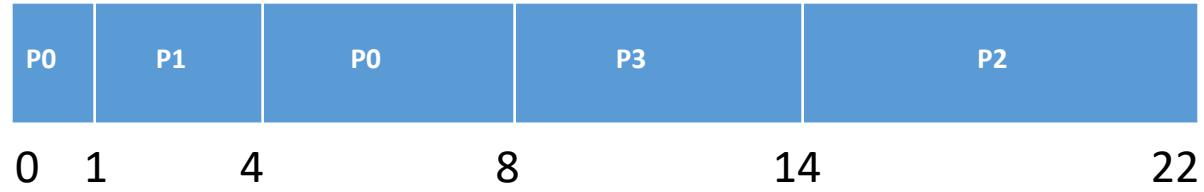
$$\text{Average Wait Time: } (3+14+8) / 4 = 6.25$$

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

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

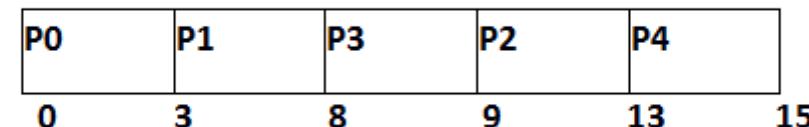
$$RR(P3) = (2+1)/1 = 3$$

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

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

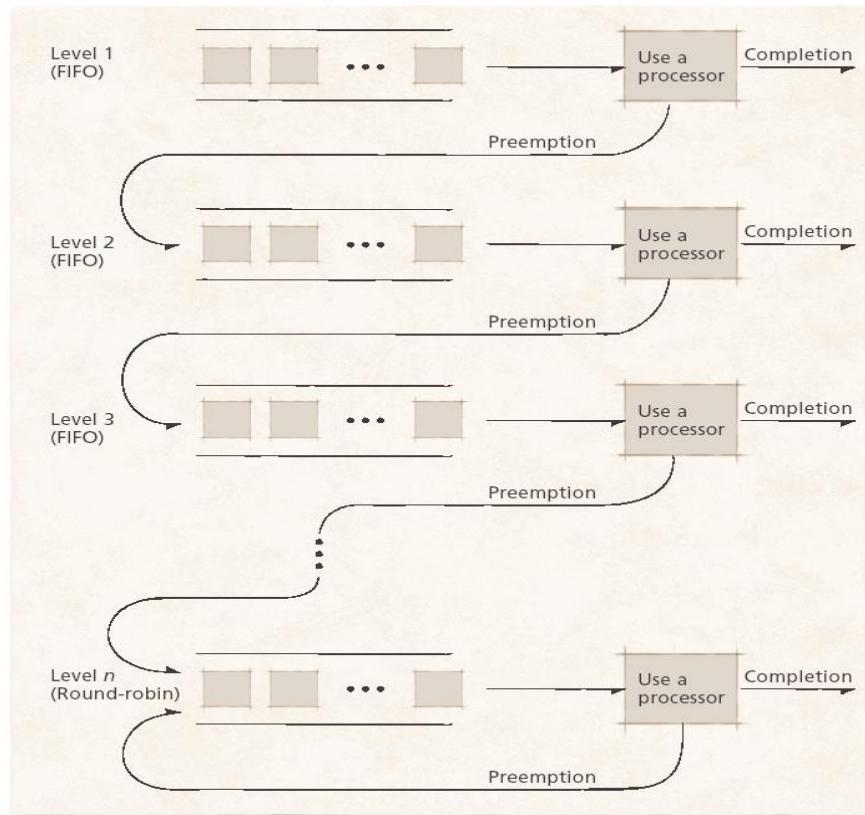


Table 9.3 Characteristics of Various Scheduling Policies

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	$\max[w]$	constant	$\min[s]$	$\min[s - e]$	$\max\left(\frac{w + s}{s}\right)$	(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Throughput	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	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
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	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.

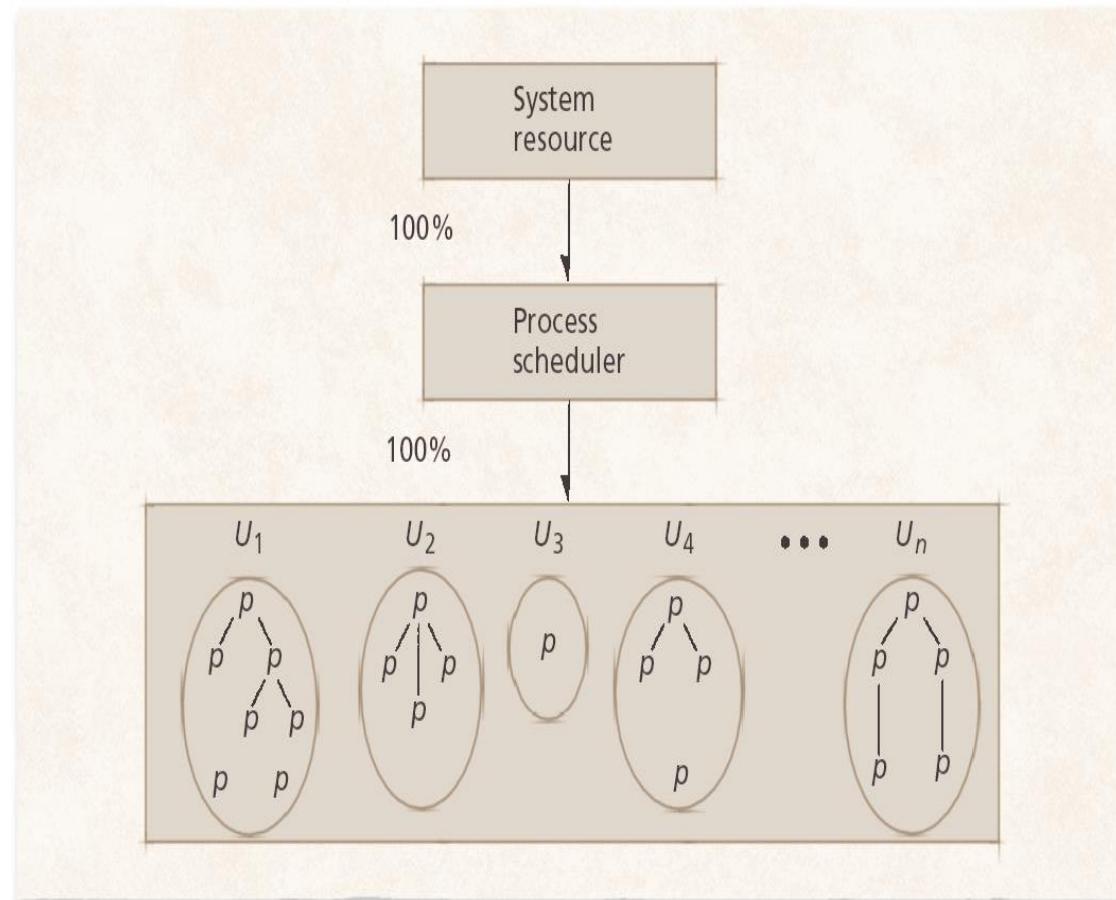
Fair Share Scheduling

It is a scheduling algorithm for computer operating systems that dynamically distributes the time quanta “equally” to its users.

- 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

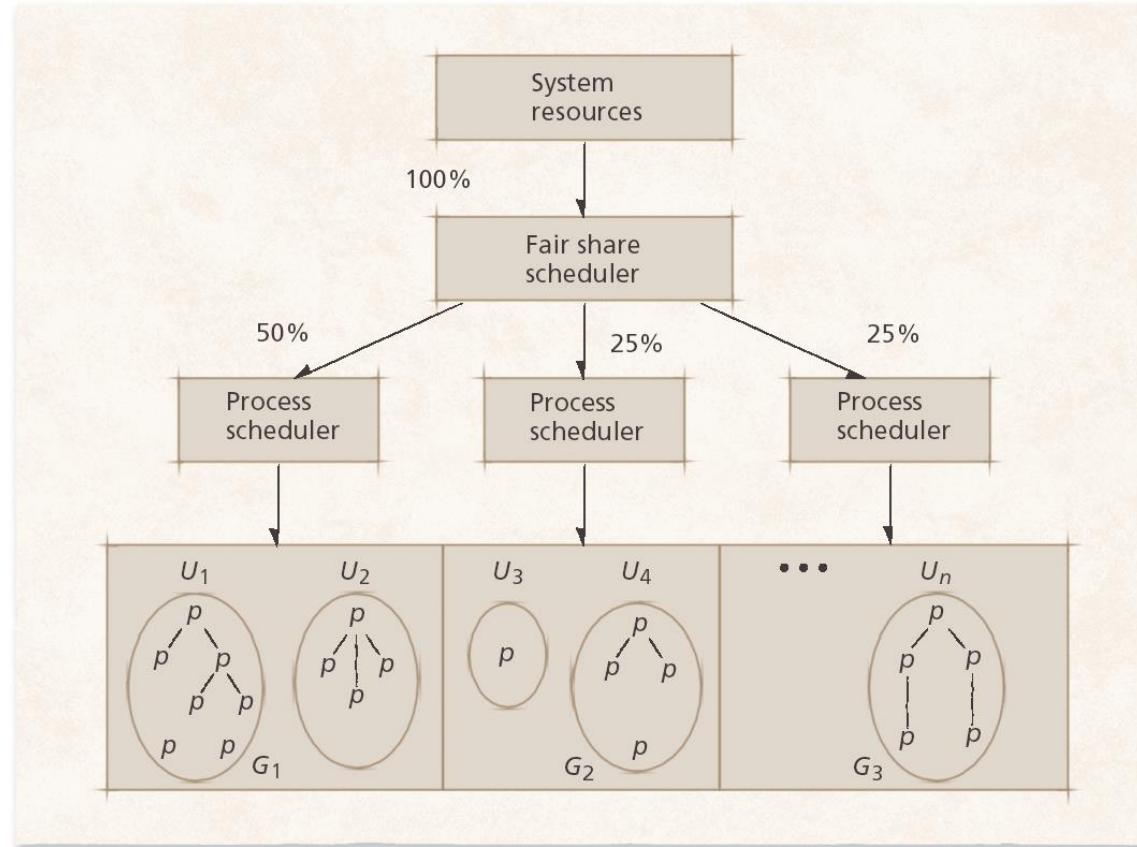
Fair Share Scheduling

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

<https://www.geeksforgeeks.org/deadline-scheduler-in-operating-system/>

- 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.

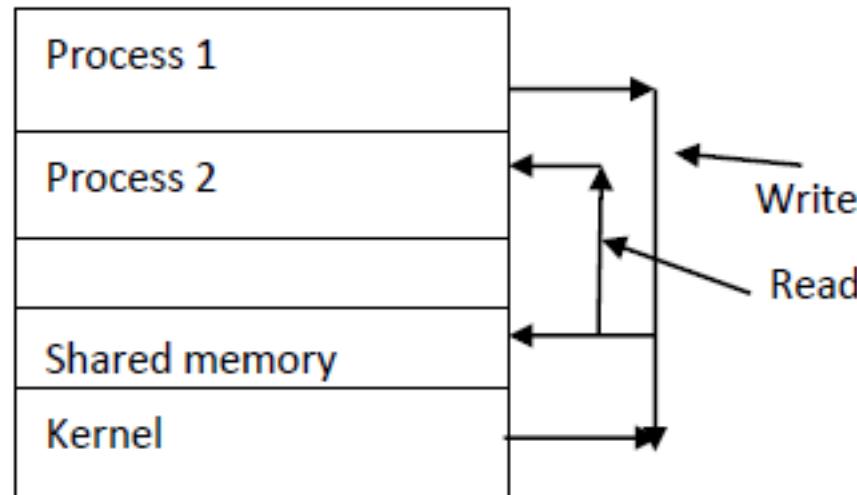
Concurrent processes generally interact through either of the following models:

- Shared memory
- Message Passing

Shared Memory model

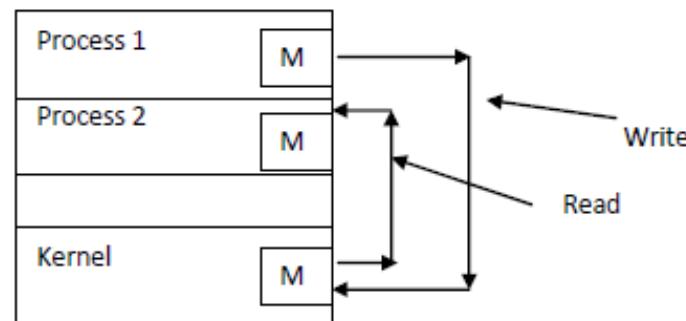
In the shared memory model, a region of memory that is shared by co-operating processes is established.

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.