

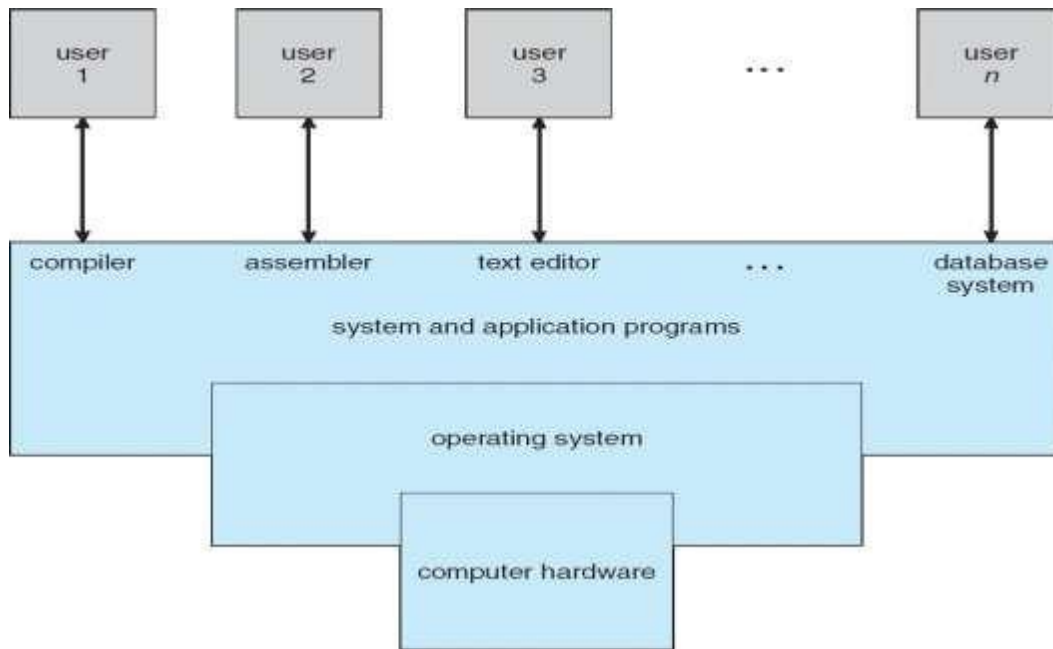
# MODULE-I

## Introduction to OS

A program that acts as an intermediary between a user of a computer and the computer hardware

Operating system goals:

- Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner
- Computer System Structure
- Computer system can be divided into four components
  - Hardware - provides basic computing resources
    - CPU, memory, I/O devices
  - Operating system
    - Controls and coordinates use of hardware among various applications and users
  - Application programs - define the ways in which the system resources are used to solve the computing problems of the users
    - Word processors, compilers, web browsers, database systems, video games
  - Users
    - People, machines, other computers



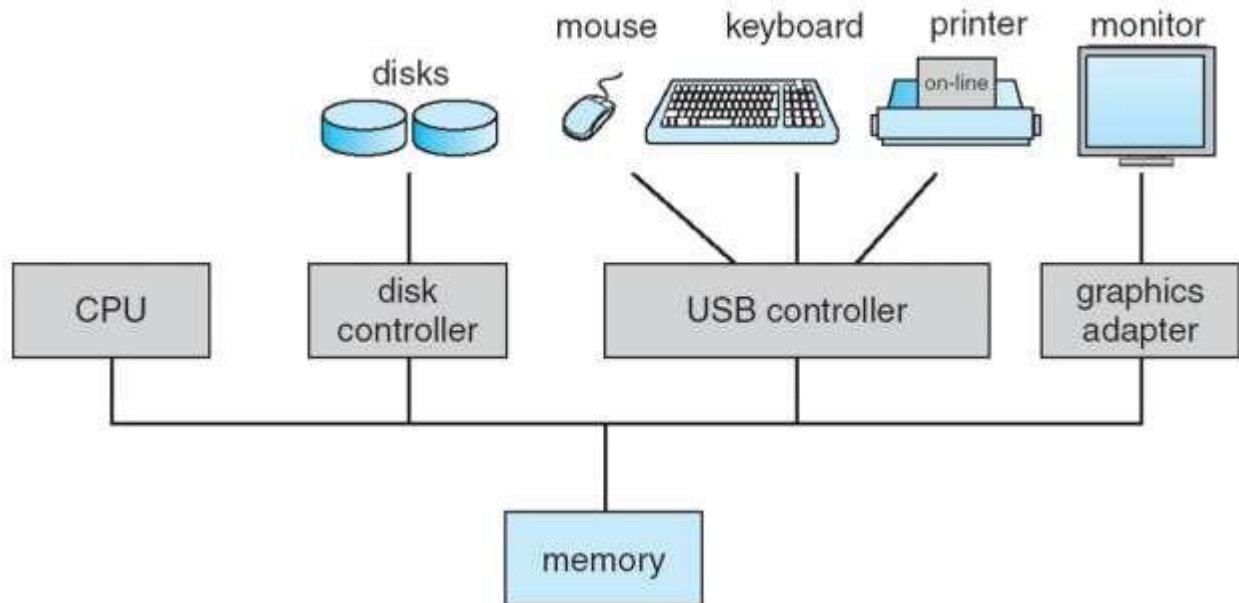
## OS Definition

- OS is a resource allocator
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a control program
  - Controls execution of programs to prevent errors and improper use of the computer

## 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
  - Loads operating system kernel and starts execution

## Computer System Organisation



- 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*
- 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 or a user request
- An operating system is **interrupt driven**
- The operating system preserves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
- **polling**
- **vectored** interrupt system
- 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 only upon I/O completion
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop (contention for memory access)
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing
- 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

## Storage Structure

- Main memory - only large storage media that the CPU can access directly
- Secondary storage - extension of main memory that provides large nonvolatile storage capacity

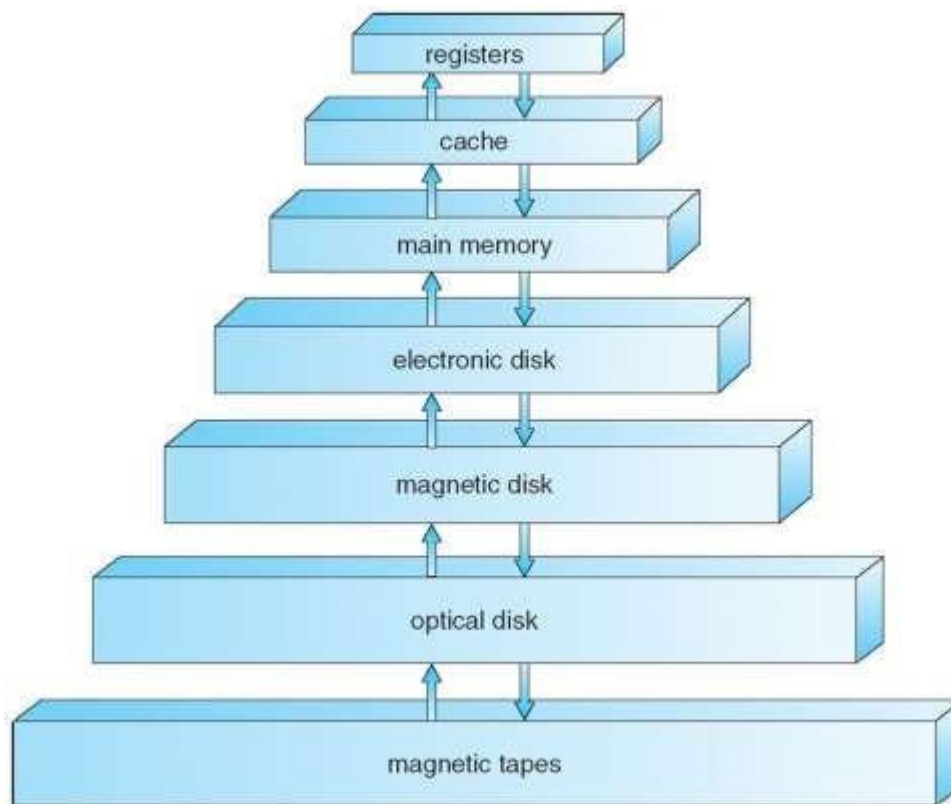
- Magnetic disks - rigid metal or glass platters covered with magnetic recording material

### 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 Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility



## 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
  - 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

## Computer System Architecture

- Most systems use a single general-purpose processor (PDAs through mainframes)
  - Most systems have special-purpose processors as well
- Multiprocessors systems growing in use and importance
  - Also known as parallel systems, tightly-coupled systems
  - Advantages include
    - Increased throughput
    - Economy of scale
    - Increased reliability - graceful degradation or fault tolerance
  - Two types
    - Asymmetric Multiprocessing

- Symmetric Multiprocessing

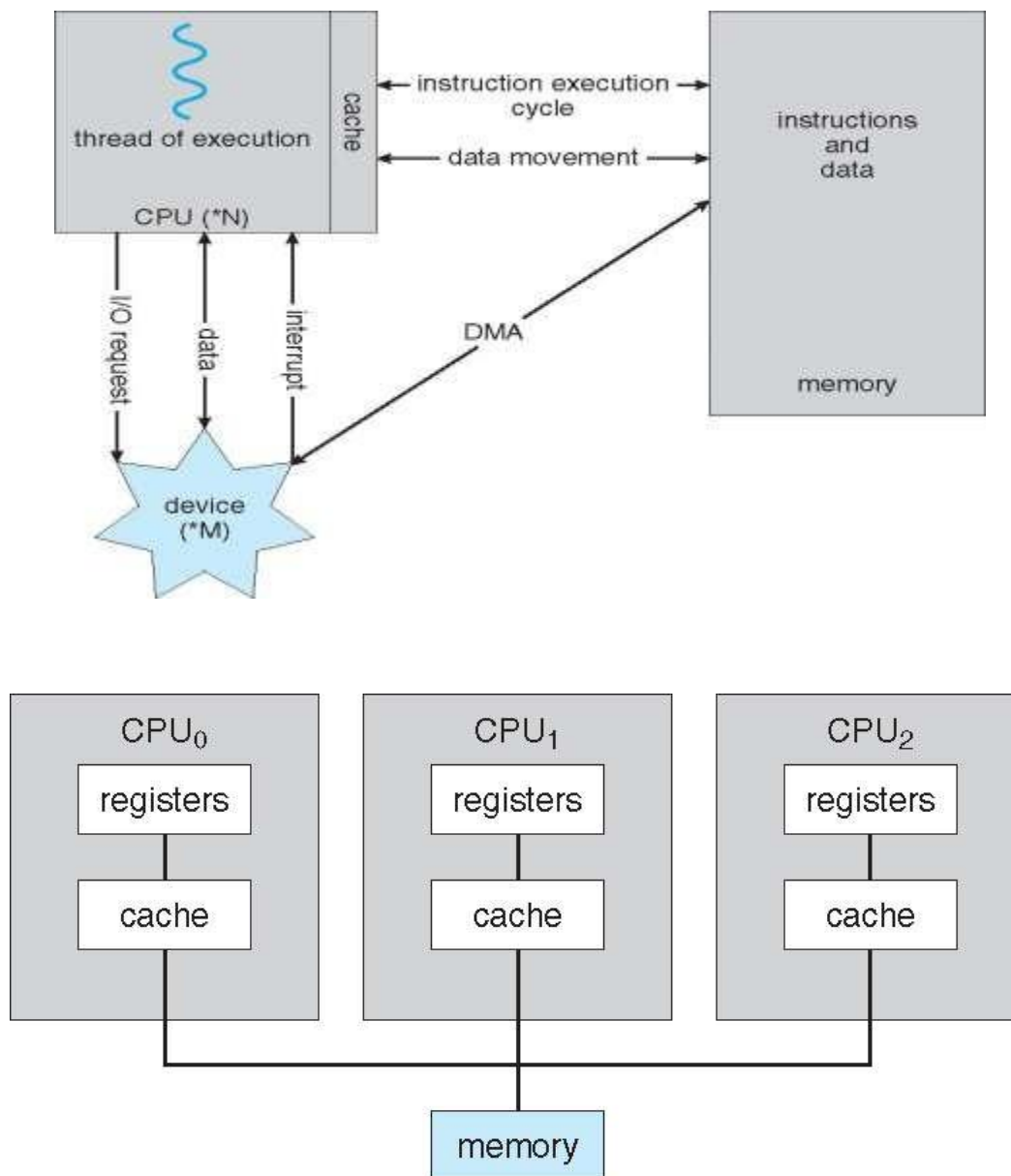


Fig: Symmetric multiprocessing architecture

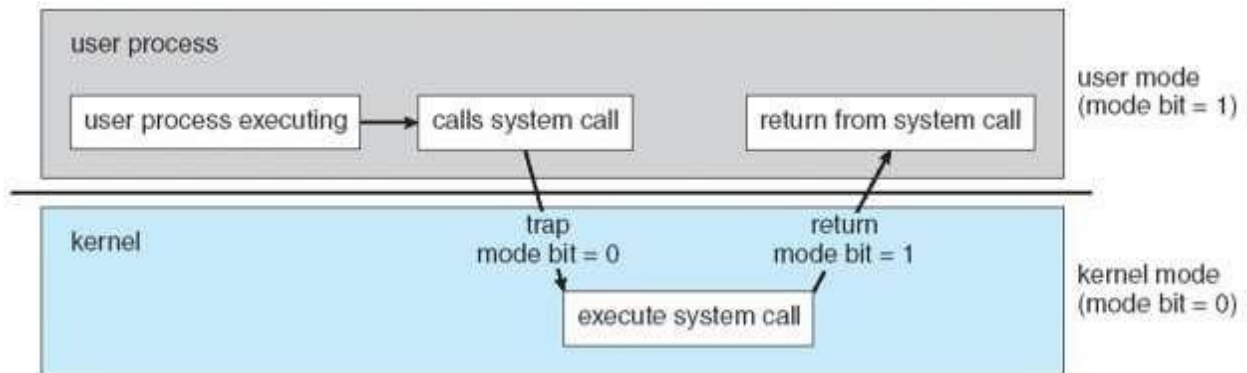
### Operating System Structure

- **Multiprogramming** needed for efficiency
  - 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
- One job selected and run via **job scheduling**
- When it has to wait (for I/O for example), OS switches to another job
- **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
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory → **process**
  - If several jobs ready to run at the same time → **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
- **Operating System Operation**
- 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
  - **User mode** and **kernel mode**
  - **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
- Timer to prevent infinite loop / process hogging resources



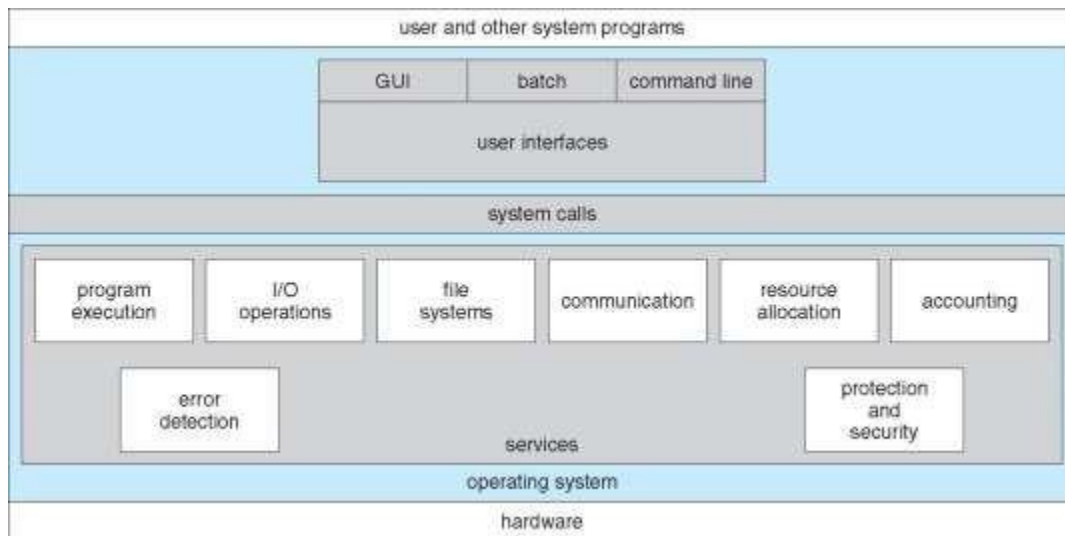
- Set interrupt after specific period
- Operating system decrements counter
- When counter zero generate an interrupt
- Set up before scheduling process to regain control or terminate program that exceeds allotted time



## OS Services

- One set of operating-system services provides functions that are helpful to the user:
  - User interface - Almost all operating systems have a user interface (UI)
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which may involve a file or an I/O device
  - File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file information, permission management.
- One set of operating-system services provides functions that are helpful to the user (Cont):
  - Communications - Processes may exchange information, on the same computer or between computers over a network

- Communications may be via shared memory or through message passing (packets moved by the OS)
- Error detection - OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system
- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
  - **Accounting** - To keep track of which users use how much and what kinds of computer resources
  - **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

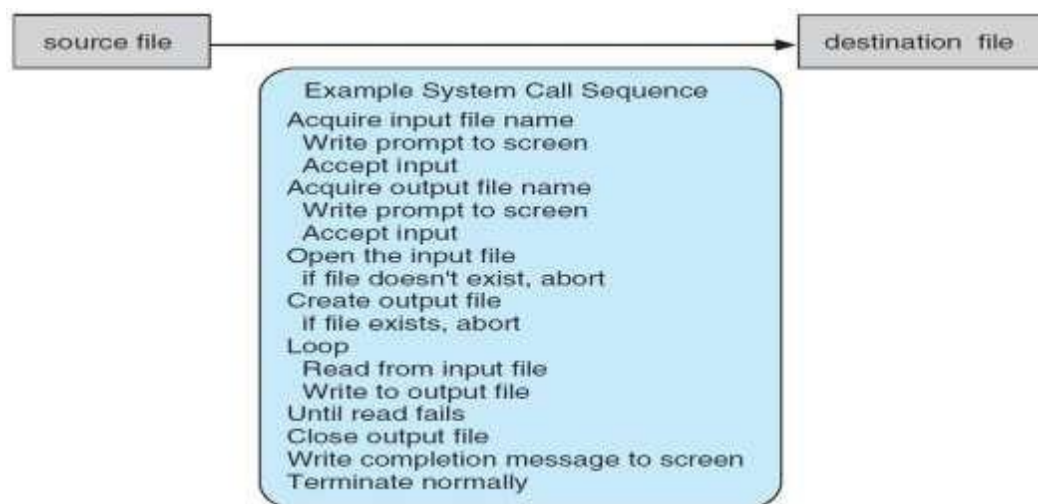


## System Call

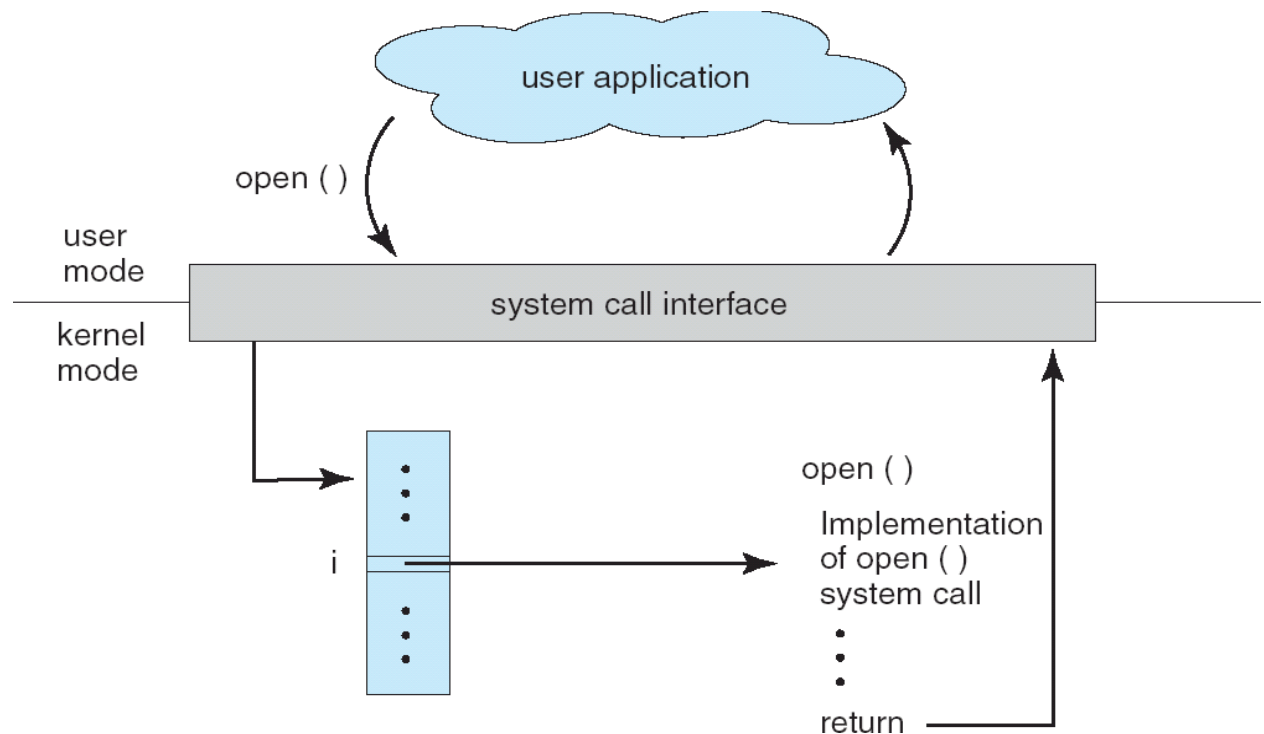
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

### Example

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



- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller 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)



#### Types of system call

- Process control
- File management
- Device management
- Information maintenance

- Communications
- Protection

	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()

## OS Structure

- n MS-DOS - written to provide the most functionality in the least space
  - 1 Not divided into modules
  - 1 Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

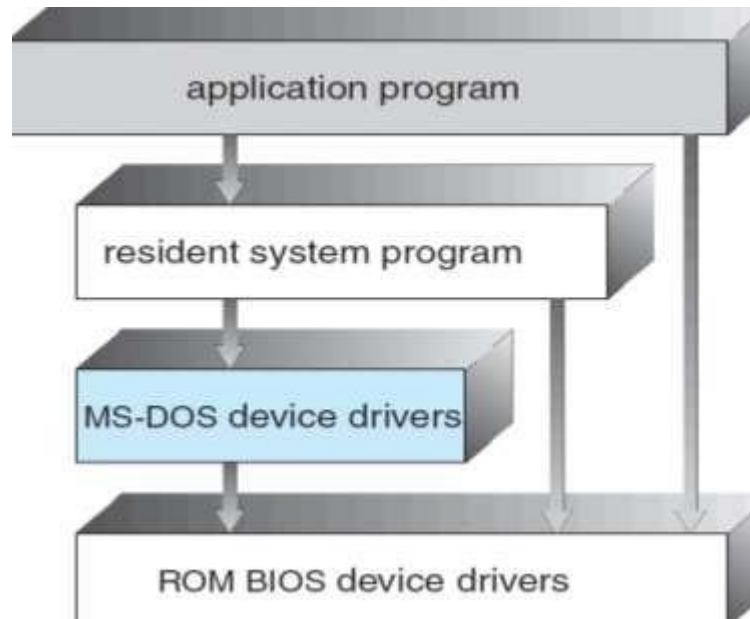


Fig: MS Dos structure

### Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

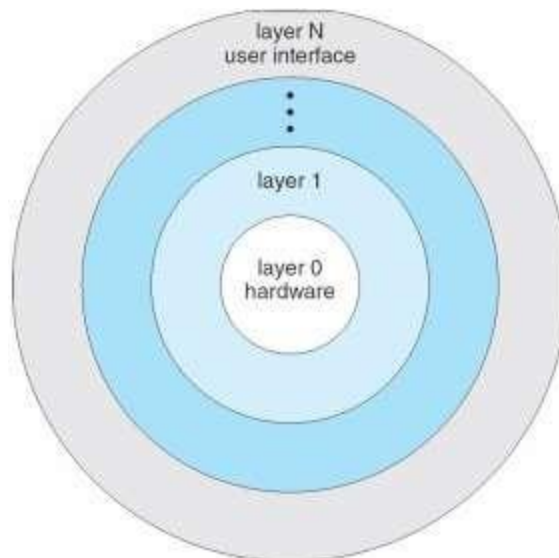


Fig: Layered System

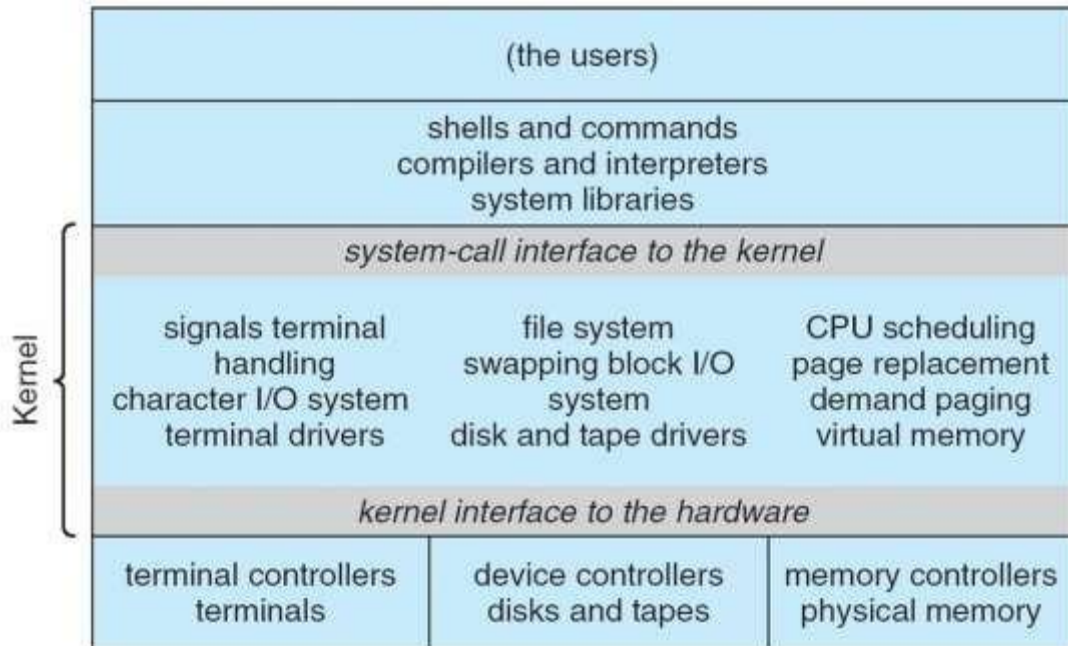


Fig: UNIX system structure

### Micro Kernel Structure

- Moves as much from the kernel into “user” space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

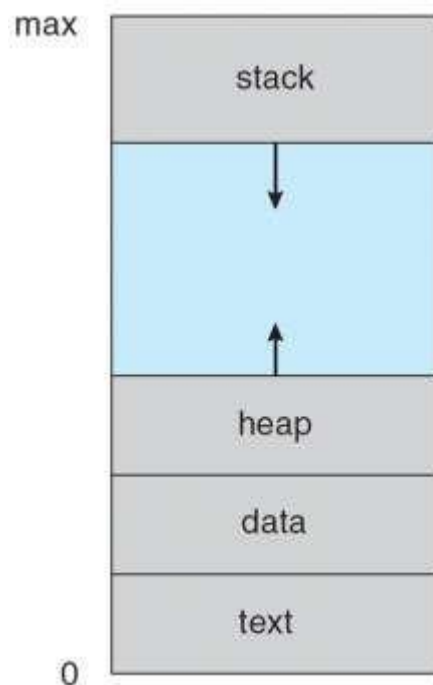
### Virtual Machine

- 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 host creates the illusion that a process has its own processor and (virtual memory)
- Each guest provided with a (virtual) copy of underlying computer

## Process Management

- An operating system executes a variety of programs:
  - Batch system - jobs
  - Time-shared systems - user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process - a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section





## Process State

As a process executes, it changes *state*

- new: The process is being created
- running: Instructions are being executed
- waiting: The process is waiting for some event to occur
- ready: The process is waiting to be assigned to a processor
- terminated: The process has finished execution

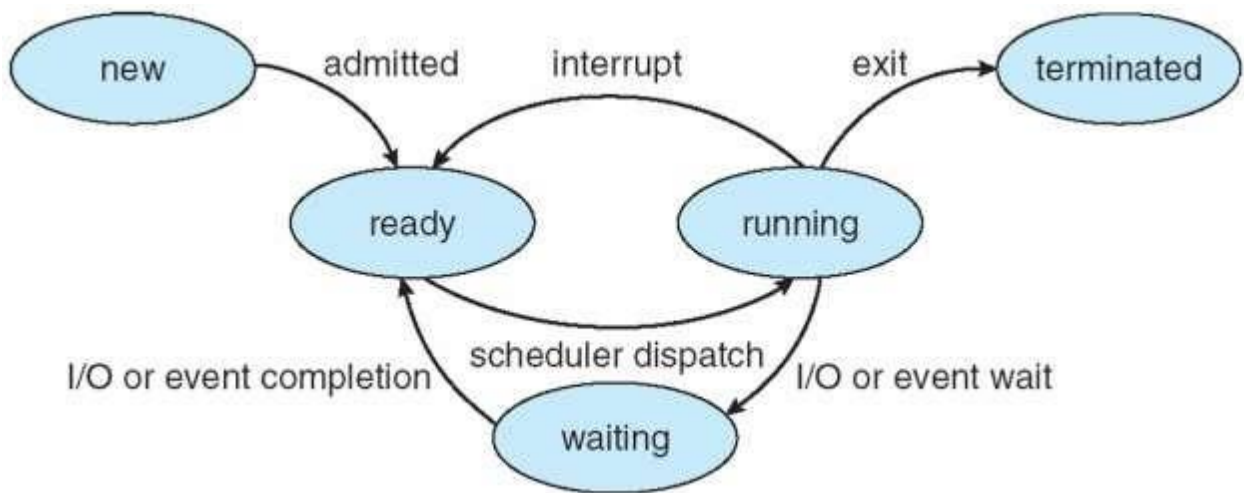


Fig: Process Transition Diagram

## PCB: Process Control Block

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



Fig: PCB

### Context Switching

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

### Process Scheduling Queues

- Job queue - set of all processes in the system
- Ready queue - set of all processes residing in main memory, ready and waiting to execute
- Device queues - set of processes waiting for an I/O device
- Processes migrate among the various queues

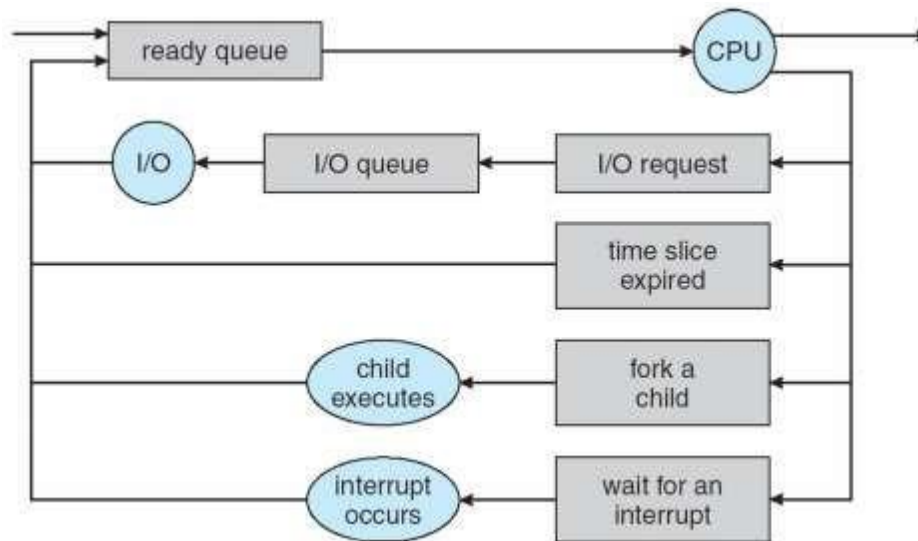
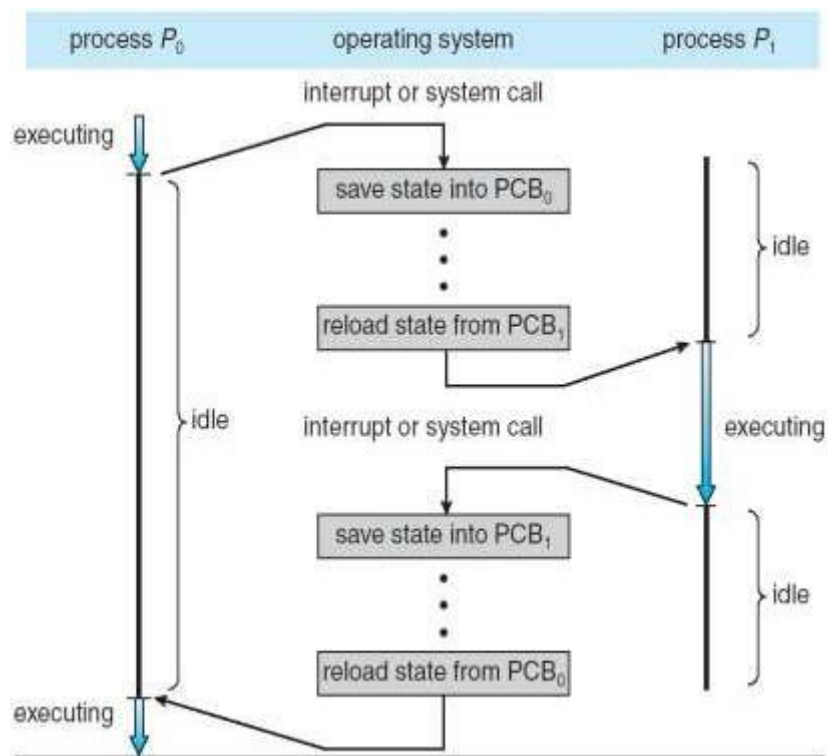


Fig: Process Scheduling



## Schedulers

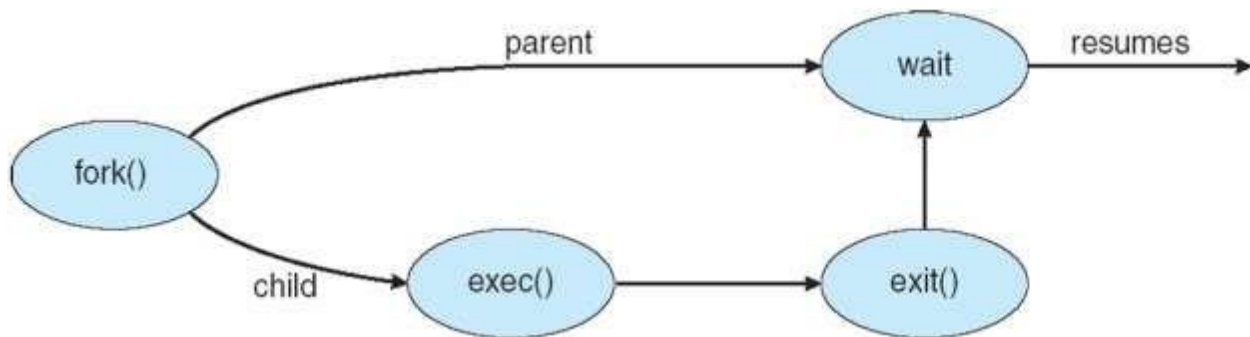
- Long-term scheduler (or job scheduler) - selects which processes should be brought into the ready queue

- Short-term scheduler (or CPU scheduler) - selects which process should be executed next and allocates CPU
- Short-term scheduler is invoked very frequently (milliseconds)  $\Rightarrow$  (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes)  $\Rightarrow$  (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - I/O-bound process - spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process - spends more time doing computations; few very long CPU bursts

## Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier (pid)**
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process

- **exec** system call used after a **fork** to replace the process' memory space with a new program



## Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - cascading termination

## Inter Process Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity

- Convenience
- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

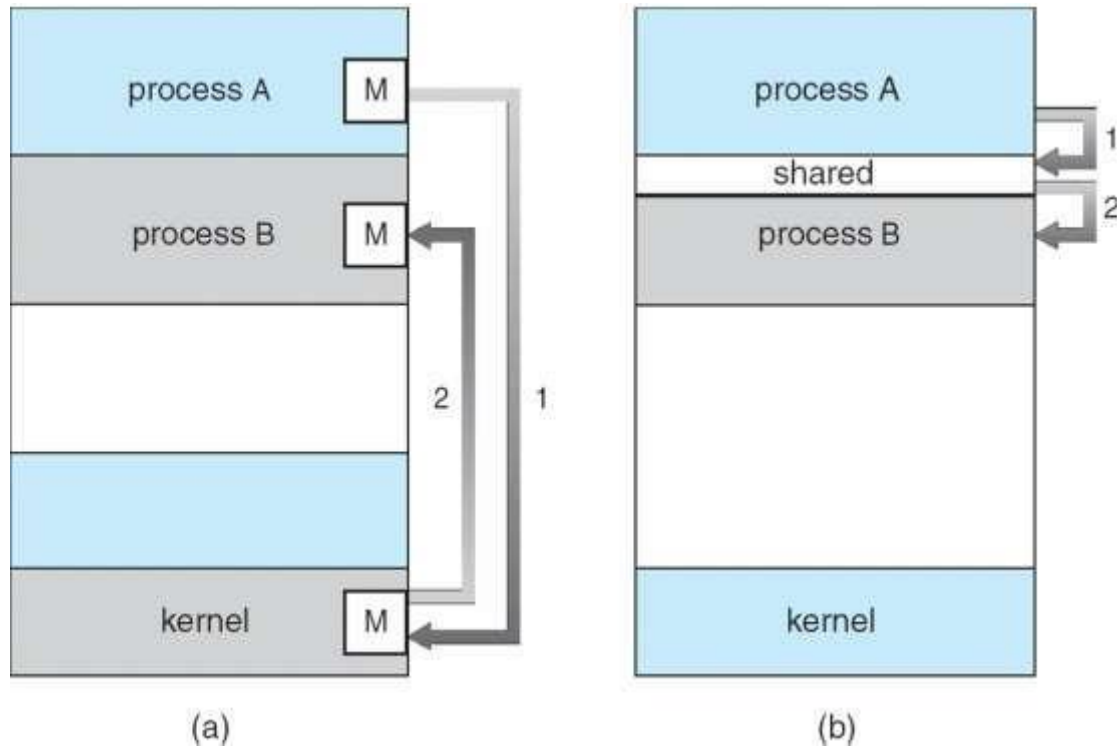


Fig:a- Message Passing, b- Shared Memory

### Cooperating Process

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity

- Convenience

## Producer Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

```
while (true) {  
    /* Produce an item */  
  
    while (((in = (in + 1) % BUFFER SIZE count) == out)  
        ; /* do nothing -- no free buffers */  
  
        buffer[in] = item;  
  
        in = (in + 1) % BUFFER SIZE;  
  
}
```

Fig: Producer Process

```
while (true) {  
  
    while (in == out)  
        ; // do nothing -- nothing to consume  
  
        // remove an item from the buffer  
  
        item = buffer[out];  
  
        out = (out + 1) % BUFFER SIZE;
```

## IPC-Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system - processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:

- `send(message)` - message size fixed or variable
  - `receive(message)`
- If  $P$  and  $Q$  wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

### Direct Communication

- Processes must name each other explicitly:
  - **send** ( $P$ , *message*) - send a message to process  $P$
  - **receive**( $Q$ , *message*) - receive a message from process  $Q$
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
- The link may be unidirectional, but is usually bi-directional

### Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links



- Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - **send**(*A, message*) - send a message to mailbox A
  - **receive**(*A, message*) - receive a message from mailbox A
- Allow a link to be associated with at most two processes
- Allow only one process at a time to execute a receive operation
- Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

### Synchronisation

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or null

### Buffering

Queue of messages attached to the link; implemented in one of three ways

1. Zero capacity - 0 messages  
Sender must wait for receiver (rendezvous)
2. Bounded capacity - finite length of  $n$  messages  
Sender must wait if link full

3. Unbounded capacity - infinite length  
Sender never waits

## Thread

- A thread is a flow of execution through the process code, with its own program counter, system registers and stack.
- A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism.
- Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

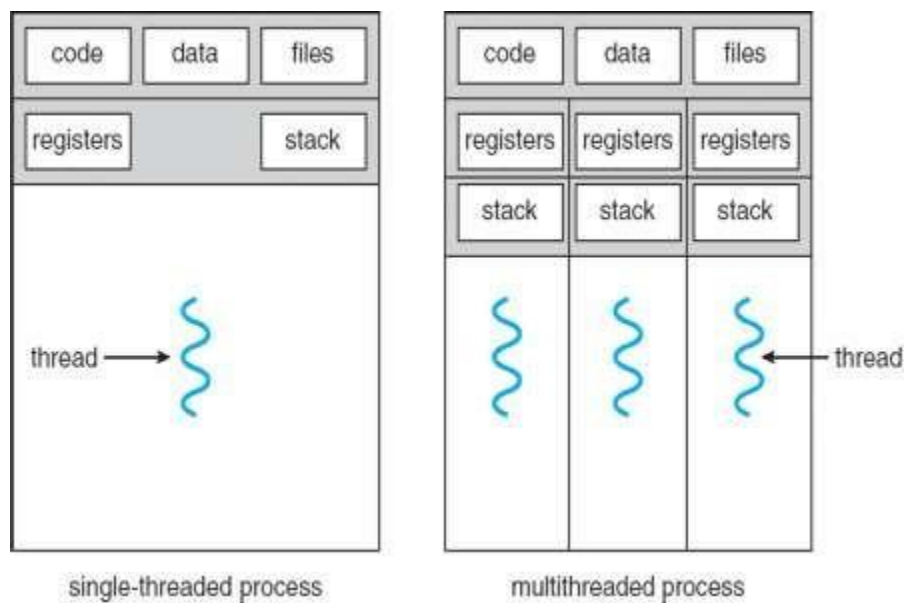


Fig: Single threaded vs multithreaded process

## Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability

## **User Threads**

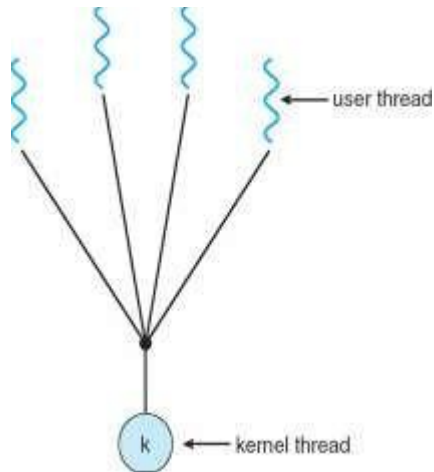
- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

## **Kernel Thread**

- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

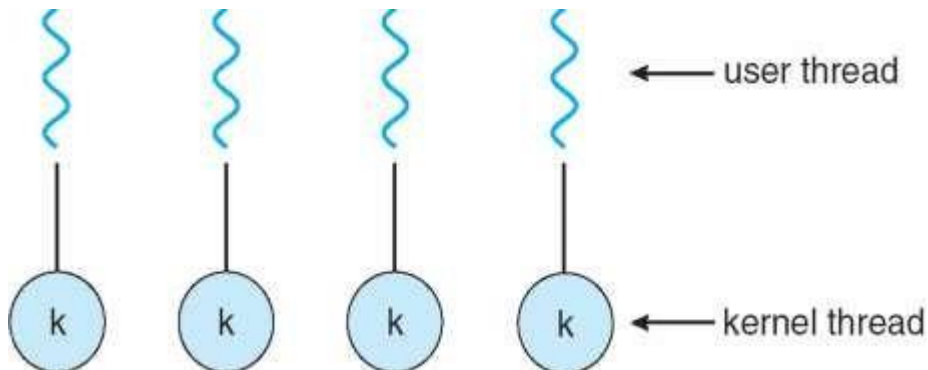
## **Multithreading Models**

- Many-to-One
  - Many user-level threads mapped to single kernel thread
  - Examples:
    - Solaris Green Threads
    - GNU Portable Threads



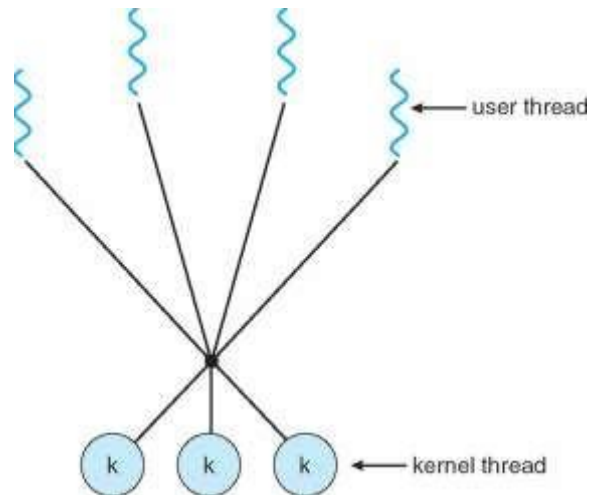
- One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



- Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



## Thread Library

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

## Pthreads

- May be provided either as user-level or kernel-level
- 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)

## Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

## Threading Issues

- Semantics of fork() and exec() system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

## 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

## 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 Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process

- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

## Difference between Process and Thread

Process	Thread
Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

## Process Scheduling

- Maximum CPU utilization obtained with multiprogramming
- CPU-I/O Burst Cycle - Process execution consists of a *cycle* of CPU execution and I/O wait
- **CPU burst** distribution

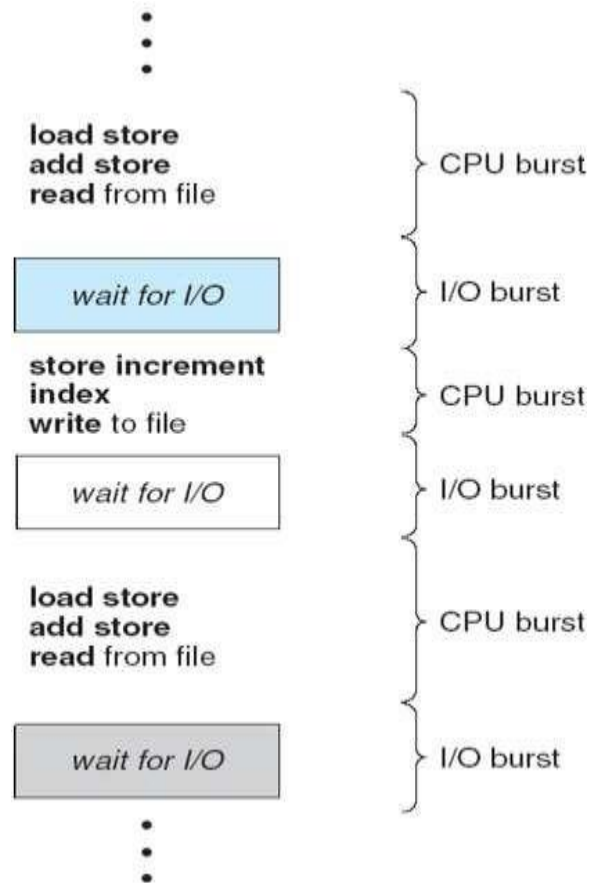


Fig: CPU burst and I/O burst

### CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

### Dispatcher



- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- Dispatch latency - time it takes for the dispatcher to stop one process and start another running

### **CPU Scheduling Criteria**

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

### **CPU Scheduling Algorithms**

#### **A. First Come First Serve Scheduling**

- Schedule the task first which arrives first
- Non preemptive In nature

#### **B. Shortest Job First Scheduling**

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal - gives minimum average waiting time for a given set of processes

- The difficulty is knowing the length of the next CPU request

### **Priority Scheduling**

- A priority number (integer) is associated with each process

- The CPU is allocated to the process with the highest priority (smallest integer  $\equiv$  highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem  $\equiv$  **Starvation** - low priority processes may never execute
- Solution  $\equiv$  **Aging** - as time progresses increase the priority of the process

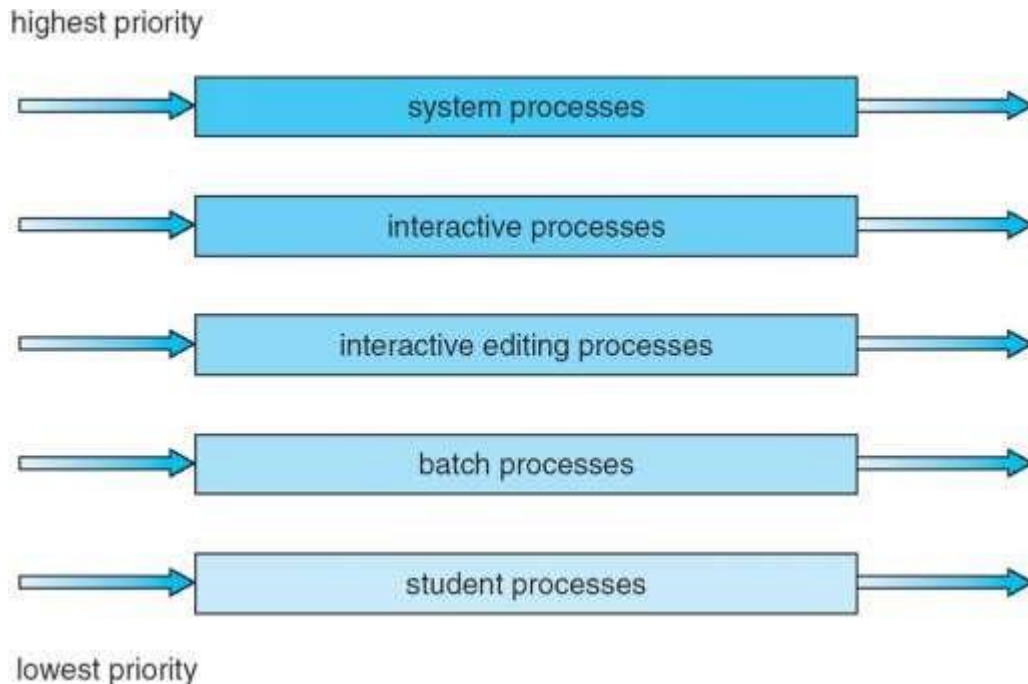
### Round Robin Scheduling

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are  $n$  processes in the ready queue and the time quantum is  $q$ , then each process gets  $1/n$  of the CPU time in chunks of at most  $q$  time units at once. No process waits more than  $(n-1)q$  time units.
- Performance
  - $q$  large  $\Rightarrow$  FIFO
  - $q$  small  $\Rightarrow q$  must be large with respect to context switch, otherwise overhead is too high

### Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)
- Each queue has its own scheduling algorithm
  - foreground - RR
  - background - FCFS

- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice - each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS



### Multilevel Feedback Queue Scheduling

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

## MODULE-II

### Process Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

<pre>Pseudocode for Producer Process while (true) {     /* produce an item and put in     nextProduced */     while (count == BUFFER_SIZE)         ; // do nothing     buffer [in] = nextProduced;     in = (in + 1) % BUFFER_SIZE;     count++; }</pre>	<pre>Pseudocode for Consumer Process while (true) {     while (count == 0)         ; // do nothing     nextConsumed = buffer[out];     out = (out + 1) %     BUFFER_SIZE;     count--;     /* consume the item in     nextConsumed }</pre>
--	--

data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.

- count++ could be implemented as

```
register1 = count
```

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

- count-- could be implemented as

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

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

```
S0: producer execute register1 = count {register1 = 5}  
S1: producer execute register1 = register1 + 1 {register1 = 6}  
S2: consumer execute register2 = count {register2 = 5}  
S3: consumer execute register2 = register2 - 1 {register2 = 4}  
S4: producer execute count = register1 {count = 6}  
S5: consumer execute count = register2 {count = 4}
```

### **Critical Section Problem**

A section of code, common to n cooperating processes, in which the processes may be accessing common variables.

A Critical Section Environment contains:

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

```

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

```

General structure of a typical process  $P_i$ .

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

### Solution to Critical Section Problem

1. Mutual Exclusion - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the  $N$  processes

## Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P<sub>i</sub> is ready!

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

## Hardware Synchronization

- Many systems provide hardware support for critical section code
- Uniprocessors - could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable

- Either test memory word and set value
- Or swap contents of two memory words

### **Solution to Critical Section Problem using Lock**

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

### **TestAndndSet Instruction**

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;

    *target = TRUE;

    return rv;
}
```

### **Solution using TestAndSet**

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {
    while ( TestAndSet (&lock ))
        ; // do nothing
    // critical section

    lock = FALSE;

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



## Swap Instruction

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

## Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

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

    // critical section

    lock = FALSE;

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

## Bounded-waiting Mutual Exclusion with TestandSet()

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

```

        key = TestAndSet(&lock);
waiting[i] = FALSE;

        // critical section

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

        // remainder section

    } while (TRUE);

```

## Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore  $S$  - integer variable
- Two standard operations modify  $S$ : wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```

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

    S--;
}

```

```

signal (S) {
    S++;
}

```

- Counting semaphore - integer value can range over an unrestricted domain

- Binary semaphore - integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

Semaphore mutex; // initialized to 1

```
do {
    wait (mutex);

    // Critical Section

    signal (mutex);

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

### **Semaphore Implementation**

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.
- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:

- block - place the process invoking the operation on the appropriate waiting queue.
- wakeup - remove one of processes in the waiting queue and place it in the ready queue.

- Implementation of wait:

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

- Implementation of signal:

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

### **Classical Problems of Synchronization**

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

## Bounded-Buffer Problem

*The pool consists of  $n$  buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value  $n$ ; the semaphore full is initialized to the value 0.*

- $N$  buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value  $N$ .
- The structure of the producer process

do {

    // produce an item in nextp

    wait (empty);

    wait (mutex);

    // add the item to the buffer

    signal (mutex);

    signal (full);

  } while (TRUE);

- The structure of the consumer process

do {

    wait (full);

    wait (mutex);

    // remove an item from buffer to nextc

    signal (mutex);

    signal (empty);

```
        // consume the item in nextc  
    } while (TRUE);
```

## Readers-Writers Problem

*Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers**. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.*

- A data set is shared among a number of concurrent processes
  - Readers - only read the data set; they do **not** perform any updates
  - Writers - can both read and write
- Problem - allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0
- The structure of a writer process

```
do {  
    wait (wrt) ;
```

```

        // writing is performed
        signal (wrt) ;
    } while (TRUE);

```

- The structure of a reader process

```

do {
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)
        wait (wrt) ;
    signal (mutex)

    // reading is performed
    wait (mutex) ;
    readcount - - ;
    if (readcount == 0)
        signal (wrt) ;
    signal (mutex) ;
} while (TRUE);

```

### **Dining-Philosophers Problem**

*Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a*

*chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.*

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1
- The structure of Philosopher  $i$ :

```
do {
    wait ( chopstick[i] );
    wait ( chopStick[ (i + 1) % 5] );

    // eat
    signal ( chopstick[i] );
    signal ( chopstick[ (i + 1) % 5] );

    // think
} while (TRUE);
```

## Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

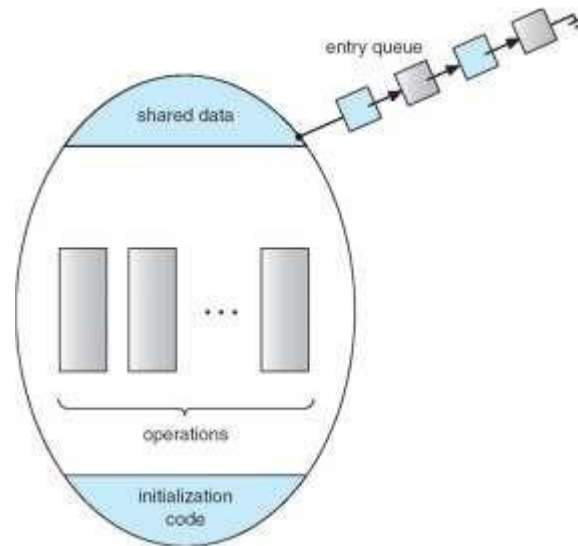
monitor monitor-name

```
{
    // shared variable declarations
    procedure P1 (...) { .... }
    ...
    procedure Pn (...) {.....}
    Initialization code ( ....) { ... }
    ...
}
```



}

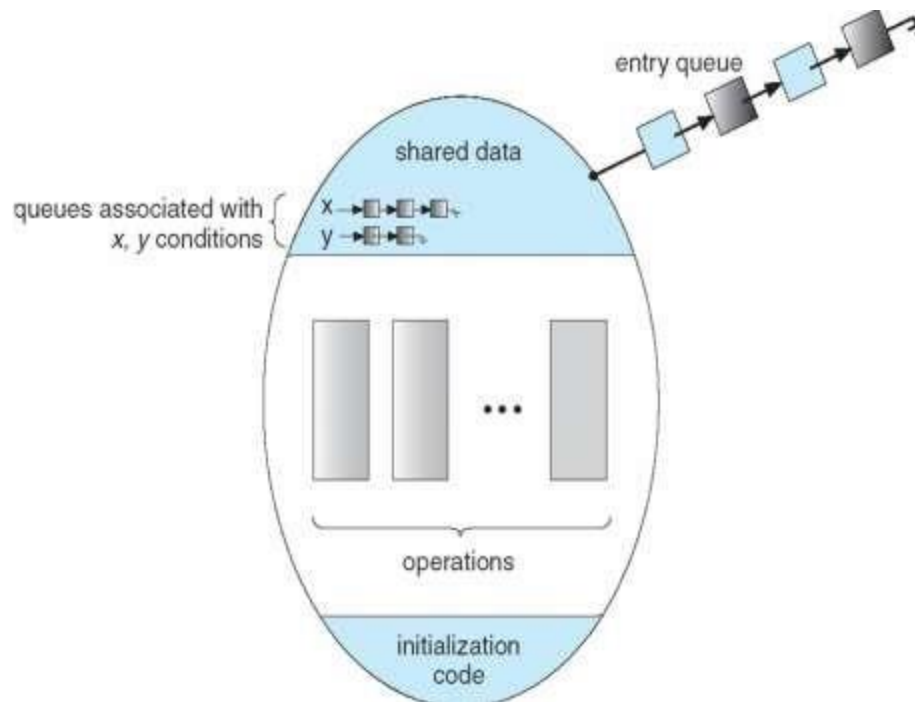
## Schematic view of a Monitor



## Condition Variables

- condition  $x, y$ ;
- Two operations on a condition variable:
  - $x.\text{wait}()$  - a process that invokes the operation is suspended.
    - $x.\text{signal}()$  - resumes one of processes (if any) that invoked  $x.\text{wait}()$

## Monitor with Condition Variables



## Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next-count = 0;
```

- Each procedure  $F$  will be replaced by

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

- Mutual exclusion within a monitor is ensured.

### Monitor Implementation

For each condition variable  $x$ , we have:

```
semaphore x_sem; // (initially = 0)
int x-count = 0;
```

The operation  $x$ .wait can be implemented as:

```
x-count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x-count--;
```

The operation  $x$ .signal can be implemented as:

```
if (x-count > 0) {
```

```

        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
    }

```

## Deadlock

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - System has 2 disk drives
  - $P_1$  and  $P_2$  each hold one disk drive and each needs another one
- Example
  - semaphores  $A$  and  $B$ , initialized to 1

$P_0$	$P_1$
wait (A);	wait(B)
wait (B);	wait(A)

## System Model

- Resource types  $R_1, R_2, \dots, R_m$  (*CPU cycles, memory space, I/O devices*)
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

## Deadlock Characterization


- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ , ...,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_0$  is waiting for a resource that is held by  $P_0$ .

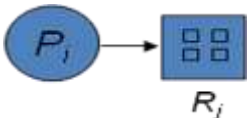
## Resource Allocation Graph

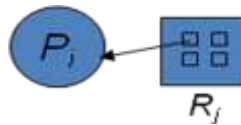
- A set of vertices  $V$  and a set of edges  $E$ .
- $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system

- request edge - directed edge  $P_i \rightarrow R_j$
- assignment edge - directed edge  $R_j \rightarrow P_i$

- Process 

- Resource Type with 4 instances 

- $P_i$  requests instance of  $R_j$  



- $P_i$  is holding an instance of  $R_j$

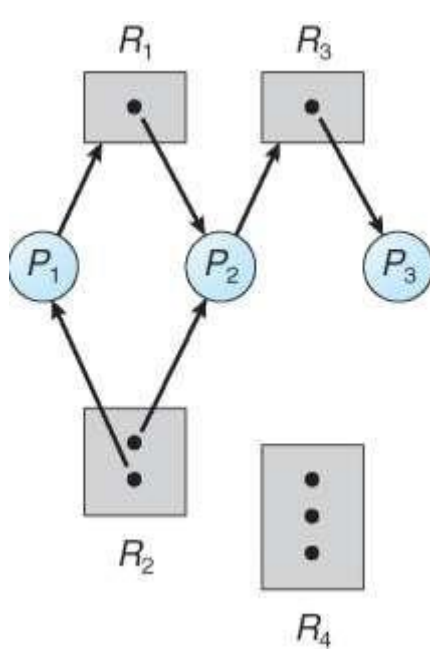


Fig: RAG

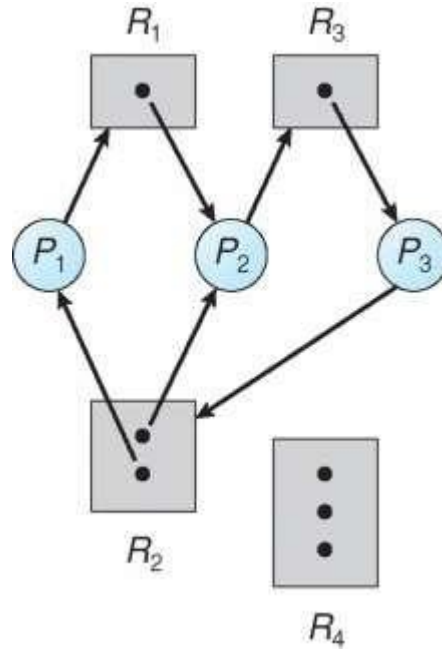


Fig: RAG with a deadlock

- If graph contains no cycles  $\Rightarrow$  no deadlock
- If graph contains a cycle  $\Rightarrow$ 
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

### Methods for Handling Deadlock

- Ensure that the system will *never* enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

### Deadlock Prevention

- **Mutual Exclusion** - not required for sharable resources; must hold for nonsharable resources

- **Hold and Wait** - must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
  - Low resource utilization; starvation possible
- **No Preemption** -
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

### Deadlock Avoidance

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

### Safe state

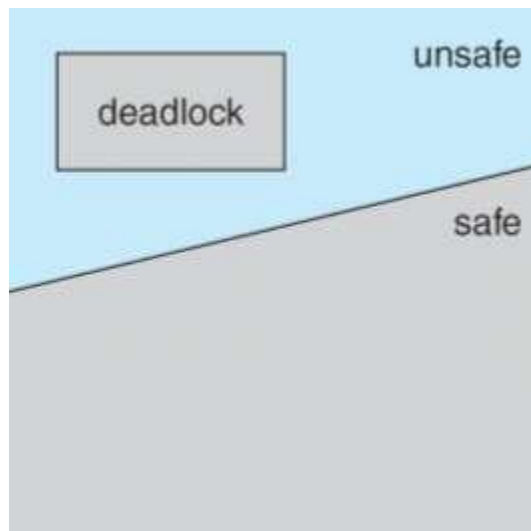
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence  $\langle P_1, P_2, \dots, P_n \rangle$  of ALL the processes in the system such that for each  $P_i$ , the resources that  $P_i$  can still

request can be satisfied by currently available resources + resources held by all the  $P_j$ , with  $j < i$

- That is:
  - If  $P_i$  resource needs are not immediately available, then  $P_i$  can wait until all  $P_j$  have finished
  - When  $P_j$  is finished,  $P_i$  can obtain needed resources, execute, return allocated resources, and terminate
  - When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on

### Facts

- |   |
|---|
| <ul style="list-style-type: none"><li>n <b>If a system is in safe state <math>\Rightarrow</math> no deadlocks</b></li><li>n <b>If a system is in unsafe state <math>\Rightarrow</math> possibility of deadlock</b></li><li>n <b>Avoidance <math>\Rightarrow</math> ensure that a system will never enter an unsafe state.</b></li></ul> |
|---|

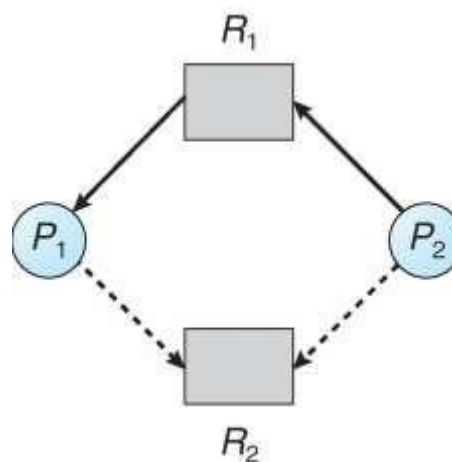


### Deadlock Avoidance Algorithm

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

## RAG Scheme

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



## Banker's Algorithm

### Assumptions

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

### Data Structure for Bankers' Algorithm

Let  $n$  = number of processes, and  $m$  = number of resources types.

- **Available:** Vector of length  $m$ . If available  $[j] = k$ , there are  $k$  instances of resource type  $R_j$  available
- **Max:**  $n \times m$  matrix. If  $Max [i,j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$



- **Allocation:**  $n \times m$  matrix. If  $\text{Allocation}[i,j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$
- **Need:**  $n \times m$  matrix. If  $\text{Need}[i,j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to complete its task

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

### Safety Algorithm

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

*Work* = *Available*

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

2. Find an  $i$  such that both:

(a) *Finish* [ $i$ ] = false

(b)  $\text{Need}_i \leq \text{Work}$

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

3. *Work* = *Work* + *Allocation* <sub>$i$</sub>

*Finish* [ $i$ ] = true

go to step 2

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

### Resource Request Algorithm

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

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

*Available* = *Available* - *Request* <sub>$i$</sub> ;

*Allocation* <sub>$i$</sub>  = *Allocation* <sub>$i$</sub>  + *Request* <sub>$i$</sub> ;

*Need* <sub>$i$</sub>  = *Need* <sub>$i$</sub>  - *Request* <sub>$i$</sub> ;

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

## Deadlock Detection

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

## Recovery from Deadlock

### A. Process Termination

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

### B. Resource Preemption

- Selecting a victim - minimize cost
- Rollback - return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

## Memory Management

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
- A pair of **base** and **limit** registers define the logical address space

## Logical vs Physical Address Space

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

## Address Binding

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

## Memory Management Unit

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

## Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded

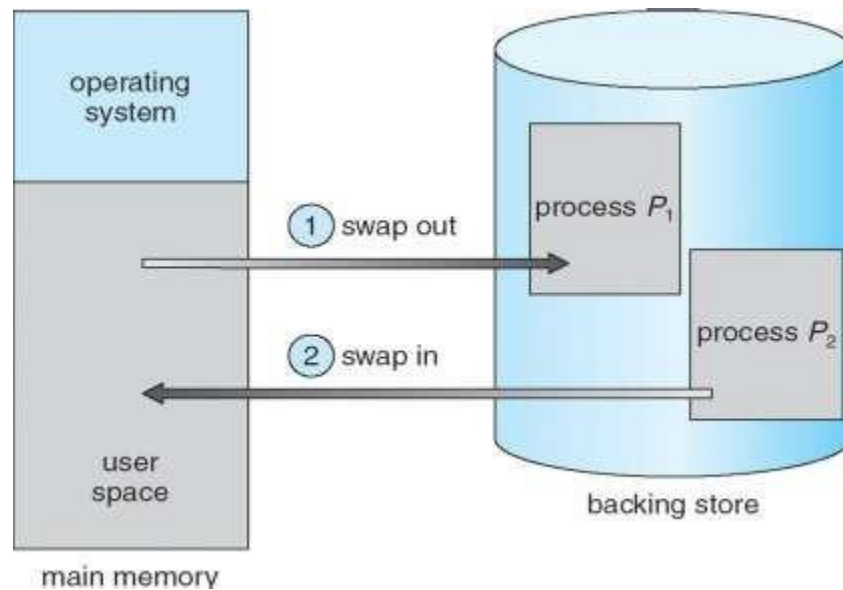
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

## Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

## Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk



## Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses - each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
- Multiple-partition allocation
  - Hole - block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about:
  - a) allocated partitions    b) free partitions (hole)

## Dynamic Storage Allocation Problem

- **First-fit:** Allocate the *first* hole that is big enough
- **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit:** Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

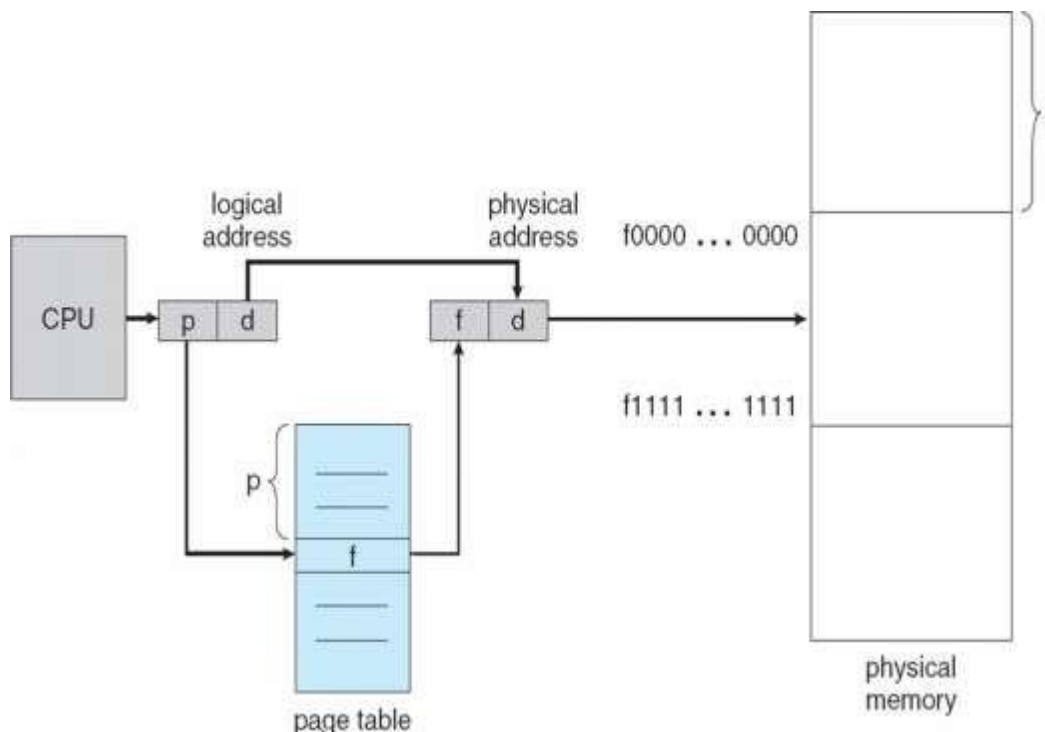
## Fragmentation

- **External Fragmentation** - total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** - allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

## Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames

- To run a program of size  $n$  pages, need to find  $n$  free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation
- Address generated by CPU is divided into:
  - **Page number ( $p$ )** - used as an index into a *page table* which contains base address of each page in physical memory
  - **Page offset ( $d$ )** - combined with base address to define the physical memory address that is sent to the memory unit

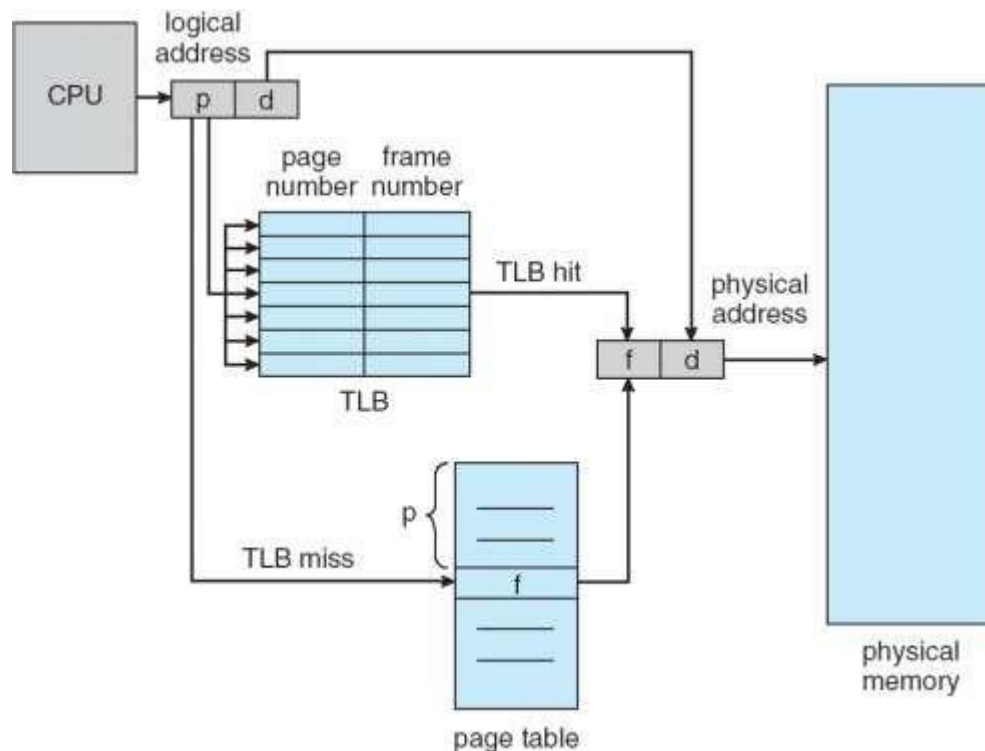


### Implementation of Page table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

### Paging with TLB



### Memory Protection

- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space

### Shared Pages

- **Shared code**

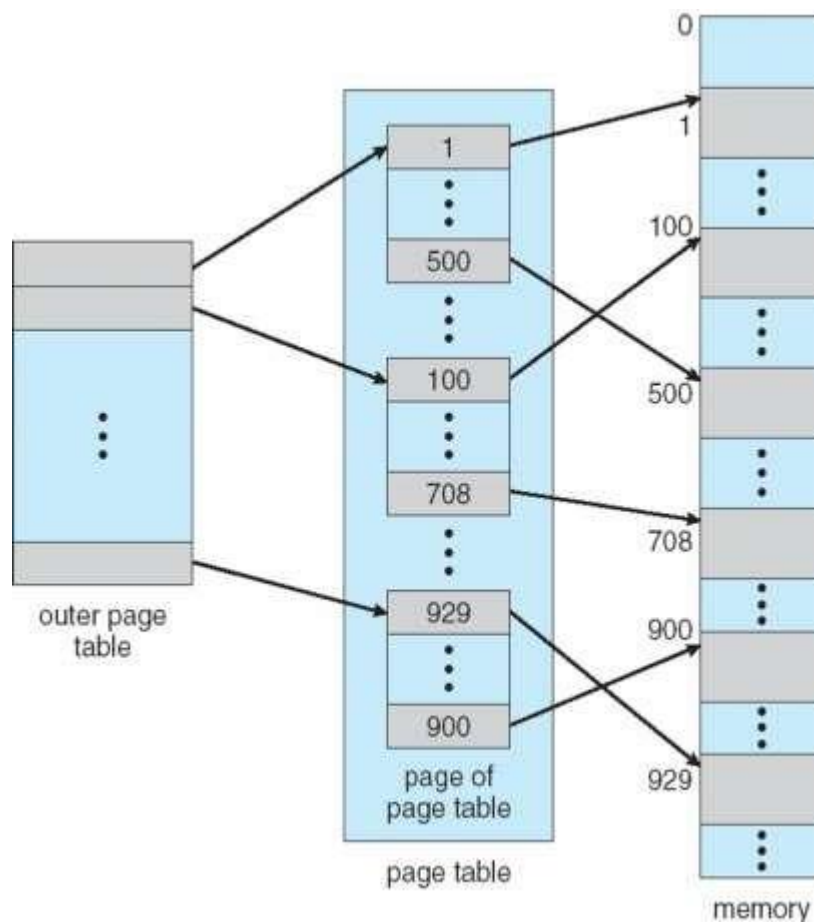


- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Shared code must appear in same location in the logical address space of all processes
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

## Structure of Page table

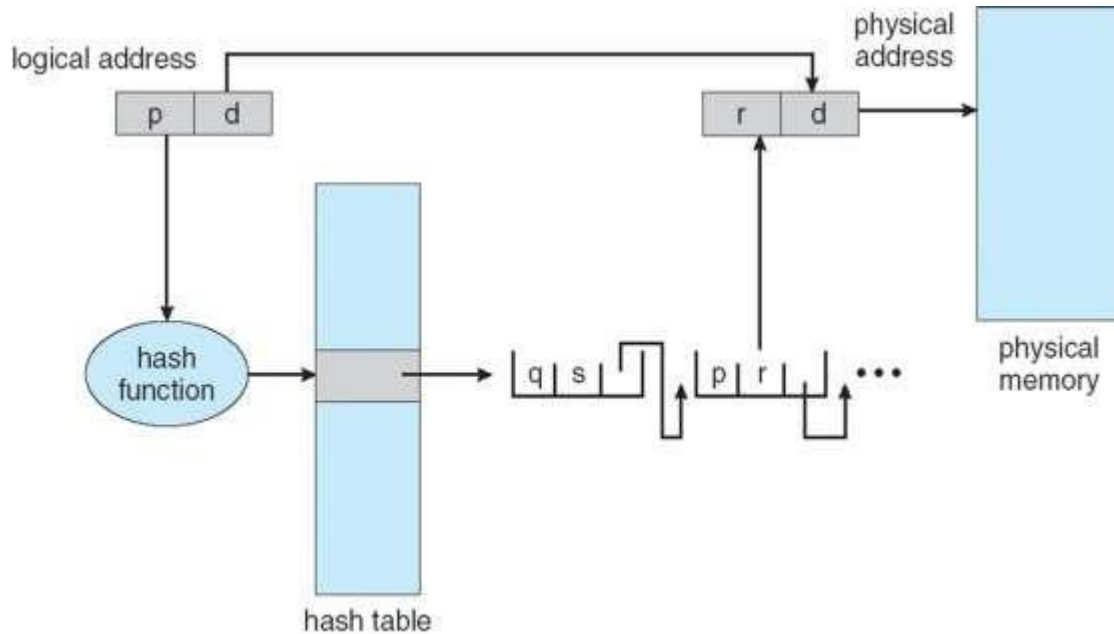
### Hierarchical Paging

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table



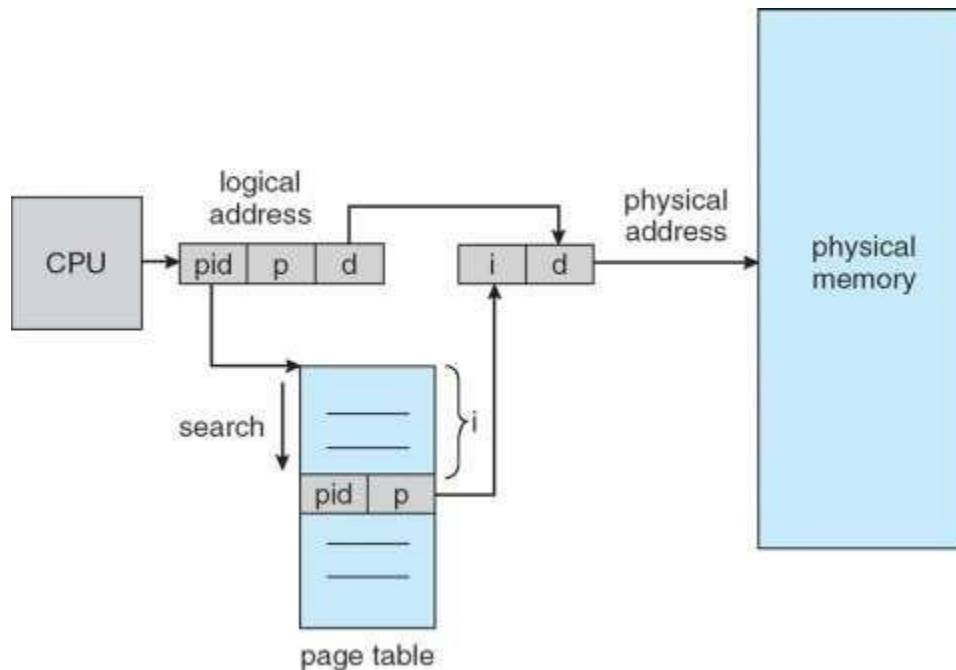
## Hashed Page Tables

- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted



## Inverted Page Tables

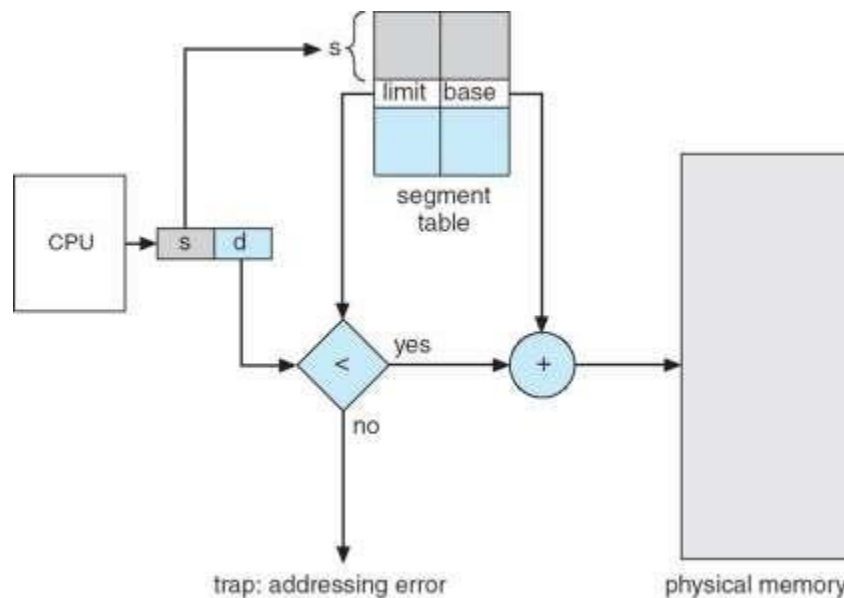
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one – or at most a few – page-table entries



## Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as: main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays
- Logical address consists of a two tuple:
  - $\langle \text{segment-number}, \text{offset} \rangle$ ,
- Segment table – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory
  - limit – specifies the length of the segment
- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates number of segments used by a program;
  - segment number  $s$  is legal if  $s < \text{STLR}$
- Protection

- With each entry in segment table associate:
  - validation bit = 0  $\Rightarrow$  illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram



## Virtual Memory Management

- **Virtual memory** - separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

## Demand Paging

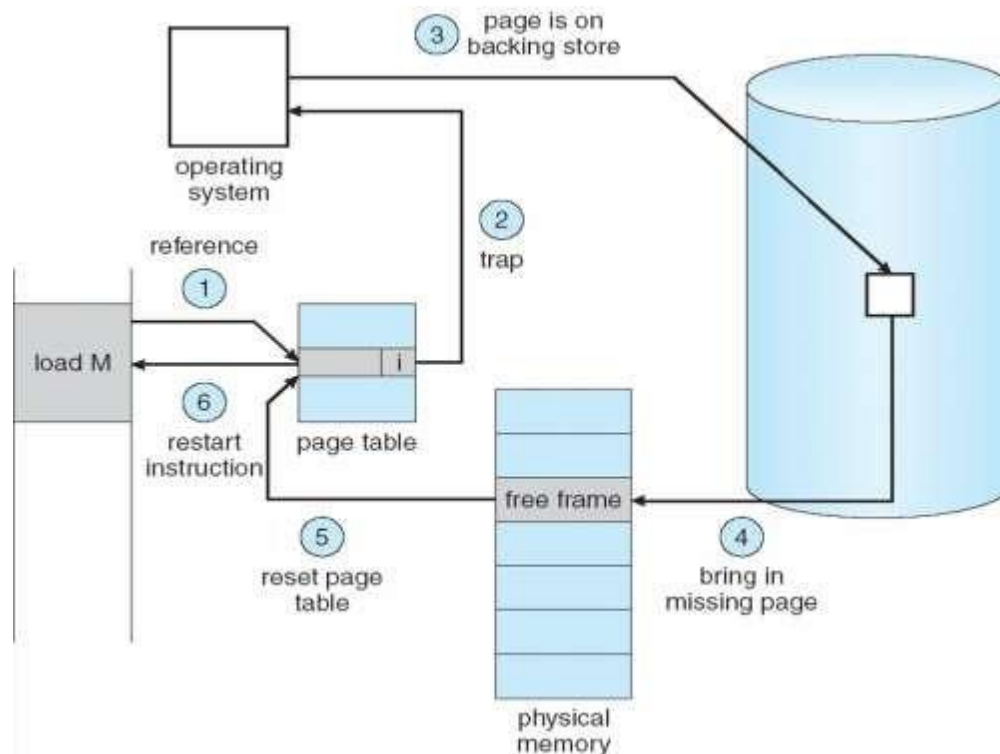
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed  $\Rightarrow$  reference to it
  - invalid reference  $\Rightarrow$  abort
  - not-in-memory  $\Rightarrow$  bring to memory
- **Lazy swapper** - never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**
- With each page table entry a valid-invalid bit is associated (**v**  $\Rightarrow$  in-memory, **i**  $\Rightarrow$  not-in-memory)
- Initially valid-invalid bit is set to **i** on all entries
- During address translation, if valid-invalid bit in page table entry is **I**  $\Rightarrow$  page fault

## Page Fault

If there is a reference to a page, first reference to that page will trap to operating system: **page fault**

1. Operating system looks at another table to decide:
  - 1 Invalid reference  $\Rightarrow$  abort
  - 1 Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**

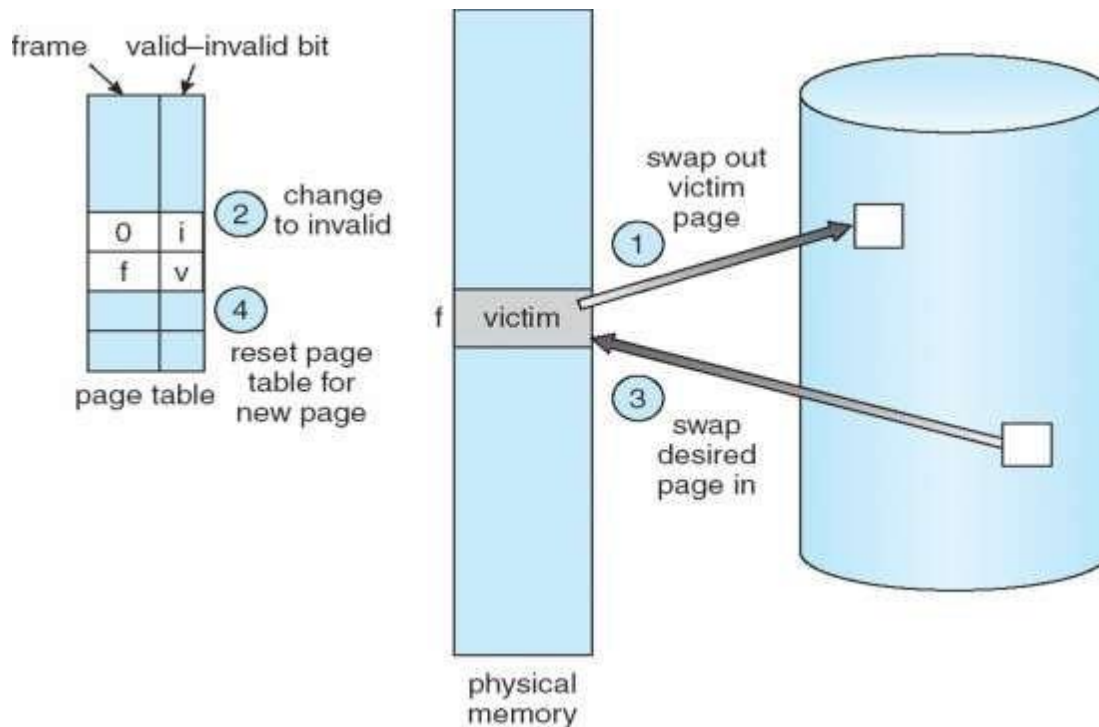
## 6. Restart the instruction that caused the page fault



## Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use modify (dirty) bit to reduce overhead of page transfers - only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory - large virtual memory can be provided on a smaller physical memory
- Find the location of the desired page on disk
- Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a victim frame
- Bring the desired page into the (newly) free frame; update the page and frame tables

- Restart the process



## Page Replacement algorithm

### FIFO (First-in-First-Out)

- A FIFO replacement algorithm associates with each page the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- Belady's Anomaly: more frames  $\Rightarrow$  more page faults** ( for some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases.)
- Ex-**

reference string

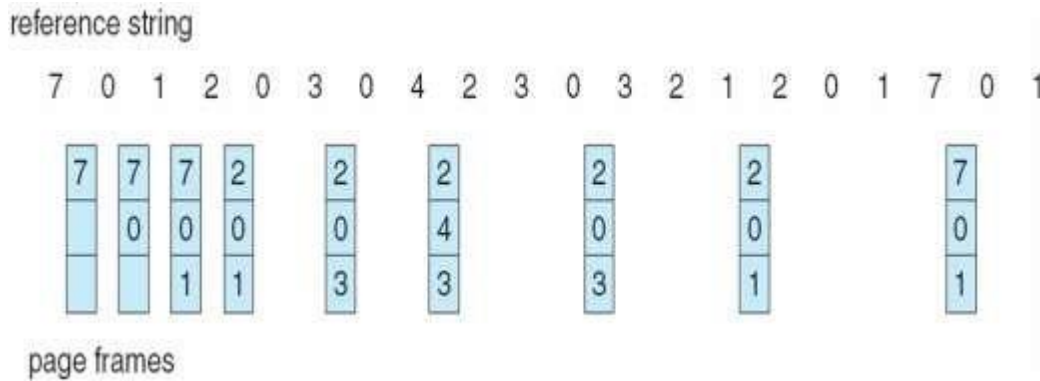
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

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

page frames

## OPTIMAL PAGE REPLACEMENT

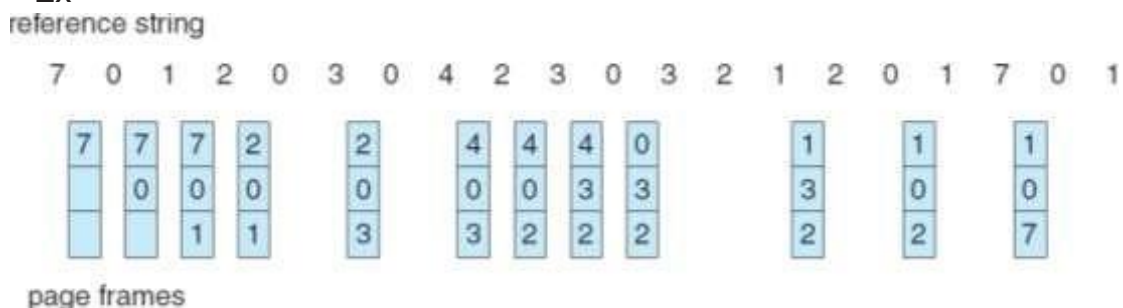
- Replace page that will not be used for longest period of time
- Ex-



## LRU (LEAST RECENTLY USED)

- LRU replacement associates with each page the time of that page's last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.

- Ex-



## Allocation of Frames

- Each process needs *minimum* number of pages
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation - Allocate according to the size of process



$s_i$  = size of process  $p_i$

$$S = \sum s_i$$

$m$  = total number of frames

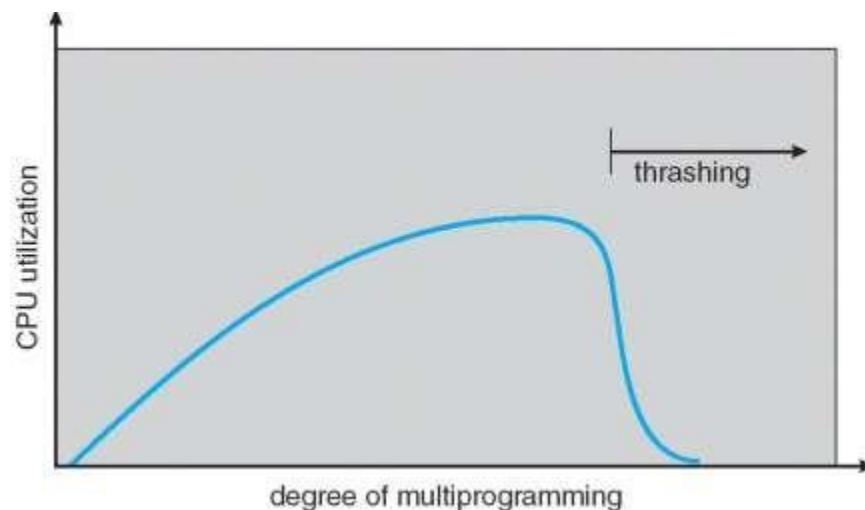
$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

### Global vs Local Allocation

- Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another
- Local replacement – each process selects from only its own set of allocated frames

### Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing**  $\equiv$  a process is busy swapping pages in and out



## MODULE-III

### File System

#### File

- Contiguous logical address space
- Types:
  - Data
    - numeric
    - character
    - binary
  - Program

#### File Structure

- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
  - 1 Program

## File Attribute

- **Name** - only information kept in human-readable form
- **Identifier** - unique tag (number) identifies file within file system
- **Type** - needed for systems that support different types
- **Location** - pointer to file location on device
- **Size** - current file size
- **Protection** - controls who can do reading, writing, executing
- **Time, date, and user identification** - data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk

## File Types

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

## File Operations

- **Create, Write, Read, Reposition within file, Delete, Truncate**
- *Open( $F_i$ )* - search the directory structure on disk for entry  $F_i$ , and move the content of entry to memory
- *Close ( $F_i$ )* - move the content of entry  $F_i$  in memory to directory structure on disk

## File Access Methods

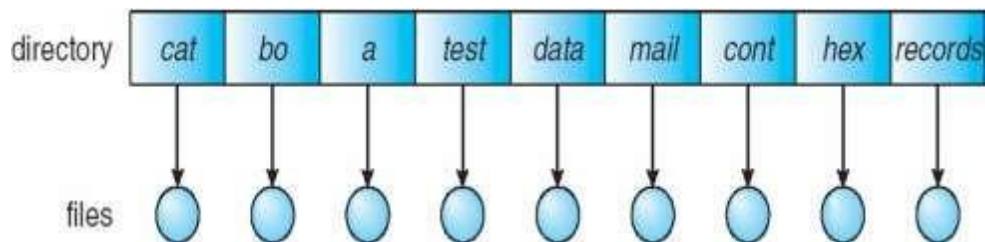
n Sequential Access	n Direct Access
read next	read $n$
write next	write $n$
reset	position to $n$
no read after last write	read next
(rewrite)	write next
	rewrite $n$
	$n$ = relative block number

sequential access	implementation for direct access
<i>reset</i>	$cp = 0;$
<i>read next</i>	$read\ cp;$ $cp = cp + 1;$
<i>write next</i>	$write\ cp;$ $cp = cp + 1;$

## Directory Structure

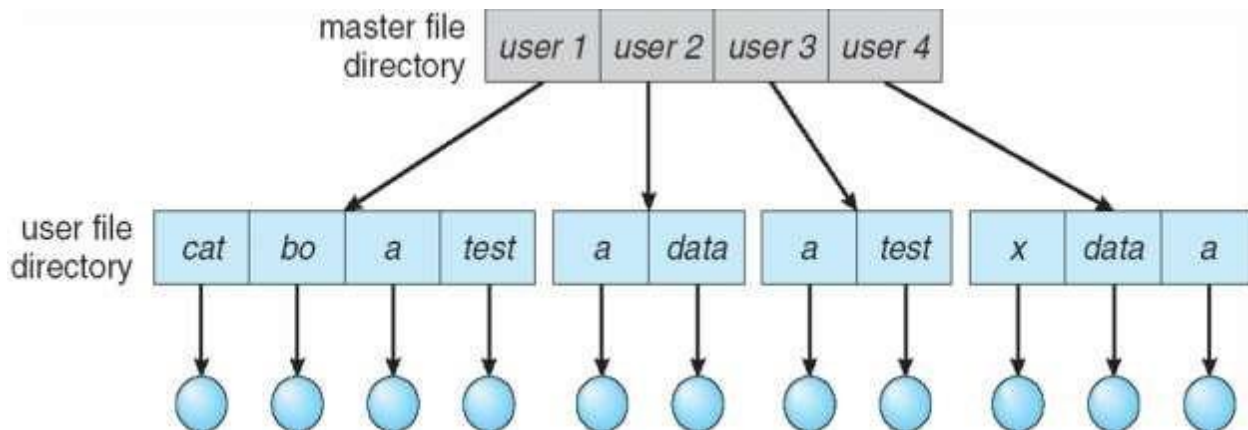
### A. Single Level Directory

- A single directory for all users
- Naming problem
- Grouping problem



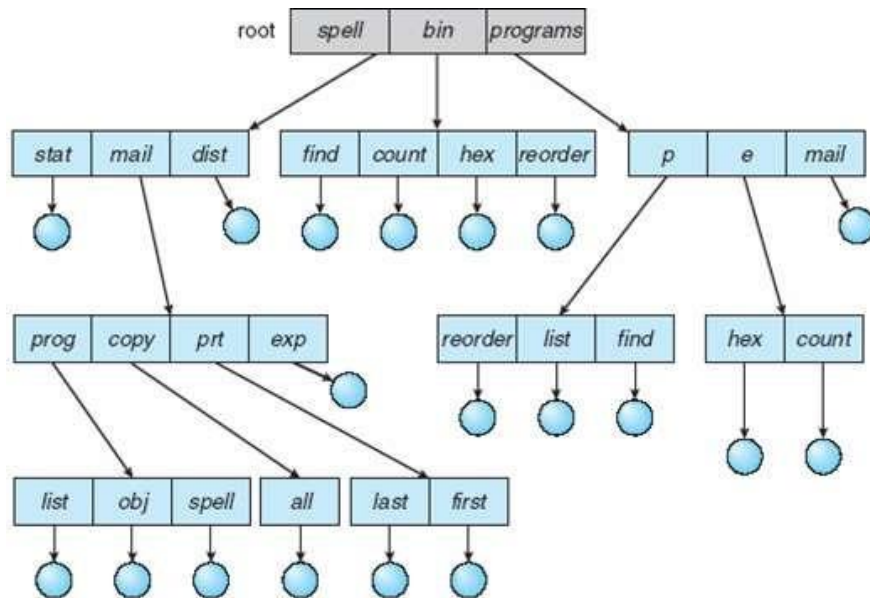
### B. Two Level Directory

- Separate directory for each user
- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

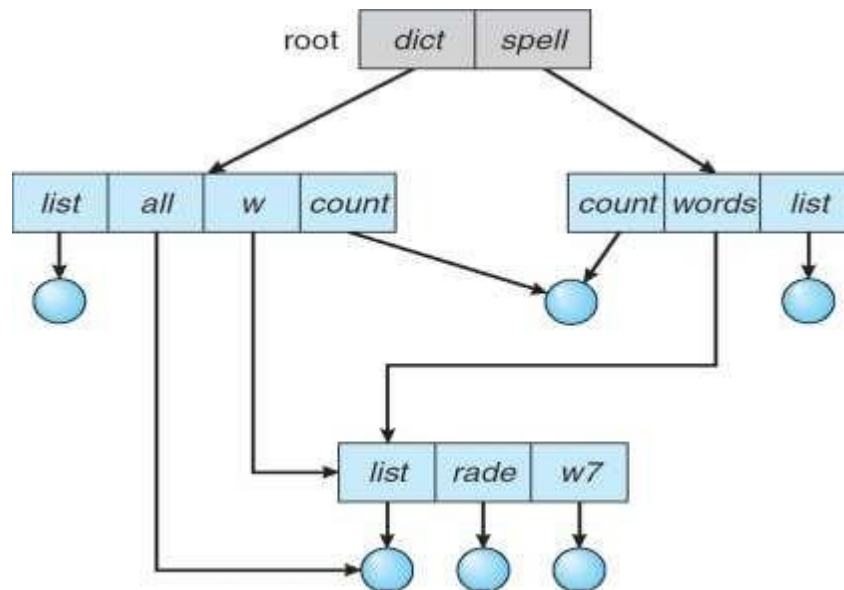


### C. Tree Structure Directory

- Efficient searching
- Grouping Capability



### D. Acyclic Graph Directories



- Have shared subdirectories and files

## File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- **User IDs** identify users, allowing permissions and protections to be per-user
- **Group IDs** allow users to be in groups, permitting group access rights
- Uses networking to allow file system access between systems
  - Manually via programs like FTP
  - Automatically, seamlessly using **distributed file systems**
  - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
  - Server can serve multiple clients
  - Client and user-on-client identification is insecure or complicated
  - **NFS** is standard UNIX client-server file sharing protocol
  - **CIFS** is standard Windows protocol
  - Standard operating system file calls are translated into remote calls
- Distributed Information Systems (**distributed naming services**) such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as NFS include all information in each request, allowing easy recovery but less security

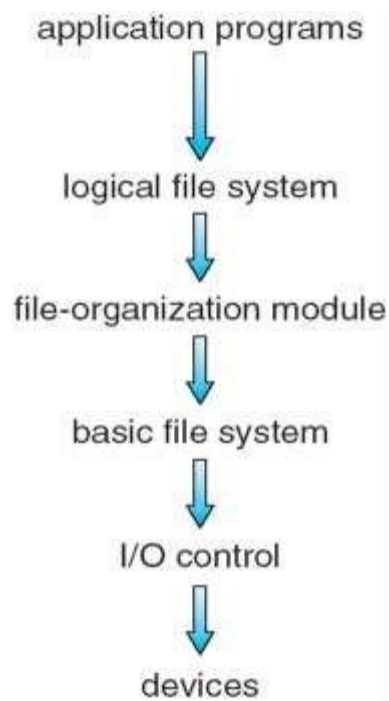
- **Consistency semantics** specify how multiple users are to access a shared file simultaneously
  - Similar to Ch 7 process synchronization algorithms
    - Tend to be less complex due to disk I/O and network latency (for remote file systems)
  - Andrew File System (AFS) implemented complex remote file sharing semantics
  - Unix file system (UFS) implements:
    - Writes to an open file visible immediately to other users of the same open file
    - Sharing file pointer to allow multiple users to read and write concurrently
  - AFS has session semantics
    - Writes only visible to sessions starting after the file is closed

## **File System Structure**

- File structure
  - Logical storage unit
  - Collection of related information
- n File system resides on secondary storage (disks)
- n File system organized into layers
- n **File control block** - storage structure consisting of information about a file



## Layered File System



## File Control Block

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

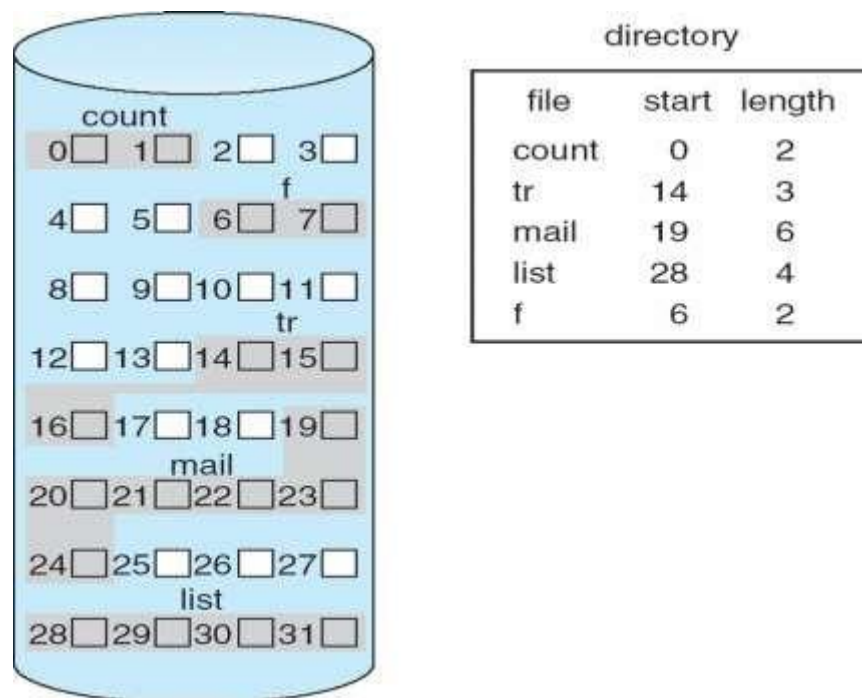
## File Allocation Methods

An allocation method refers to how disk blocks are allocated for files:

### A. Contiguous Allocation

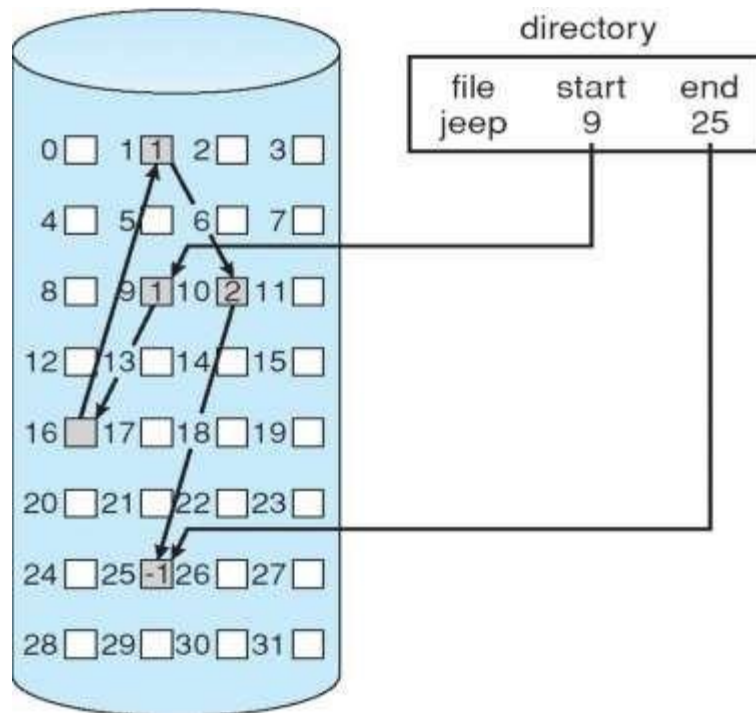
- n Each file occupies a set of contiguous blocks on the disk

- n Simple - only starting location (block #) and length (number of blocks) are required
- n Random access
- n Wasteful of space (dynamic storage-allocation problem)
- n Files cannot grow

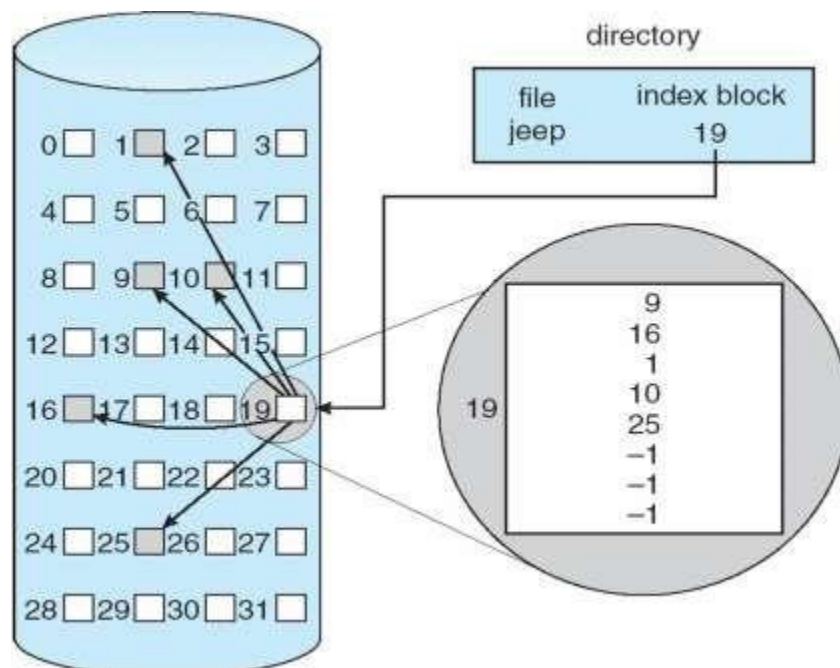


## B. Linked Allocation

- n Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
- n Simple - need only starting address
- n Free-space management system - no waste of space
- n No random access
- n Mapping



### C. Indexed Allocation



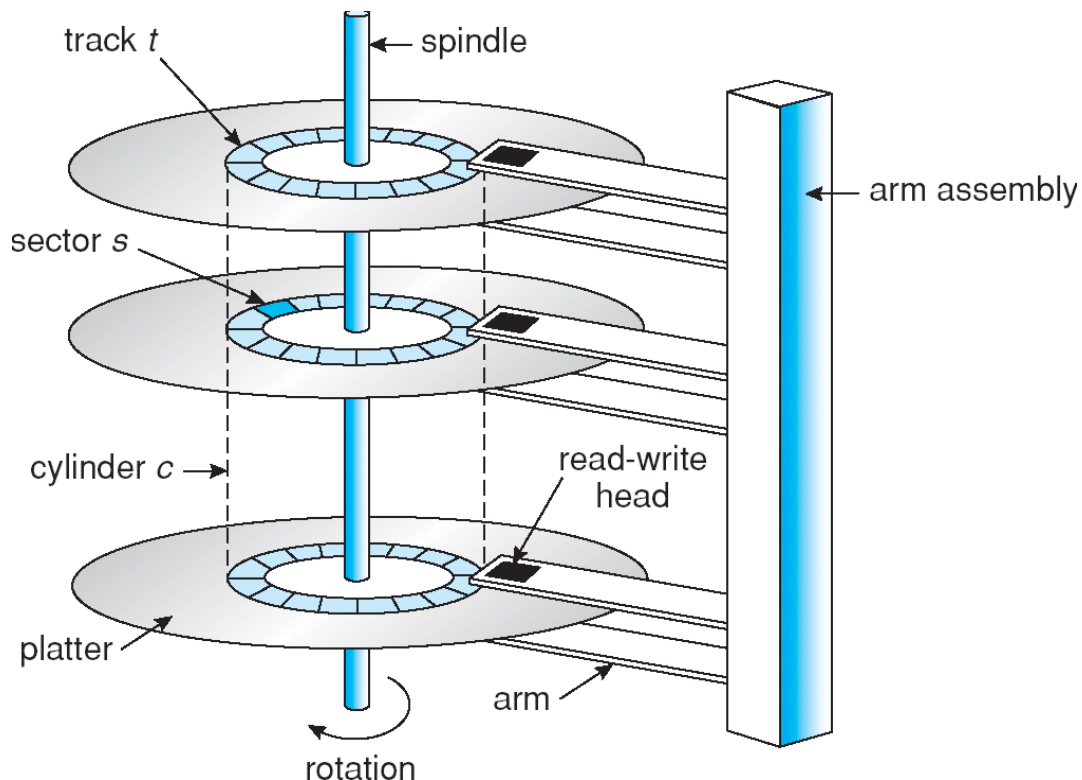
- n Brings all pointers together into the *index block*.
- n Need index table
- n Random access

- n Dynamic access without external fragmentation, but have overhead of index block.

## Secondary Storage Structure

### Magnetic Disk

- Magnetic disks provide bulk of secondary storage of modern computers
  - Drives rotate at 60 to 200 times per second
  - **Transfer rate** is rate at which data flow between drive and computer
  - **Positioning time (random-access time)** is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
  - **Head crash** results from disk head making contact with the disk surface
    - That's bad
- Disks can be removable
- Drive attached to computer via **I/O bus**
  - Busses vary, including **EIDE, ATA, SATA, USB, Fibre Channel, SCSI**
  - **Host controller** in computer uses bus to talk to **disk controller** built into drive or storage array



## Magnetic Tap

- Was early secondary-storage medium
- Relatively permanent and holds large quantities of data
- Access time slow
- Random access ~1000 times slower than disk
- Mainly used for backup, storage of infrequently-used data, transfer medium between systems
- Kept in spool and wound or rewound past read-write head
- Once data under head, transfer rates comparable to disk
- 20-200GB typical storage

## Disk Structure

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

## Disk Scheduling

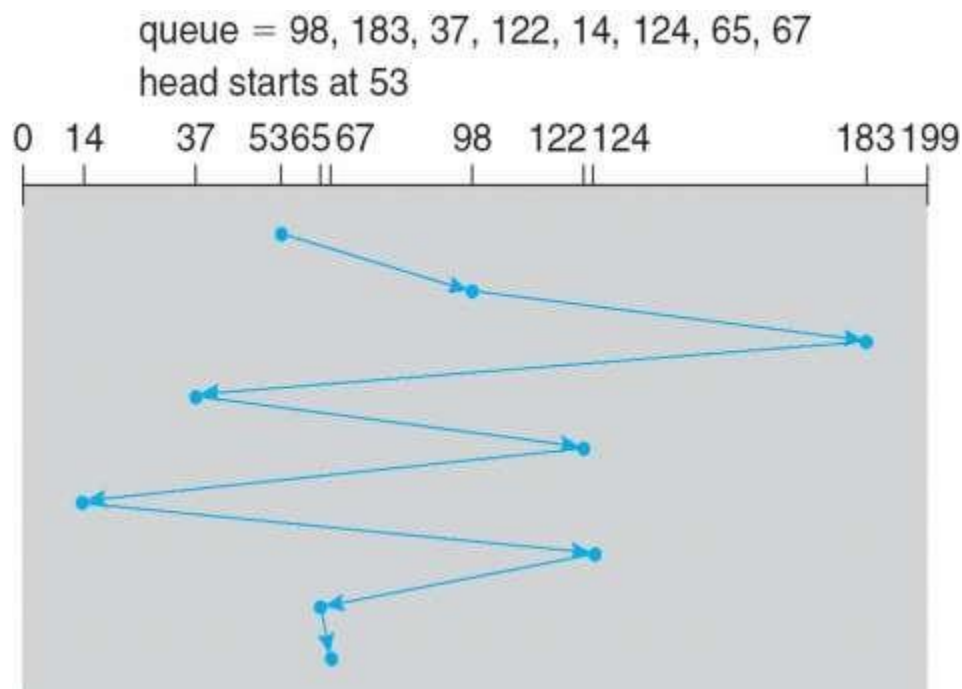
- The operating system is responsible for using hardware efficiently – for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components
  - *Seek time* is the time for the disk are to move the heads to the cylinder containing the desired sector.

- *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head.
- Minimize seek time
- Seek time  $\approx$  seek distance
- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

## Disk Scheduling Algorithms

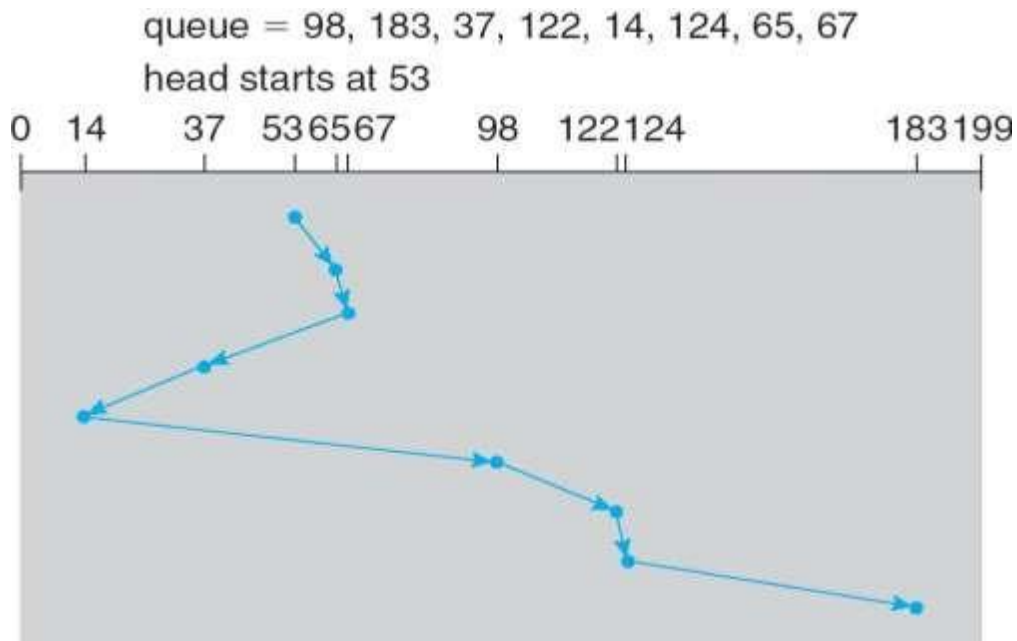
### FCFS

- This algorithm is intrinsically fair, but it generally does not provide the fastest service.



### SSTF (Shortest Seek Time First)

- n Selects the request with the minimum seek time from the current head position.
- n SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.



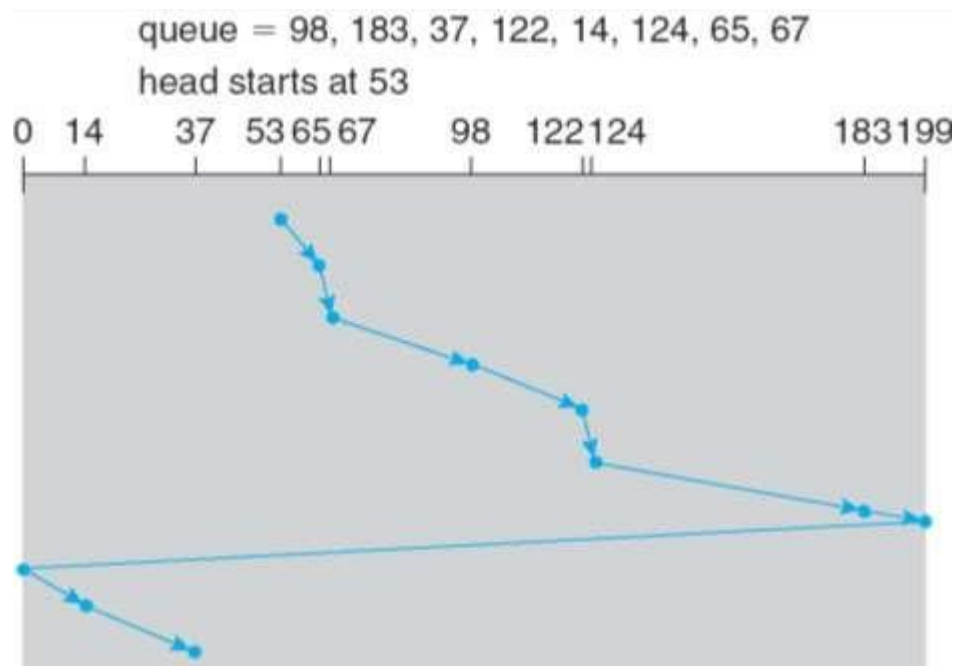
## SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Sometimes called the *elevator algorithm*.



## C-SCAN

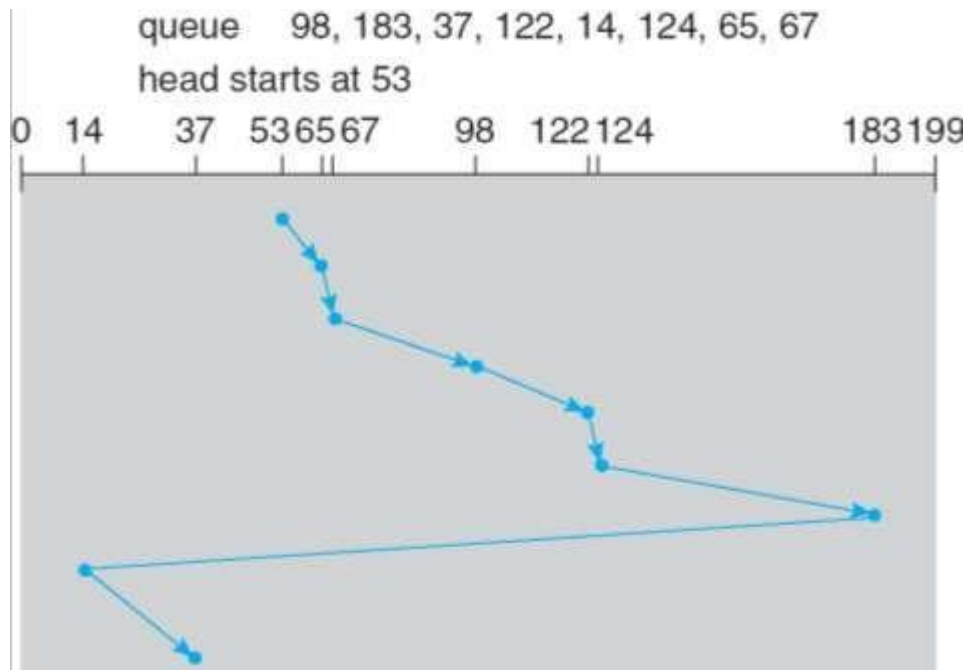
- Provides a more uniform wait time than SCAN.
- The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one



## C-LOOK

- Version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.





## Disk Management

- *Low-level formatting, or physical formatting* – Dividing a disk into sectors that the disk controller can read and write.
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk.
  - *Partition* the disk into one or more groups of cylinders.
  - *Logical formatting* or “making a file system”.
- Boot block initializes system.
  - The bootstrap is stored in ROM.
  - *Bootstrap loader* program.
- Methods such as *sector sparing* used to handle bad blocks.
- The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding**.

## Swap Space Management

- Swap-space – Virtual memory uses disk space as an extension of main memory.

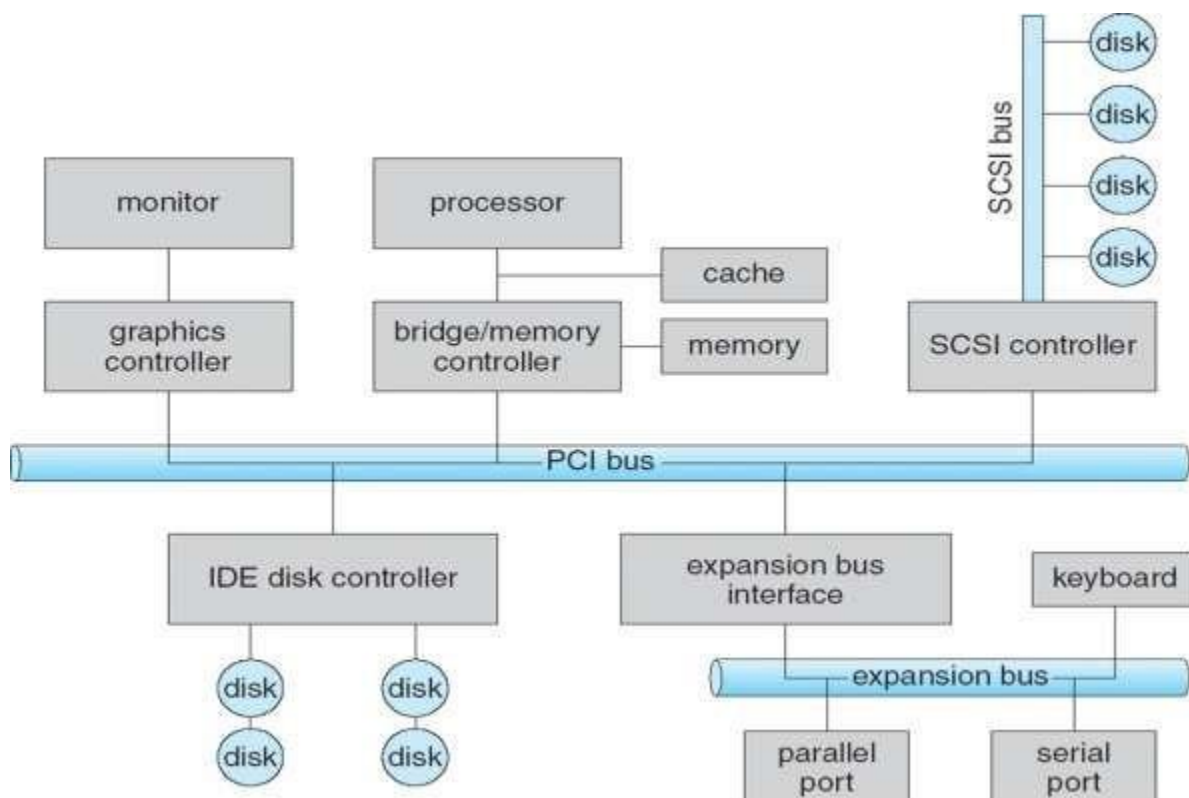
- Swap-space can be carved out of the normal file system or, more commonly, it can be in a separate disk partition.
- A swap space can reside in one of two places: it can be carved out of the normal file system, or it can be in a separate disk partition.
- If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space.
- Alternatively, swap space can be created in a separate **raw partition**. No file system or directory structure is placed in this space.
- A separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition.

## I/O Systems

### I/O Hardware

- A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point, or **port**—for example, a serial port.
- If devices share a common set of wires, the connection is called a bus.
- A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires.
- When device *A* has a cable that plugs into device *B*, and device *B* has a cable that plugs into device *C*, and device *C* plugs into a port on the computer, this arrangement is called a **daisy chain**. A daisy chain usually operates as a bus.
- A **PCI bus** (the common PC system bus) connects the processor-memory subsystem to fast devices, and an **expansion bus** connects relatively slow devices, such as the keyboard and serial and USB ports.
- Disks are connected together on a **Small Computer System Interface (SCSI)** bus plugged into a SCSI controller.

- A **controller** is a collection of electronics that can operate a port, a bus, or a device.
- A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port.
- But the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a **host adapter**) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages.

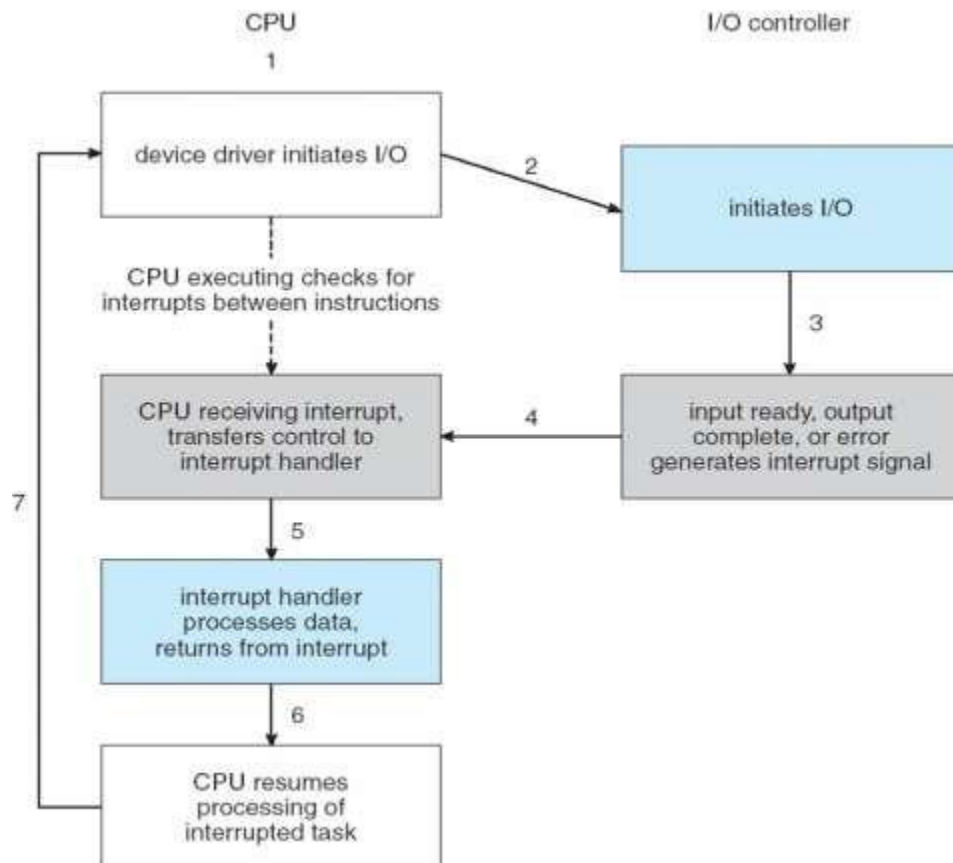


## Polling

- Determines state of device
  - command-ready
  - busy
  - Error
- **Busy-wait** cycle to wait for I/O from device

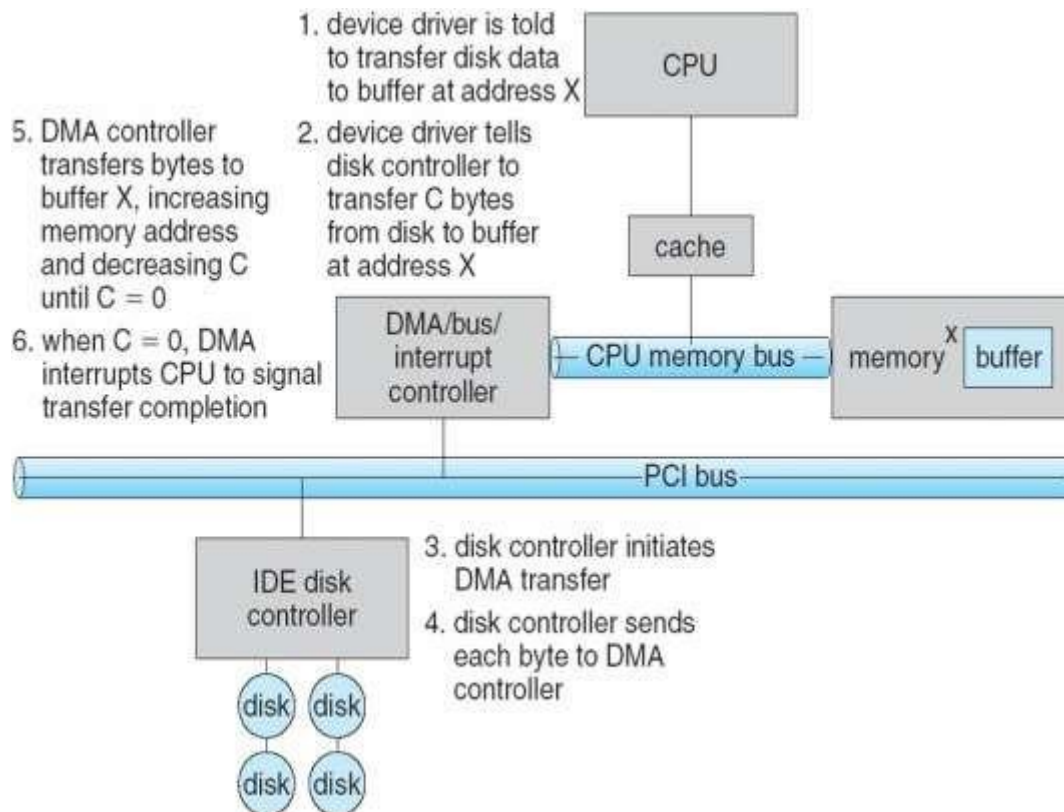
## Interrupt

- CPU **Interrupt-request line** triggered by I/O device
- **Interrupt handler** receives interrupts
- **Maskable** to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler
- Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors.
- The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.
- The maskable interrupt is used by device controllers to request service.
- Interrupt mechanism also used for exceptions



## Direct Memory Access

- Used to avoid **programmed I/O** for large data movement
- Requires **DMA** controller
- Bypasses CPU to transfer data directly between I/O device and memory

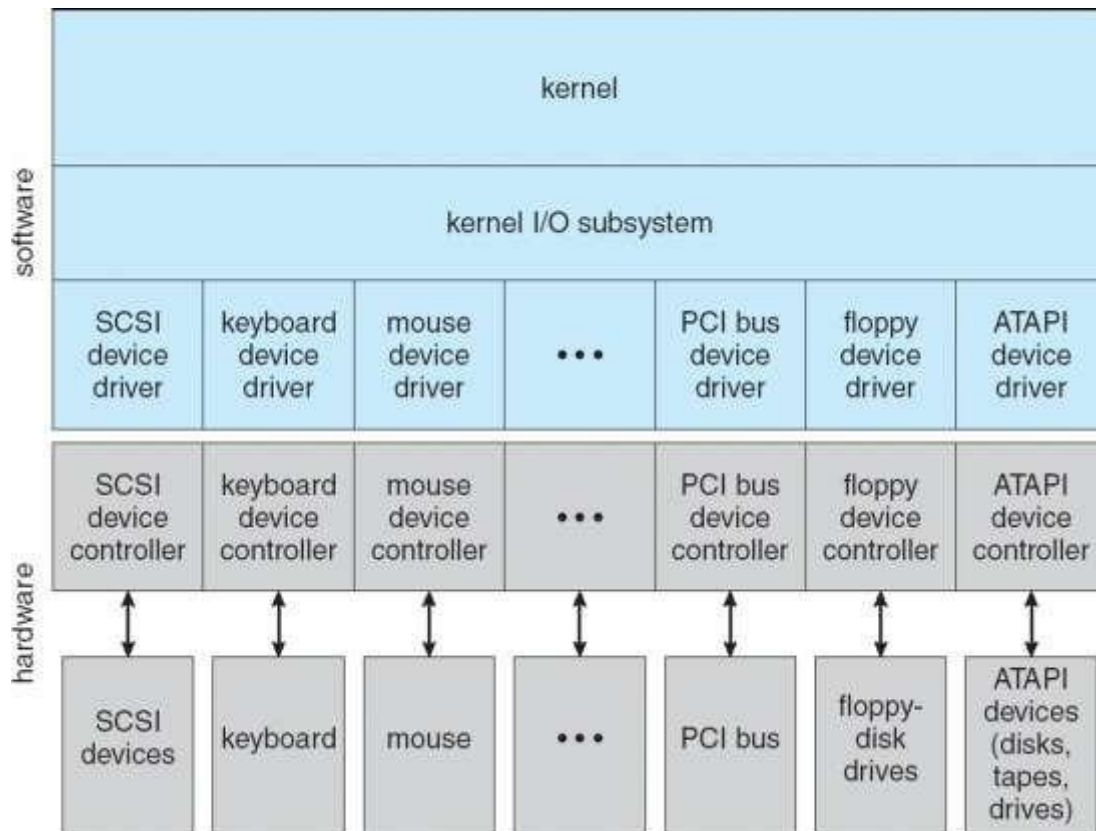


## Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel
- Devices vary in many dimensions
  - **Character-stream** or **block**
  - **Sequential** or **random-access**
  - **Sharable** or **dedicated**
  - **Speed of operation**

- read-write, read only, or write only

## Kernel I/O Structure



## Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read-write	CD-ROM graphics controller disk

## Block and Character Devices

- Block devices include disk drives
  - Commands include read, write, seek
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Character devices include keyboards, mice, serial ports
  - Commands include get, put
  - Libraries layered on top allow line editing

## Network Devices

- Varying enough from block and character to have own interface
- Unix and Windows NT/9x/2000 include socket interface
  - Separates network protocol from network operation
  - Includes select functionality
- Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

## Clock and Timers

- Provide current time, elapsed time, timer
- **Programmable interval timer** used for timings, periodic interrupts

## Blocking and Non-blocking I/O

- **Blocking** - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs
- **Nonblocking** - I/O call returns as much as available
  - User interface, data copy (buffered I/O)
  - Implemented via multi-threading

- Returns quickly with count of bytes read or written
- **Asynchronous** - process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed

## **Kernel I/O Subsystem**

- **Scheduling**
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness
- **Buffering** - store data in memory while transferring between devices
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain “copy semantics”
- **Caching** - fast memory holding copy of data
  - Always just a copy
  - Key to performance
- **Spooling** - hold output for a device
  - If device can serve only one request at a time
  - i.e., Printing
- **Device reservation** - provides exclusive access to a device
  - System calls for allocation and deallocation
  - 1 Watch out for deadlock

## **Reference**

Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin,  
 "Operating System Concepts, Ninth Edition ",