



# Unit - I

# Operating System

# Overview



# Operating System

- A program that controls the execution of application programs
- A program that controls the execution of users programs
- An interface between applications and hardware

# Operating System Objectives

- Convenience
  - Makes the computer more convenient to use
- Efficiency
  - Allows computer system resources to be used in an efficient manner
- Ability to evolve
  - Permit effective development, testing, and introduction of new system functions without interfering with service

## 2.

# Layers of Computer System

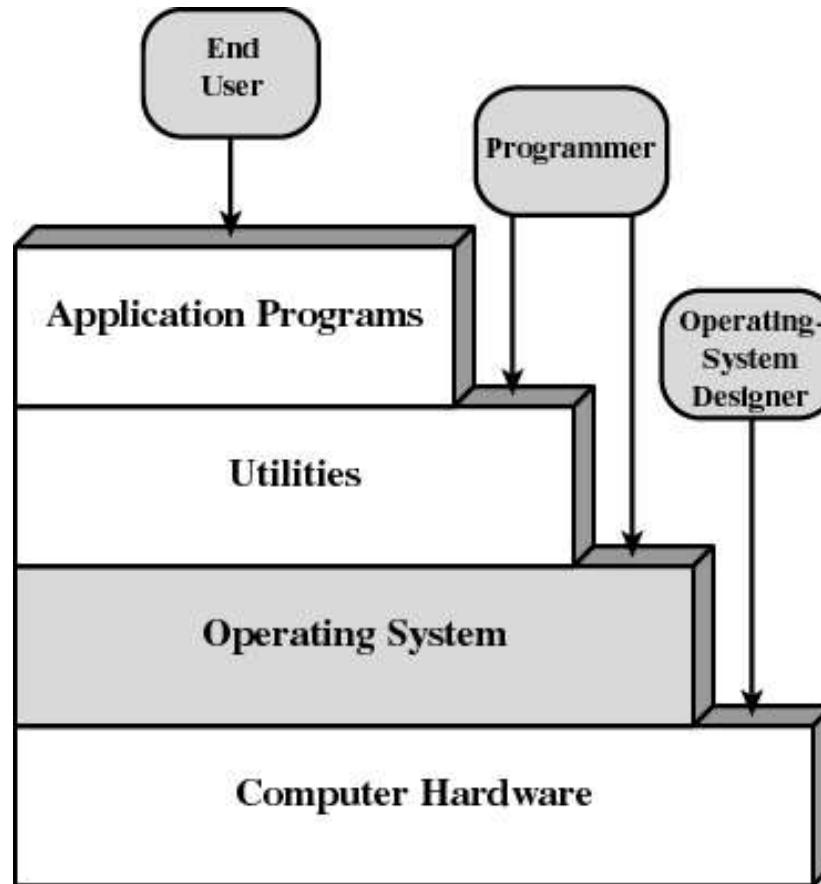


Figure 2.1 Layers and Views of a Computer System

# Kernel

- Heart (vital portion) of the operating system
- Next to Hardware
- It always resides in main memory
- Contains most-frequently used functions
- Also called the nucleus

# Services Provided by the Operating System

- Program development
  - Editors, compilers, linker, loader and debuggers
- Program execution
- Access to I/O devices
- Controlled access to files
- Access to other plug and play devices
- Security to users programs and applications

# Services Provided by the Operating System

- Error detection and response
  - internal and external hardware errors
    - memory error
    - device failure
  - software errors
    - arithmetic overflow
    - access forbidden memory locations

# Services Provided by the Operating System

- Accounting
  - collect statistics
  - monitor performance

## Computer System

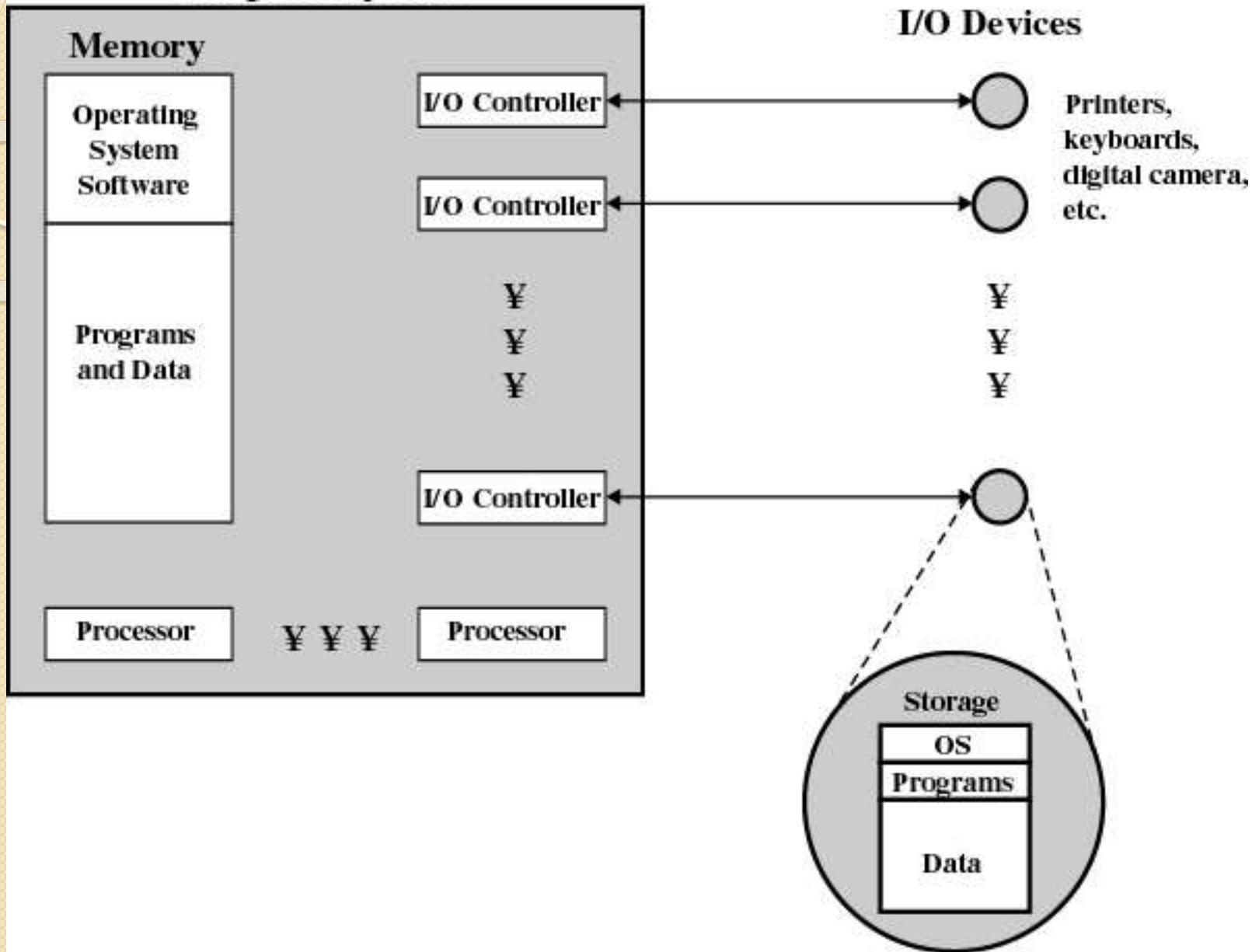


Figure 2.2 The Operating System as Resource Manager

# Evolution of Operating Systems

## 4.1 Serial Processing

- No operating system
- Machines run from a console with display lights and toggle switches, input device, and printer
- Schedule time
- Setup included loading the compiler, source program, saving compiled program, and loading and linking

# Evolution of Operating Systems

## 4.2 Simple Batch Systems

- Monitors
  - Software that controls the running programs
  - Batch jobs together
  - Program branches back to monitor when finished
  - Resident monitor is in main memory and available for execution

# Job Control Language (JCL)

- Special type of programming language
- Provides instruction to the monitor
  - what compiler to use
  - what data to use

# Hardware Features

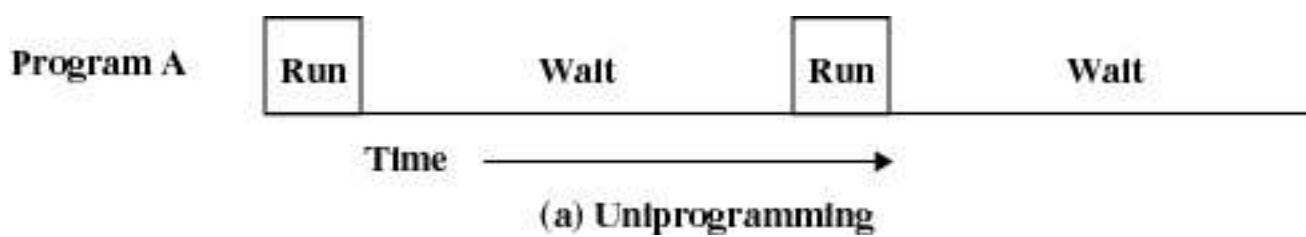
- Memory protection
  - do not allow the memory area containing the monitor to be altered
- Timer
  - prevents a job from monopolizing the system

# Hardware Features

- Memory protection
  - do not allow the memory area containing the monitor to be altered
- Timer
  - prevents a job from monopolizing the system

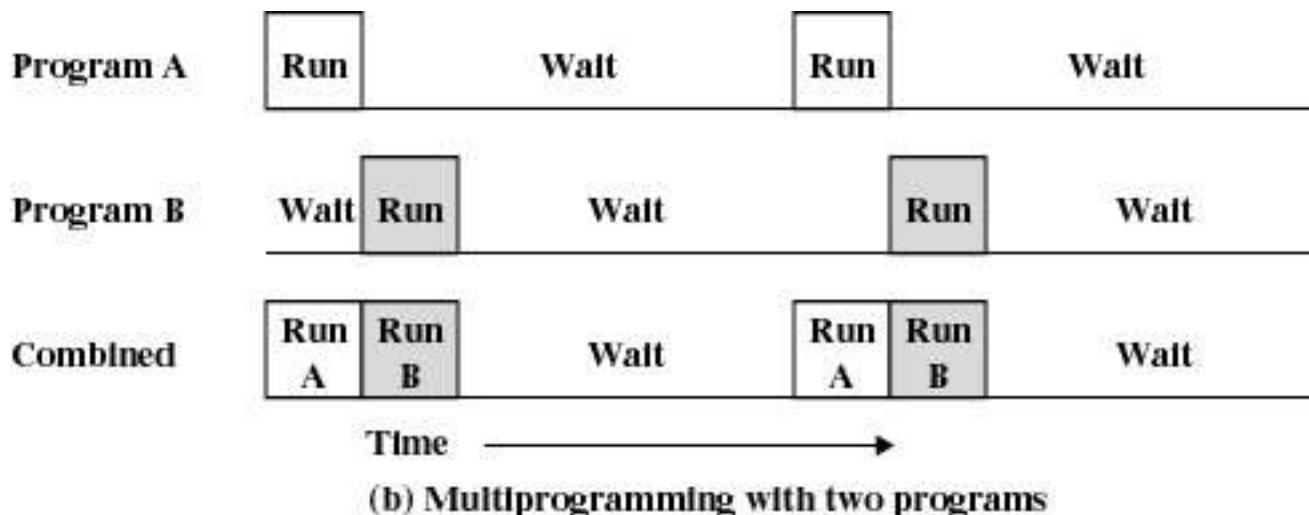
## 4.3 Uniprogramming

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

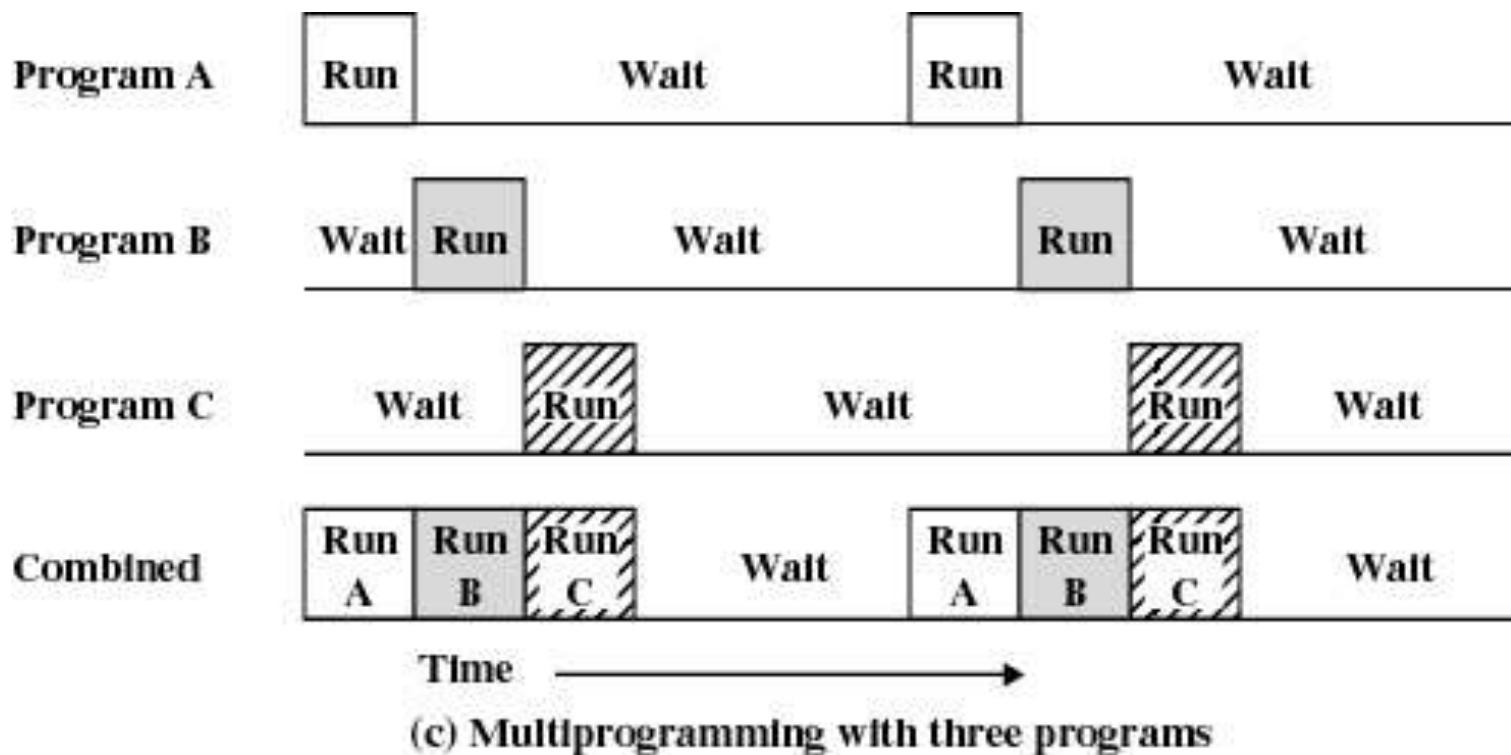


## 4.4 Multiprogramming

- When one job needs to wait for I/O, the processor can switch to the other job

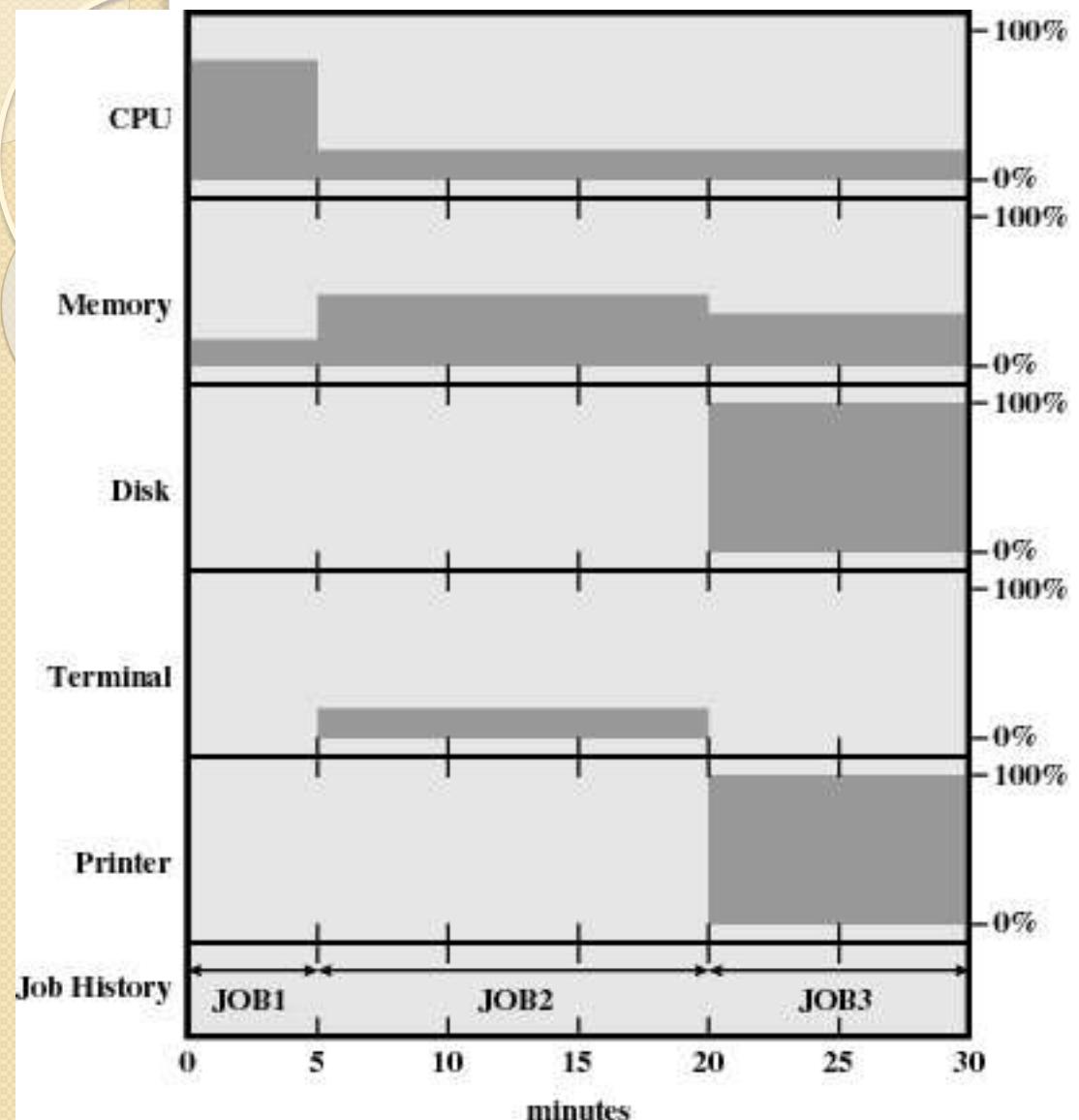


# Multiprogramming

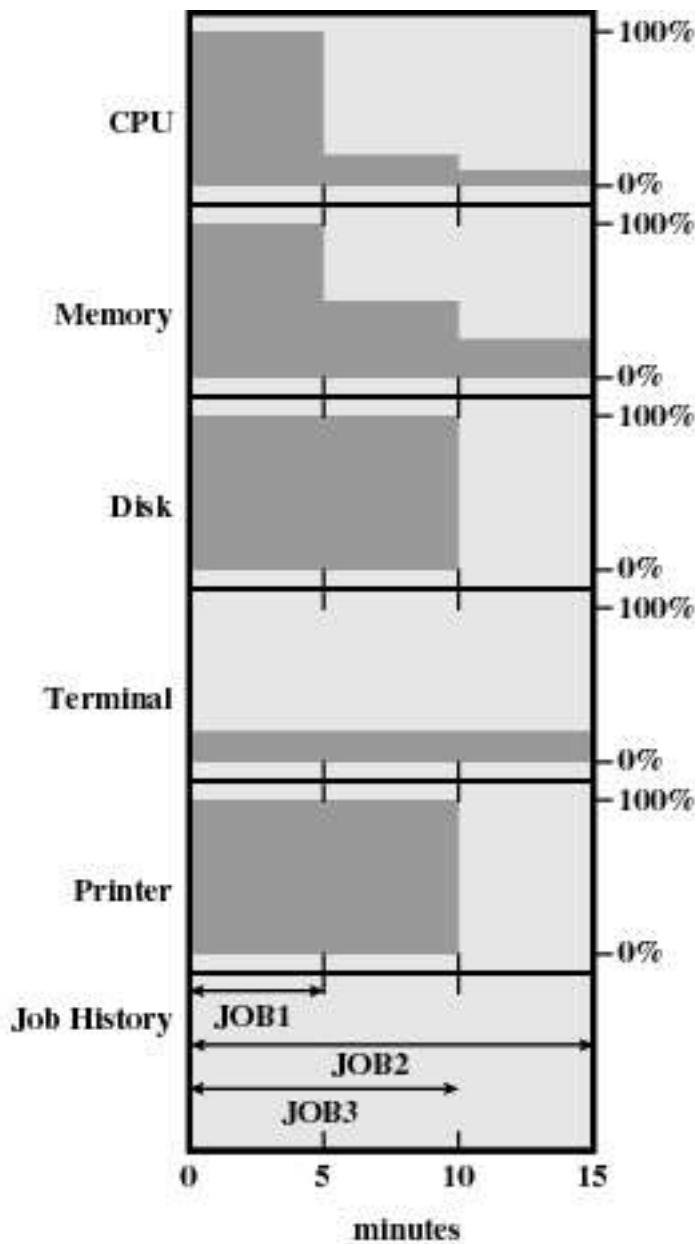


# Example

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min.	15 min.	10 min.
Memory required	50K	100 K	80 K
Need disk?	No	No	Yes
Need terminal	No	Yes	No
Need printer?	No	No	Yes



(a) Uniprogramming



(b) Multiprogramming

Figure 2.6 Utilization Histograms

# Effects of Multiprogramming

	Uniprogramming	Multiprogramming
Processor use	22%	43%
Memory use	30%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min.	15 min.
Throughput rate	6 jobs/hr	12 jobs/hr

## 4.5 Time Sharing

- Using multiprogramming to handle multiple interactive jobs
- Processor's time is shared among multiple users
- Multiple users simultaneously access the system through terminals

# Batch Multiprogramming versus Time Sharing

	<b>Batch Multiprogramming</b>	<b>Time Sharing</b>
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

## Compatible time sharing system (developed by MIT)

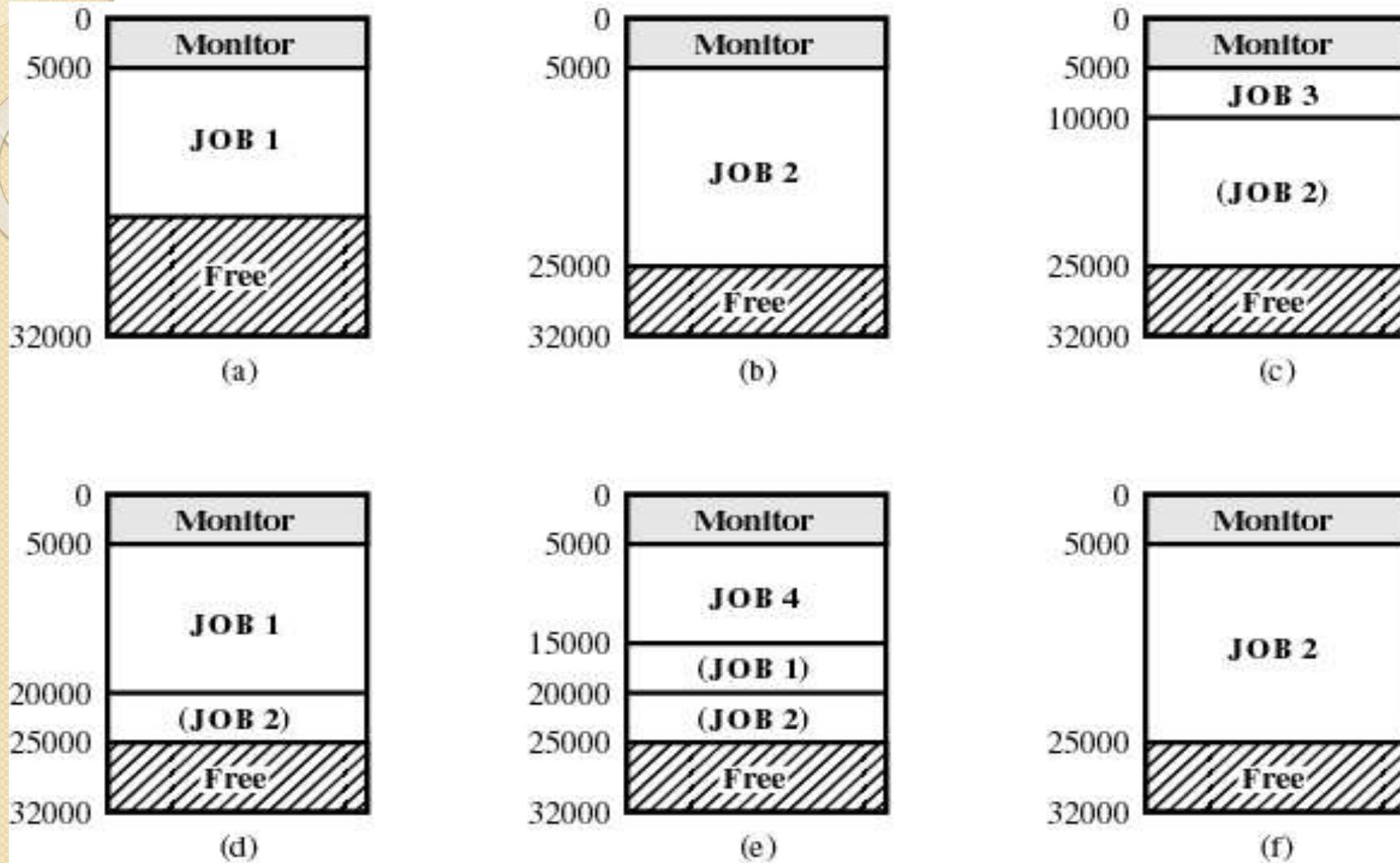


Figure 2.7 CTSS Operation

# Major Achievements

- Running processes simultaneously
- Memory Management
- Information protection and security
- Scheduling and resource management

# Processes

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

# Process

- **Consists of three components**
  - An executable program
  - Associated data needed by the program
  - Execution context of the program
    - All information the operating system needs to manage the process

# Process

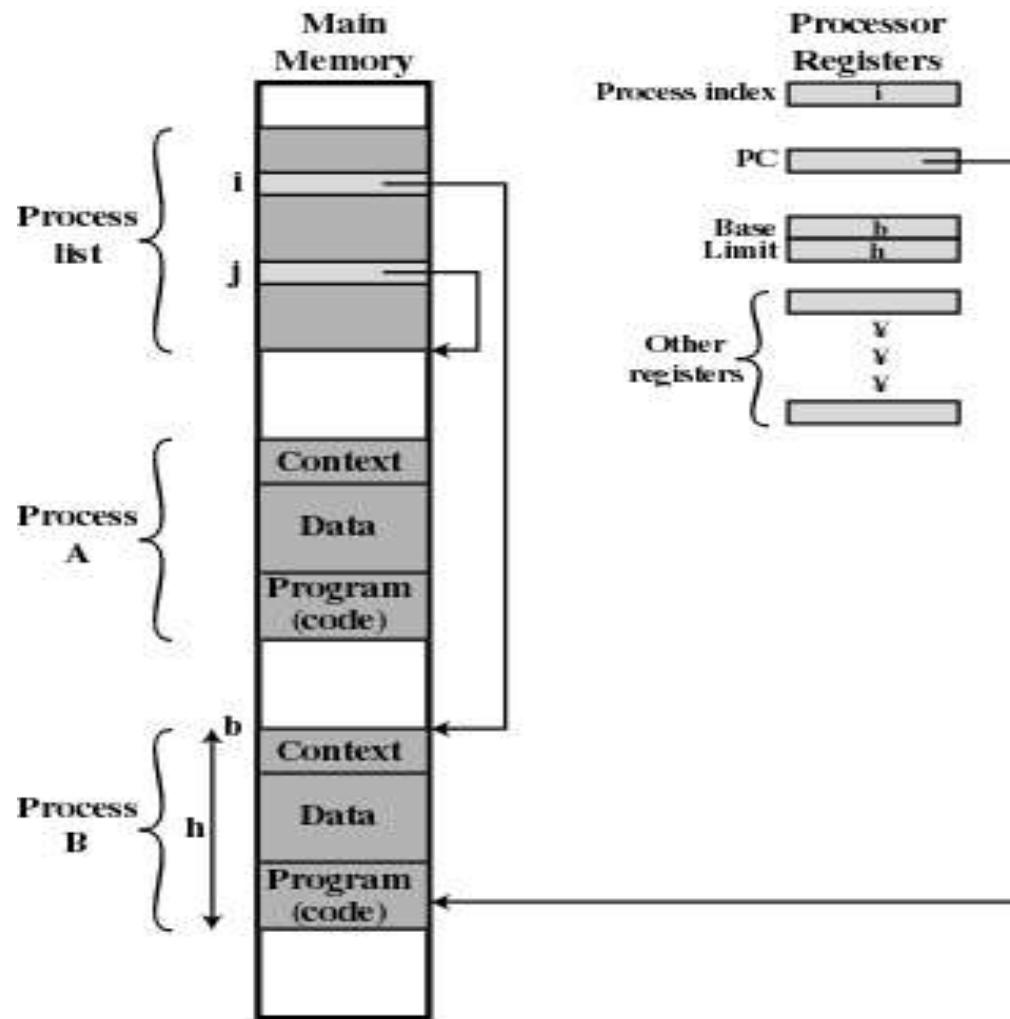


Figure 2.8 Typical Process Implementation

# Memory Management

- Process isolation
- Automatic allocation and management
- Support for modular programming
- Protection and access control
- Long-term storage

# Virtual Memory

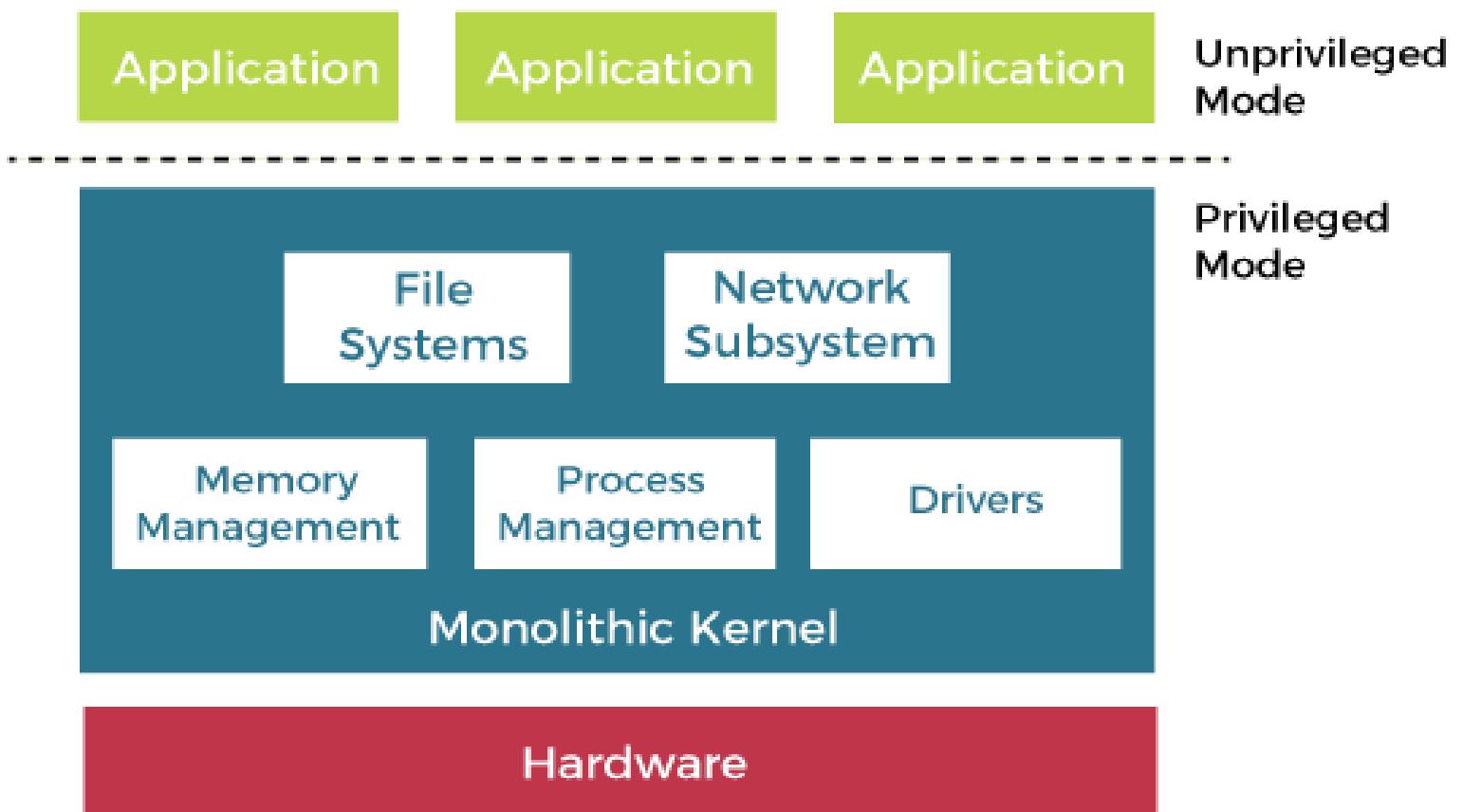
- Allows programmers to address memory from a logical point of view
- While one process is written out to secondary store and the successor process read in there in no interruption

# File System

- Implements long-term store
- Information stored in named objects called files

# Monolithic Architecture

## Monolithic Kernel System



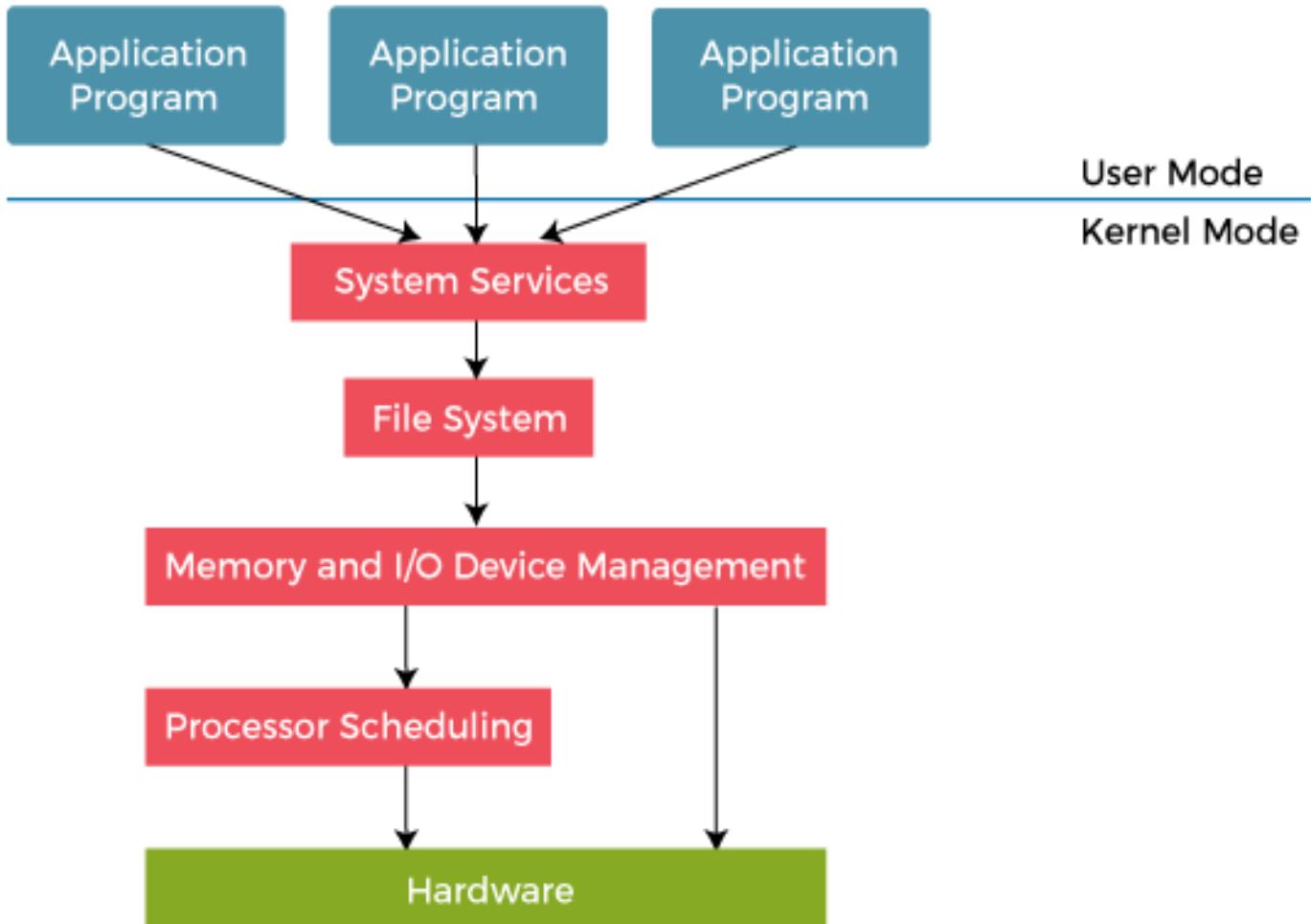
## **Advantages of Monolithic Architecture:**

- Simple and easy to implement structure.
- Faster execution due to direct access to all the services

## **Disadvantages of Monolithic Architecture:**

- The addition of new features or removal of obsolete features is very difficult.
- Security issues are always there because there is no isolation among various servers present in the kernel.

# Kernel and User Space in Layered Architecture



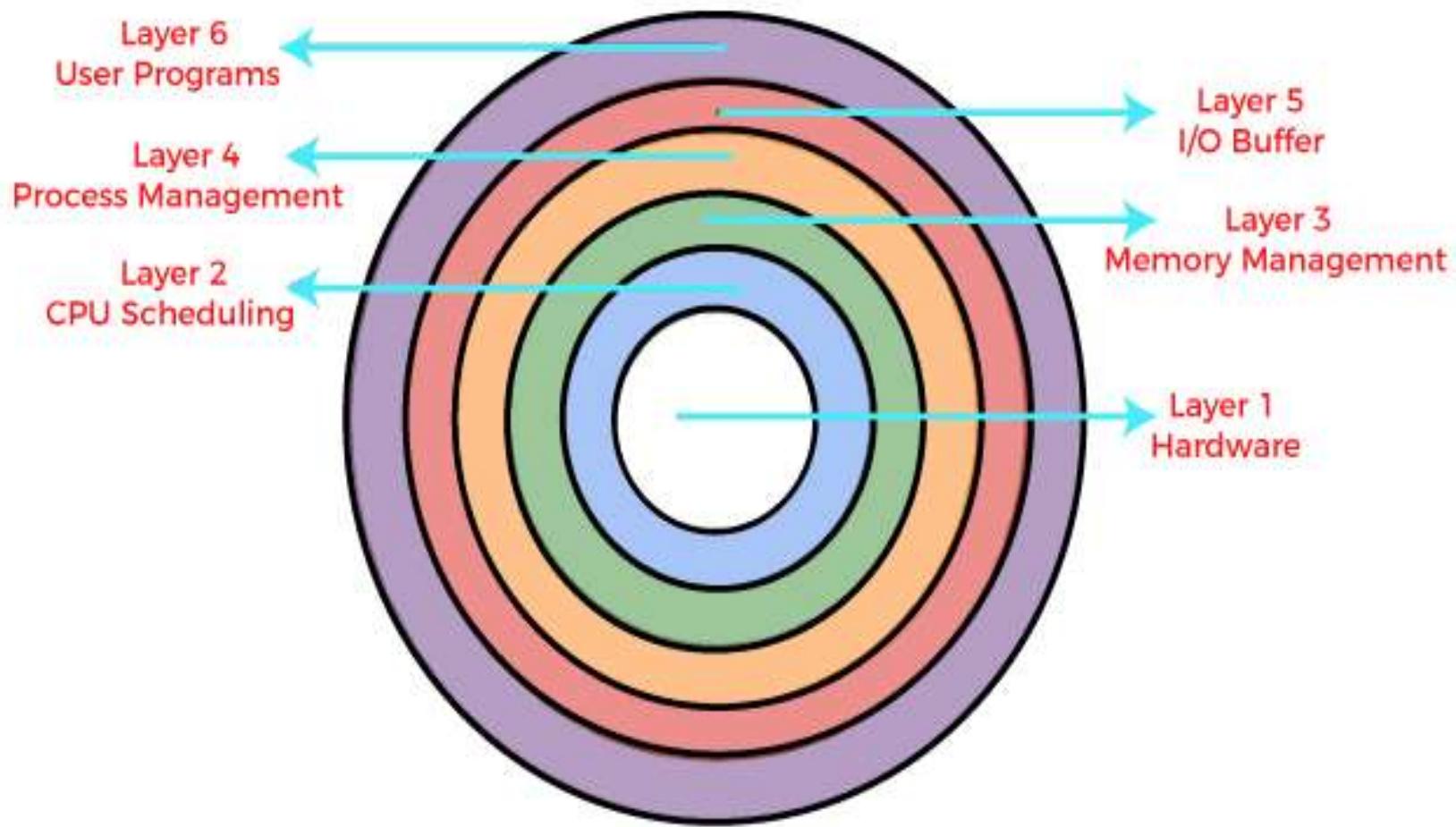
# Advantages of Layered Structure

- **Modularity:** This design promotes modularity as each layer performs only the tasks it is scheduled to perform.
- **Easy debugging:** As the layers are discrete so it is very easy to debug. Suppose an error occurs in the CPU scheduling layer. The developer can only search that particular layer to debug, unlike the Monolithic system where all the services are present.
- **Easy update:** A modification made in a particular layer will not affect the other layers.
- **No direct access to hardware:** The hardware layer is the innermost layer present in the design. So a user can use the services of hardware but cannot directly modify or access it, unlike the Simple system in which the user had direct access to the hardware.
- **Abstraction:** Every layer is concerned with its functions. So the functions and implementations of the other layers are abstract to it.

# Disadvantages of Layered Structure

- **Complex and careful implementation:** As a layer can access the services of the layers below it, so the arrangement of the layers must be done carefully. For example, the backing storage layer uses the services of the memory management layer. So it must be kept below the memory management layer. Thus with great modularity comes complex implementation.
- **Slower in execution:** If a layer wants to interact with another layer, it requests to travel through all the layers present between the two interacting layers. Thus it increases response time, unlike the Monolithic system, which is faster than this. Thus an increase in the number of layers may lead to a very inefficient design.
- **Functionality:** It is not always possible to divide the functionalities. Many times, they are interrelated and can't be separated.
- **Communication:** No direct communication between non-adjacent layers.

# Layered Architecture



# Information Protection and Security

- **Access control**
  - regulate user access to the system
- **Information flow control**
  - regulate flow of data within the system and its delivery to users
- **Certification**
  - proving that access and flow control perform according to specifications

# 7. Scheduling and Resource Management

- Fairness
  - give equal and fair access to all processes
- Differential responsiveness
  - discriminate between different classes of jobs
- Efficiency
  - maximize throughput, minimize response time, and accommodate as many uses as possible

# Major Elements of Operating System

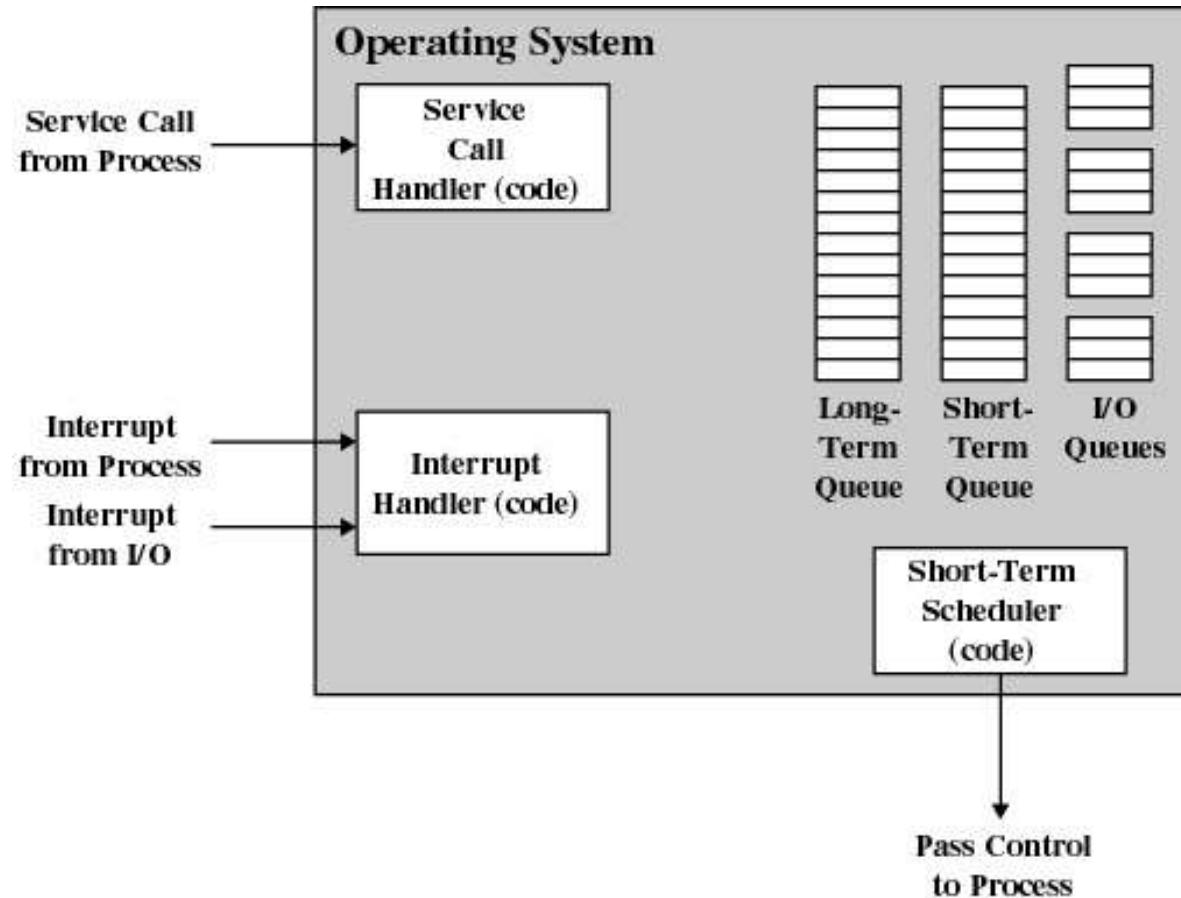


Figure 2.11 Key Elements of an Operating System for Multiprogramming

# System Structure

- View the system as a series of levels
- Each level performs a related subset of functions
- Each level relies on the next lower level to perform more primitive functions
- This decomposes a problem into a number of more manageable subproblems

# Operating System Design Hierarchy

Level name	Objects	Example Operations
I3. Shell	User programming	Statements in shell language environment
I2. User processes	User processes	Quit, kill, suspend, resume
I1. Directories	Directories	Create, destroy, attach, search, list
I0. Devices	External devices, such as printer, displays and keyboards	Open, close, read, write
I9. File system	Files	Create, destroy, open, close, read, write
I8. Communications	Pipes	Create, destroy, open, close, read, write

# Operating System Design Hierarchy

<b>Level name</b>	<b>Objects</b>	<b>Example Operations</b>
7. Virtual Memory	Segments, pages	Read, write, fetch
6. Local secondary	Blocks of data, device	Read, write, allocate, free store channels
5. Primitive processes	primitive process,	Suspend, resume, wait, signal, semaphores, ready list

# Operating System Design Hierarchy

<b>Level name</b>	<b>Objects</b>	<b>Example Operations</b>
4 Interrupts	Interrupt-handling	Invoke, mask, unmask, Programs retry
3 Procedures	Procedure, call stack,	Mark stack, call, return
2 Instruction Set subtract	Evaluation stack, micro- program interpreter, branch scalar and array data	Load, store, add,
1 Electronic circuit	Registers, gates, buses,	Clear, transfer, activate, etc.

# Characteristics of Modern Operating Systems

- Microkernel architecture
  - assigns only a few essential functions to the kernel
    - address space
    - Inter-process communication (IPC)
    - basic scheduling

# Characteristics of Modern Operating Systems

- Multithreading
  - process is divided into threads that can run simultaneously
  - Thread
  - dispatchable unit of work
  - executes sequentially and is interruptible
  - Process is a collection of one or more threads

# Characteristics of Modern Operating Systems

- Multithreading
  - process is divided into threads that can run simultaneously
  - Thread
  - dispatchable unit of work
  - executes sequentially and is interruptible
  - Process is a collection of one or more threads

# Characteristics of Modern Operating Systems

- **Distributed operating systems**
  - provides the illusion of a single main memory and single secondary memory space
  - used for distributed file system

# Characteristics of Modern Operating Systems

- Object-oriented design
  - used for adding modular extensions to a small kernel
  - enables programmers to customize an operating system without disrupting system integrity



Continued.....

Unit -2

Threads

and

Processes

# Objectives

- To introduce the notion of a thread — a smallest dispatchable unit of CPU utilization that forms the basis of multithreaded computer systems
- To examine issues related to multithreaded programming

# I.

## Process

- **Resource ownership** - process is allocated a virtual address space to hold the process image
- **Scheduling/execution** - follows an execution path that may be interleaved with other processes
- These two characteristics are treated independently by the operating system

# Process

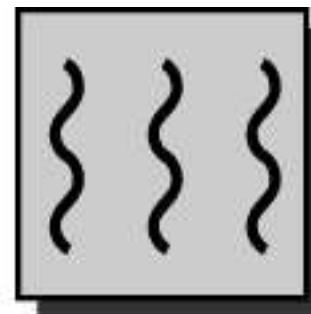
- Dispatching is referred to as a thread
- Resource of ownership is referred to as a process or task
- Protected access to processors, other processes, files, and I/O resources

# Multithreading

- Operating system supports multiple threads of execution within a single process
- MS-DOS supports a single thread
- UNIX supports multiple user processes but only supports one thread per process
- Windows 2000, Solaris, Linux, Mach, and OS/2 support multiple threads



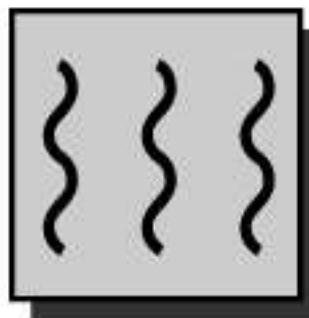
one process  
one thread



one process  
multiple threads



multiple processes  
one thread per process

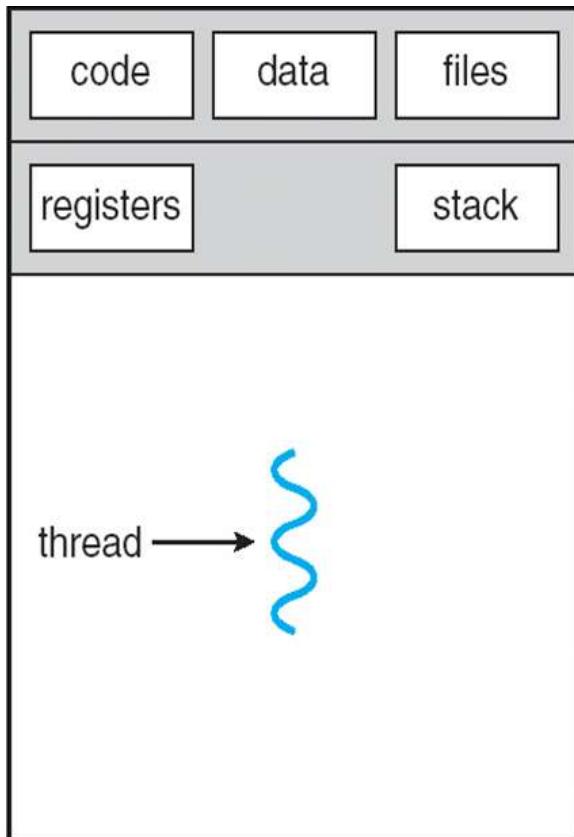


multiple processes  
multiple threads per process

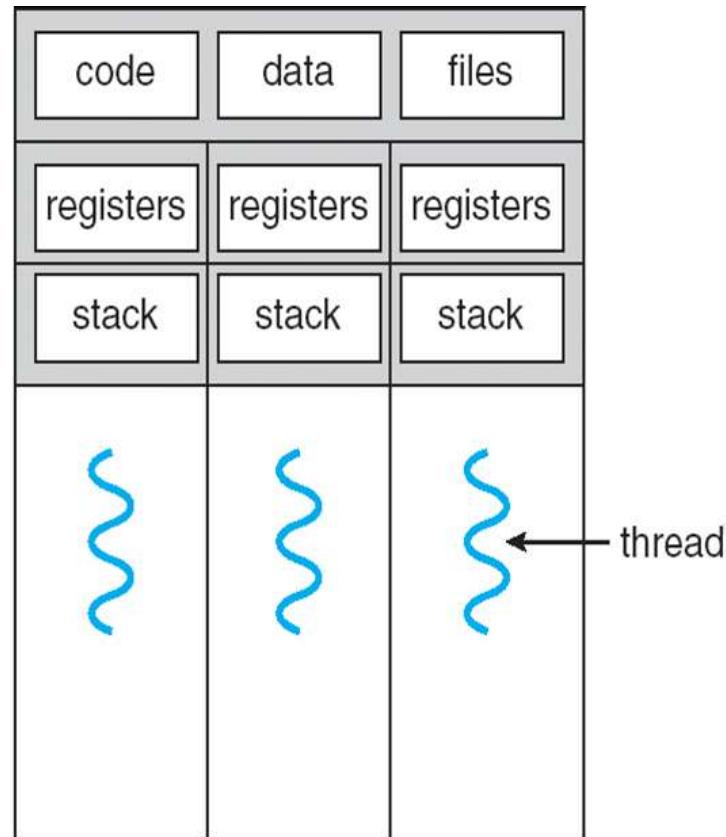
{ = instruction trace

Figure 4.1 Threads and Processes [ANDE97]

# Single and Multithreaded Processes



single-threaded process



multithreaded process

### 3.

## Thread

- An execution state (running, ready, etc.)
- Saved thread context when not running
- Has an execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its process
  - all threads of a process share this

# Threads

- Suspending a process involves suspending all threads of the process since all threads share the same address space
- Termination of a process, terminates all threads within the process

# Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel

# Uses of Threads in a Single-User Multiprocessing System

- Foreground to background work
- Asynchronous processing
- Speed execution
- Modular program structure

# Thread States

- States associated with a change in thread state
  - Spawn
    - Issue another thread
  - Block
  - Unblock
  - Finish
    - Deallocate register context and stacks

# 5.

# Levels of Threads

## 5.1 User-Level Threads

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

## 5.2 Kernel-Level Threads

- W2K, Linux, and OS/2 are examples of this approach
- Kernel maintains context information for the process and the threads
- Scheduling is done on a thread basis

## 5.3 Combined Approaches

- Example is Solaris
- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space

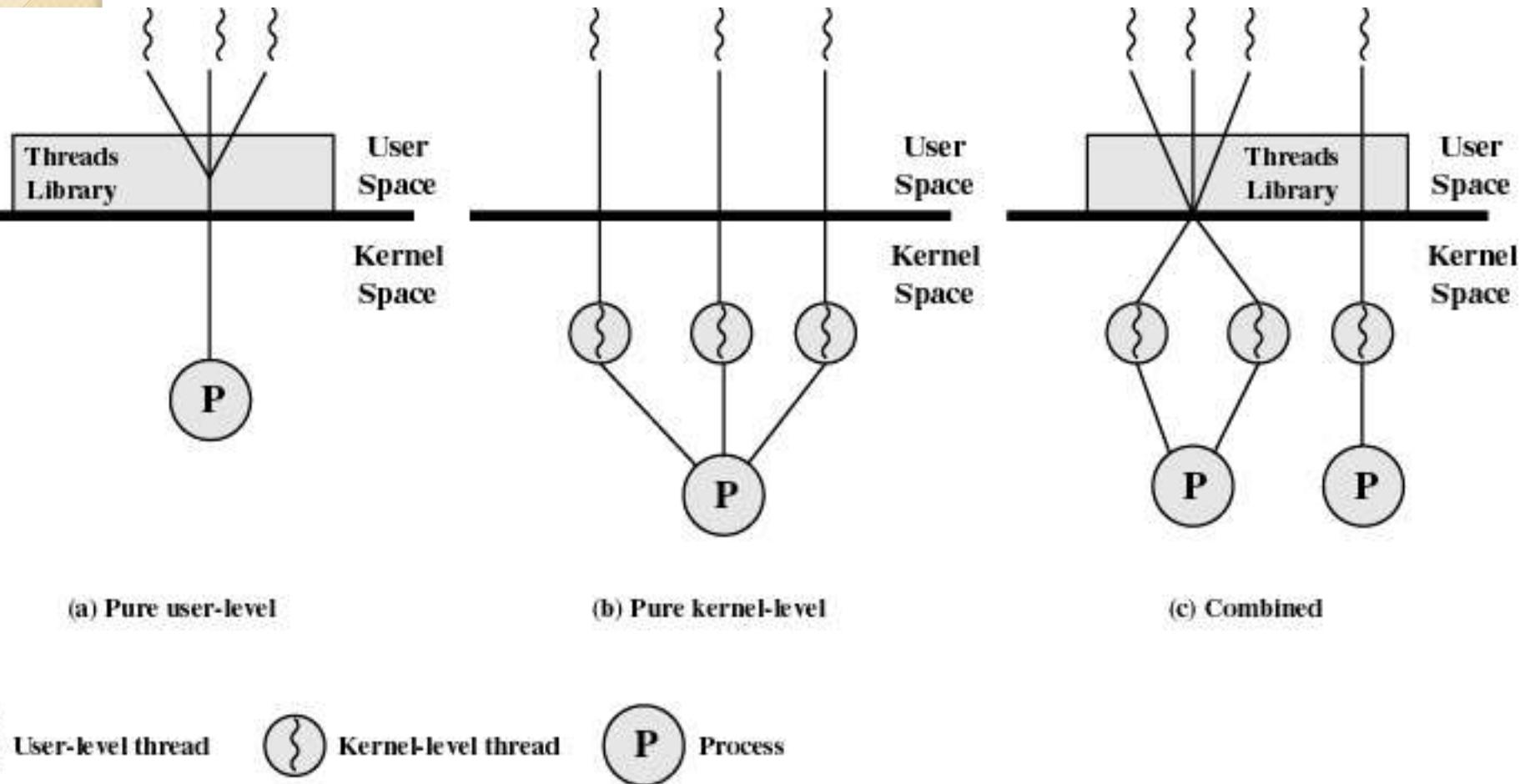


Figure 4.6 User-Level and Kernel-Level Threads

# Relationship Between Threads and Processes

<b>Threads:Process</b>	<b>Description</b>	<b>Example Systems</b>
I:I	<b>Each thread of execution is a unique process with its own address space and resources.</b>	<b>Traditional UNIX implementations</b>
M:I	<b>A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.</b>	<b>Windows NT, Solaris, OS/2, OS/390, MACH</b>

# Relationship Between Threads and Processes

<b>Threads:Process</b>	<b>Description</b>	<b>Example Systems</b>
<b>I:M</b>	<b>A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.</b>	Ra (Clouds), Emerald
<b>M:M</b>	<b>Combines attributes of M:I and I:M cases</b>	TRIX



Unit 2

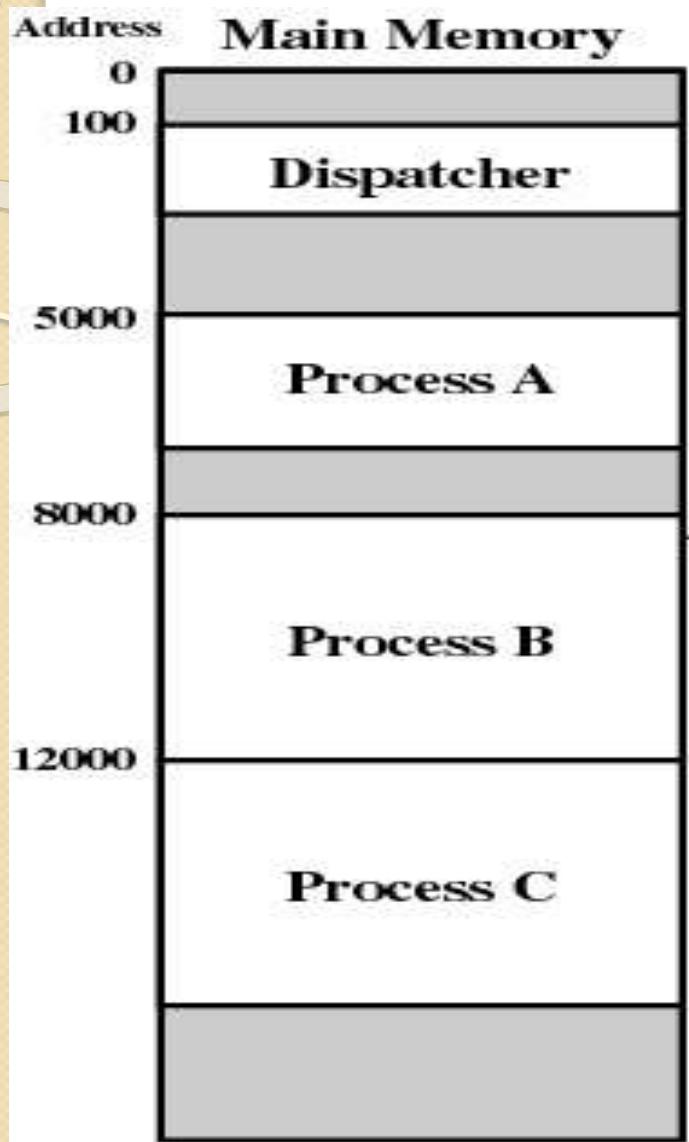
Process  
and  
Process Scheduling

# Major Requirements of an Operating System

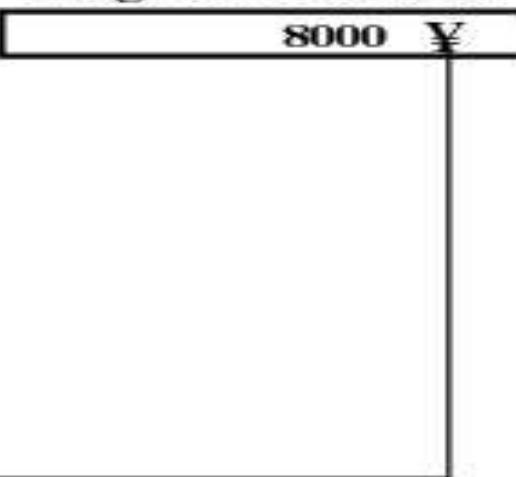
- Interleave the execution of several processes to maximize processor utilization while providing reasonable response time
- Allocate resources to processes
- Support inter-process communication and
- Creation of user processes

# Process

- Also called a task
- Execution of an individual program
- Can be traced
  - list the sequence of instructions that execute



**Program Counter**



**Figure 3.1 Snapshot of Example Execution (Figure 3.3) at Instruction Cycle 13**

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A

(b) Trace of Process B

(c) Trace of Process C

5000 = Starting address of program of Process A

8000 = Starting address of program of Process B

12000 = Starting address of program of Process C

**Figure 3.2 Traces of Processes of Figure 3.1**

1	5000	27	12004
2	5001	28	12005
3	5002		-----Time out
4	5003	29	100
5	5004	30	101
6	5005	31	102
		32	103
		33	104
		34	105
7	100	35	5006
8	101	36	5007
9	102	37	5008
10	103	38	5009
11	104	39	5010
12	105	40	5011
13	8000		-----Time out
14	8001	41	100
15	8002	42	101
16	8003	43	102
		44	103
		45	104
		46	105
17	100	47	12006
18	101	48	12007
19	102	49	12008
20	103	50	12009
21	104	51	12010
22	105	52	12011
23	12000		-----Time out
24	12001		
25	12002		
26	12003		

100 = Starting address of dispatcher program

shaded areas indicate execution of dispatcher process;  
 first and third columns count instruction cycles;  
 second and fourth columns show address of instruction being executed

Figure 3.3 Combined Trace of Processes of Figure 3.1

### 3.

# Process Creation (how?)

- Submission of a batch job
- User logs on
- Created to provide a service such as printing
- Process creates another process
- Batch job issues *Halt* instruction
- User logs off
- Quit an application
- Error and fault conditions

# Reasons for Process Termination

- Normal completion
- Time limit exceeded
- Memory unavailable
- Bounds violation
- Protection error
  - example write to read-only file
- Arithmetic error
- Time overrun
  - process waited longer than a specified maximum for an event

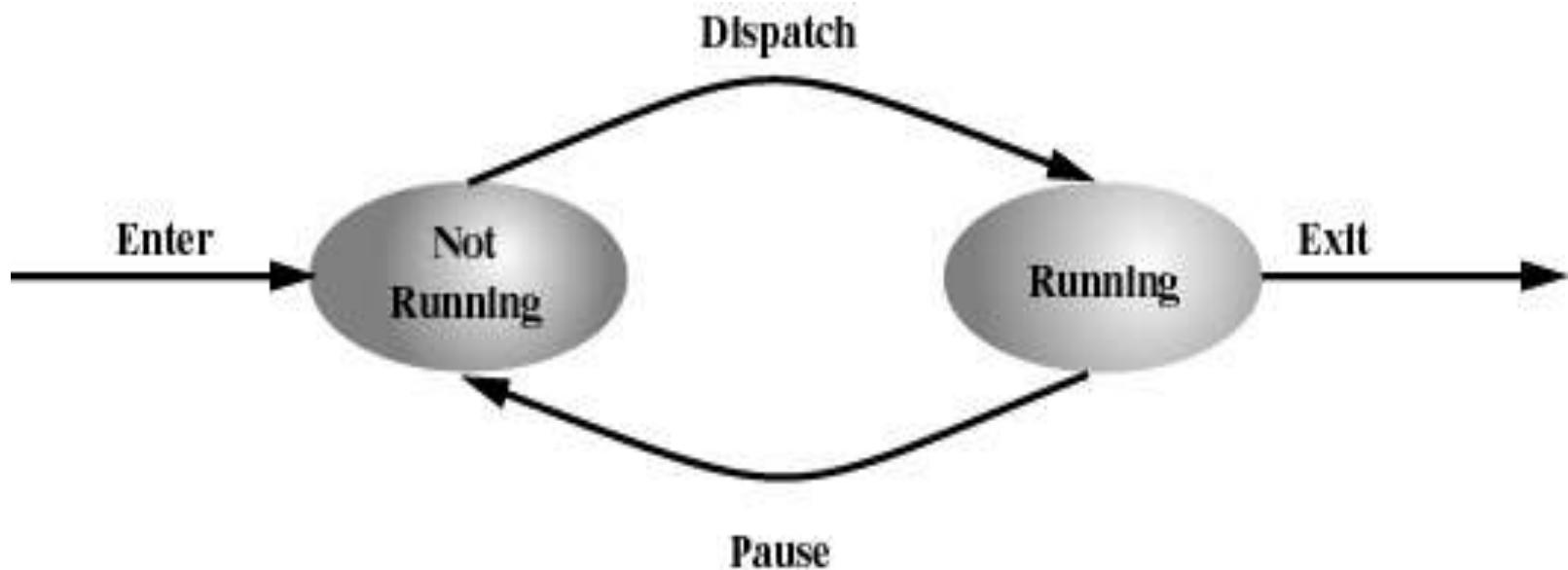
# Reasons for Process Termination

- I/O failure
- Invalid instruction
  - happens when try to execute data
- Privileged instruction
- Data misuse
- Operating system intervention
  - such as when deadlock occurs
- Parent terminates so child processes terminate
- Parent request

## 4.

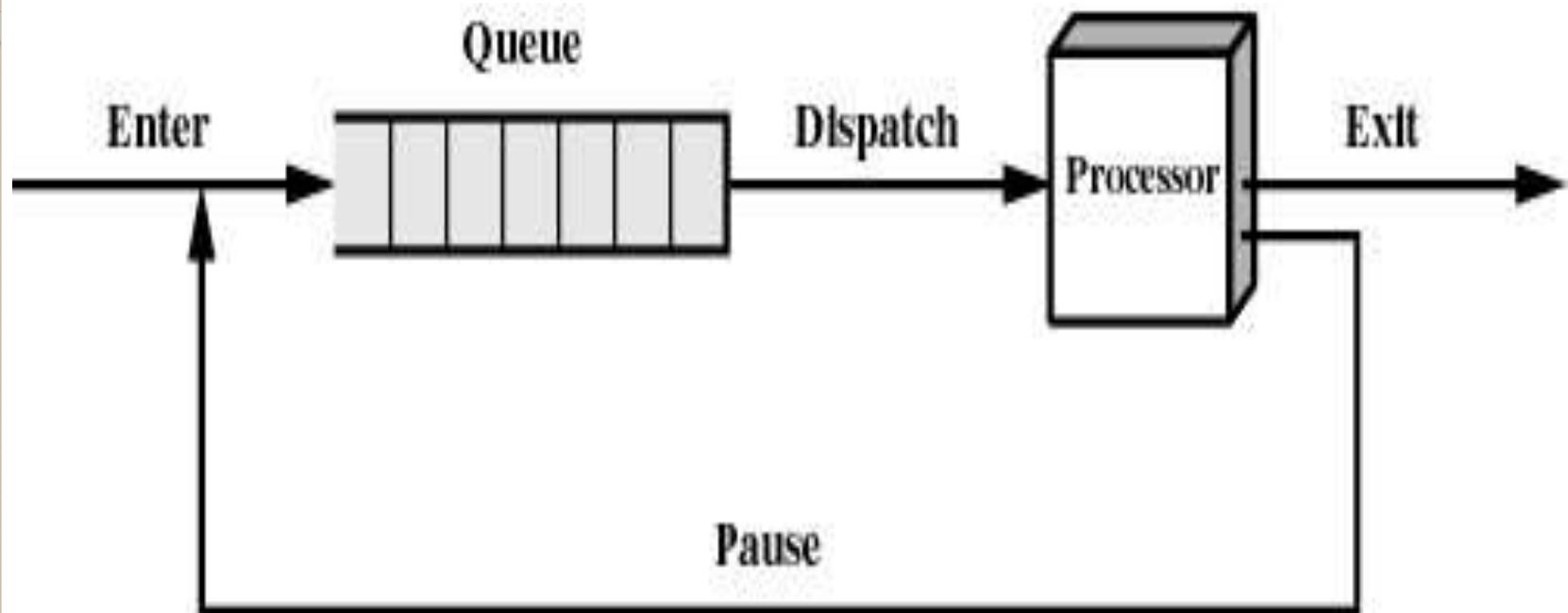
## Two-State Process Model

- Process may be in one of two states
  - Running
  - Not-running



(a) State transition diagram

# Not-Running Process in a Queue



(b) Queuing diagram

# Processes

- Not-running
  - ready to execute
- Blocked
  - waiting for I/O
- Dispatcher cannot just select the process that has been in the queue the longest because it may be blocked

# 5.

## A Five-State Model

- Running
- Ready
- Blocked
- New
- Exit

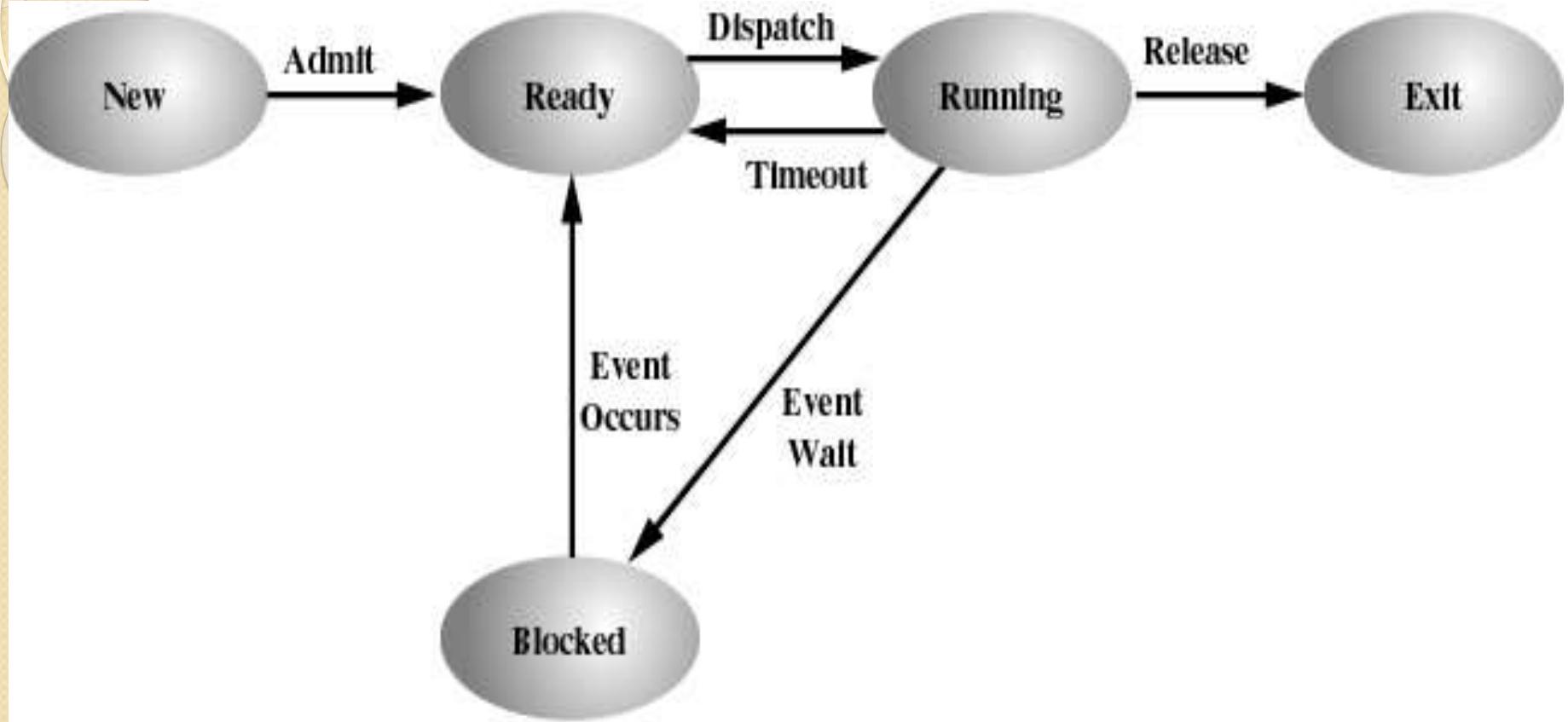
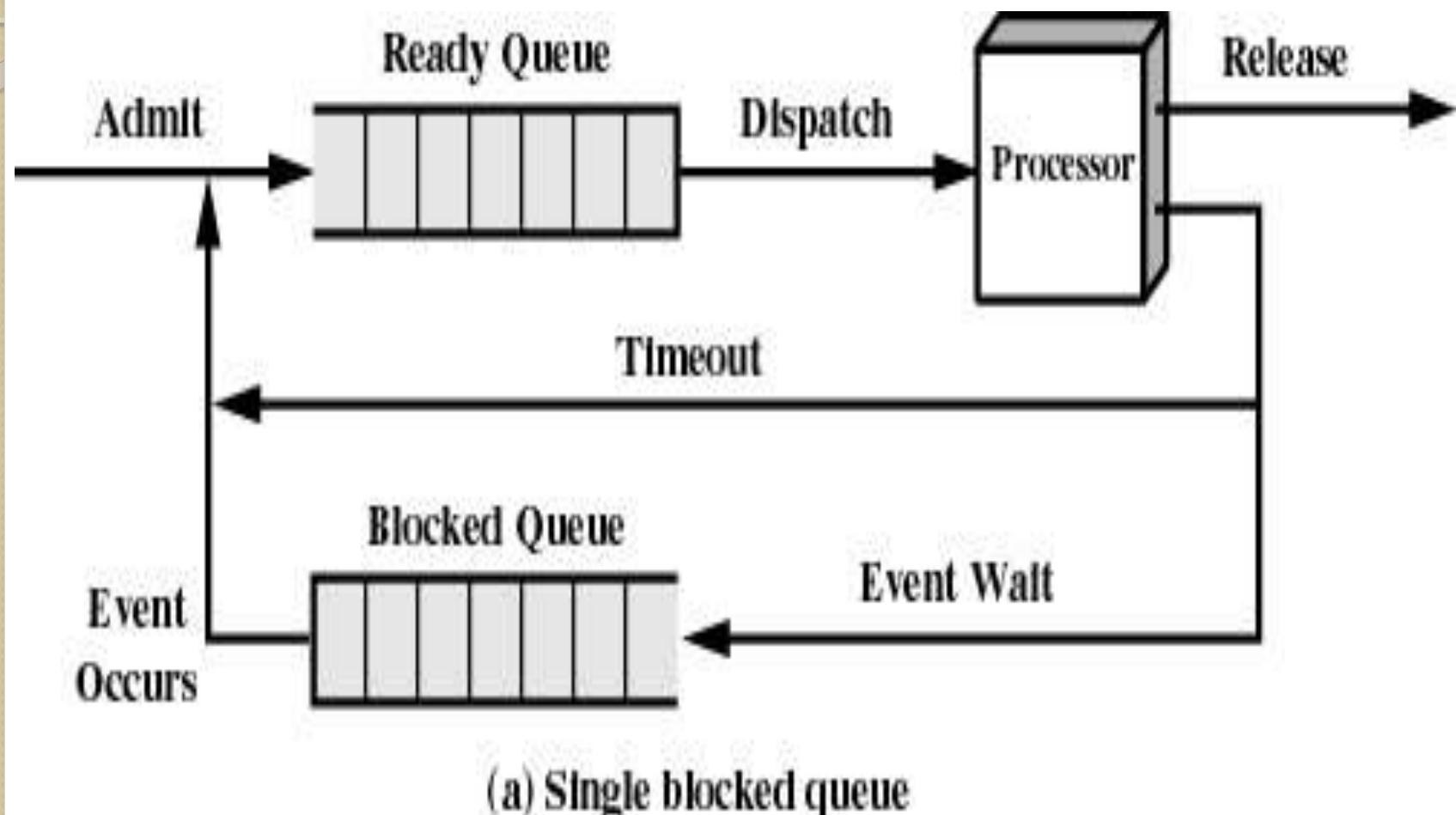
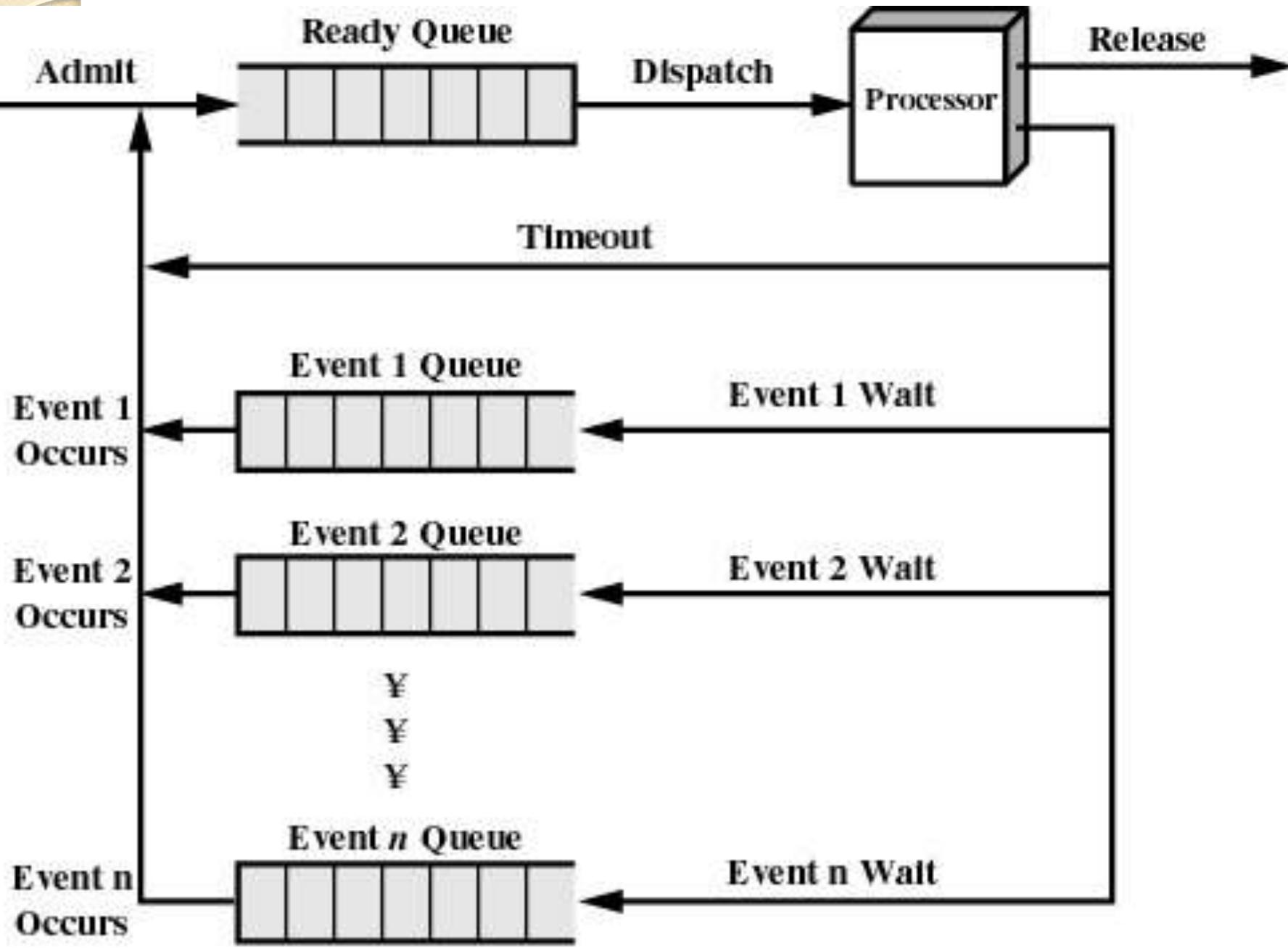


Figure 3.5 Five-State Process Model

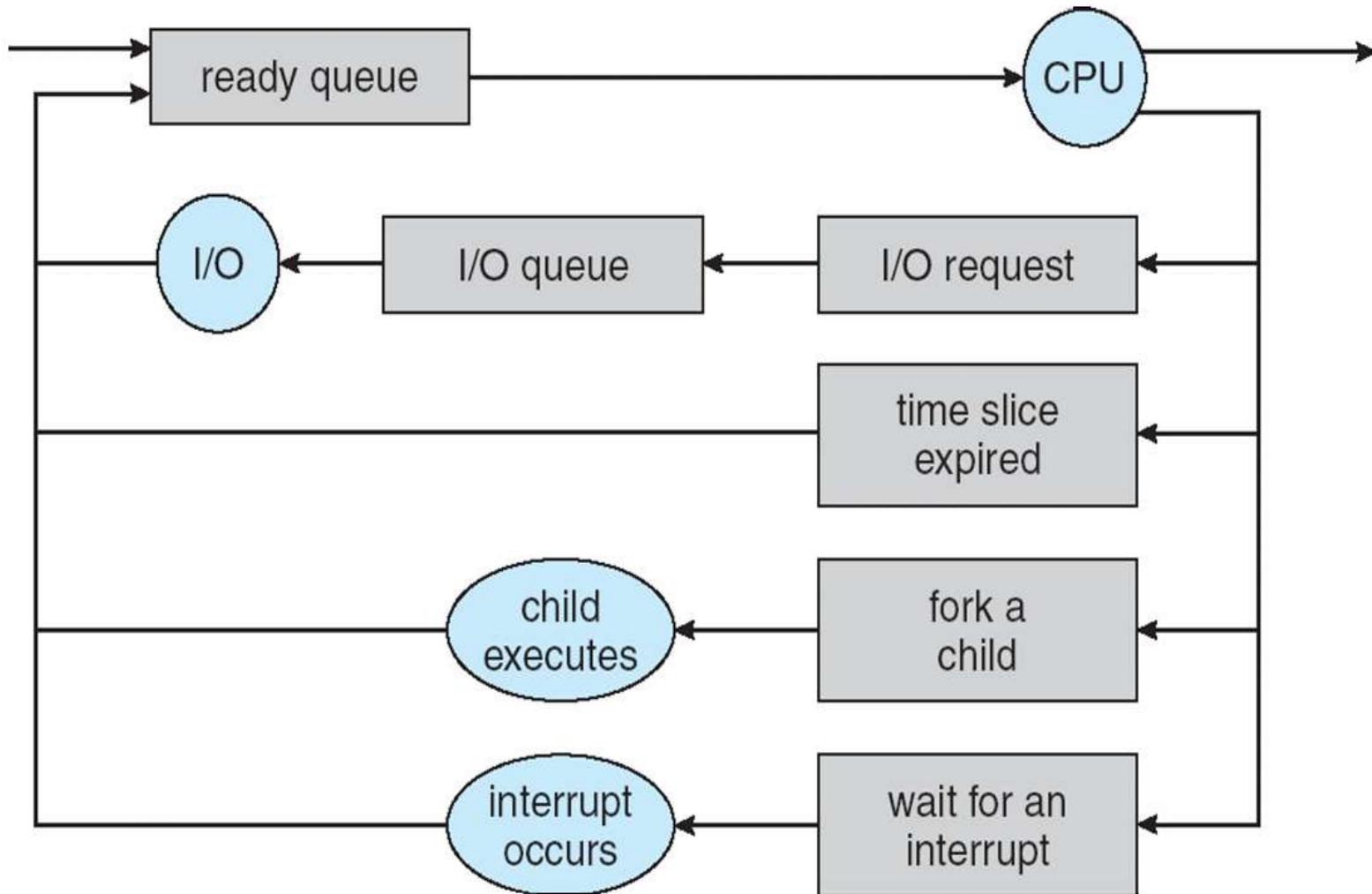
# Using Two Queues





(b) Multiple blocked queues

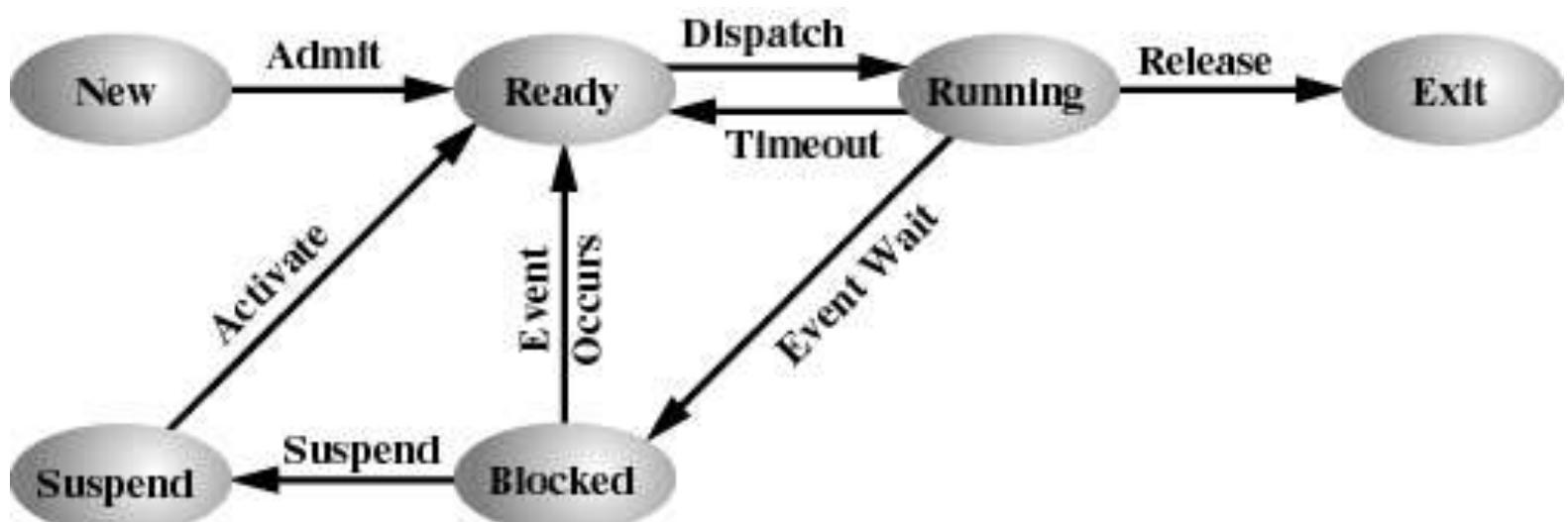
# Example



# Suspended Processes

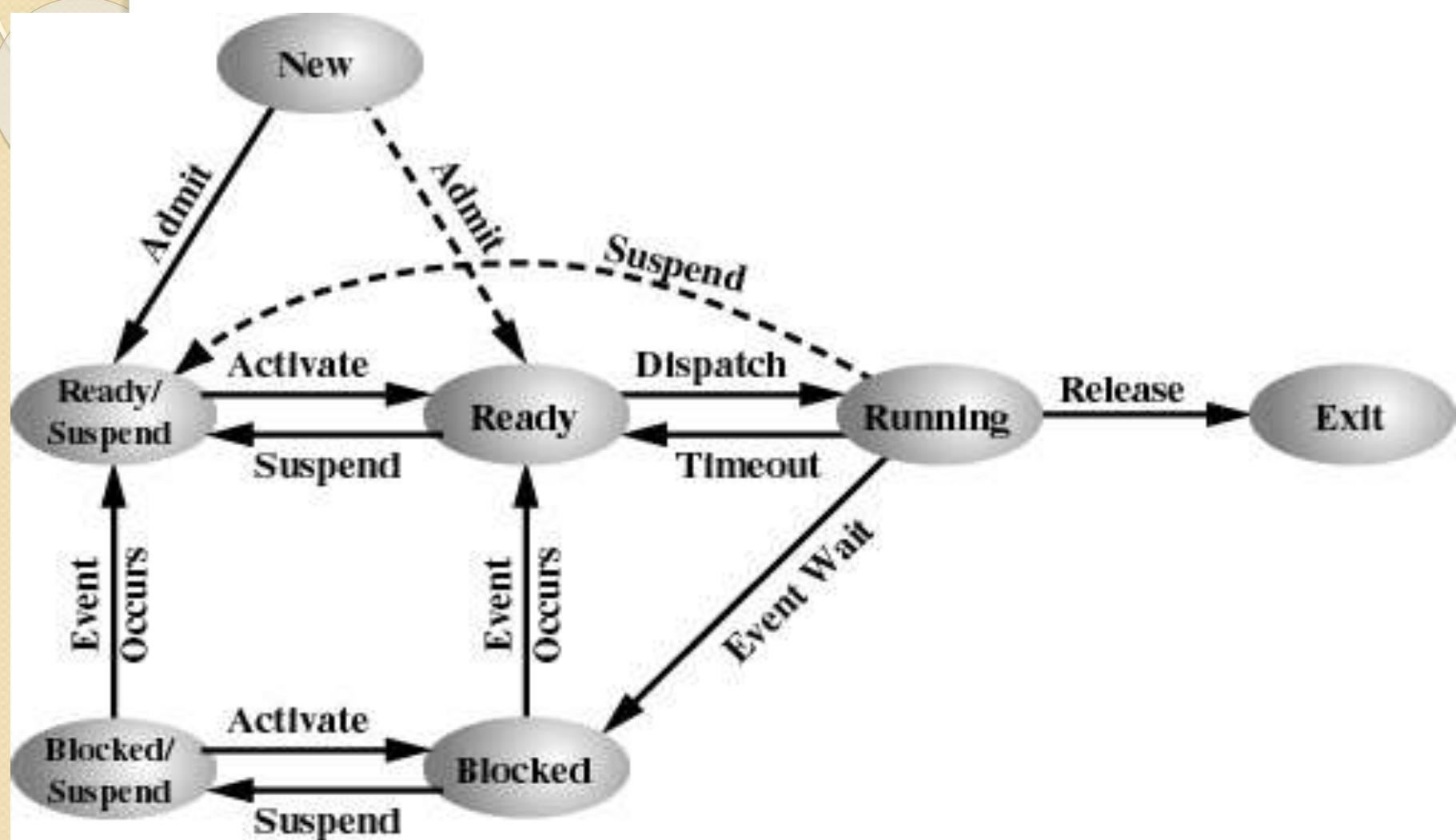
- Processor is faster than I/O so all processes could be waiting for I/O
- Swap these processes to disk to free up more memory
- Blocked state becomes suspend state when swapped to disk
- Two new states
  - Blocked, suspend
  - Ready, suspend

# One Suspend State



(a) With One Suspend State

# Two Suspend States



(b) With Two Suspend States

# Operating System Control Structures

- Information about the current status of each process and resource
- Tables are constructed for each entity the operating system manages

# Memory Tables

- Allocation of main memory to processes
- Allocation of secondary memory to processes
- Protection attributes for access to shared memory regions
- Information needed to manage virtual memory

7.b

## I/O Tables

- I/O device is available or assigned
- Status of I/O operation
- Location in main memory being used as the source or destination of the I/O transfer

## File Tables

- Existence of files
- Location on secondary memory
- Current Status
- Attributes
- Sometimes this information is maintained by a file-management system

## Process Table

- Where process is located
- Attributes necessary for its management
  - Process ID
  - Process state
  - Location in memory

# Process Location

- Process includes set of programs to be executed
  - Data locations for local and global variables
  - Any defined constants
  - Stack
- Process control block
  - Collection of attributes
- Process image
  - Collection of program, data, stack, and attributes

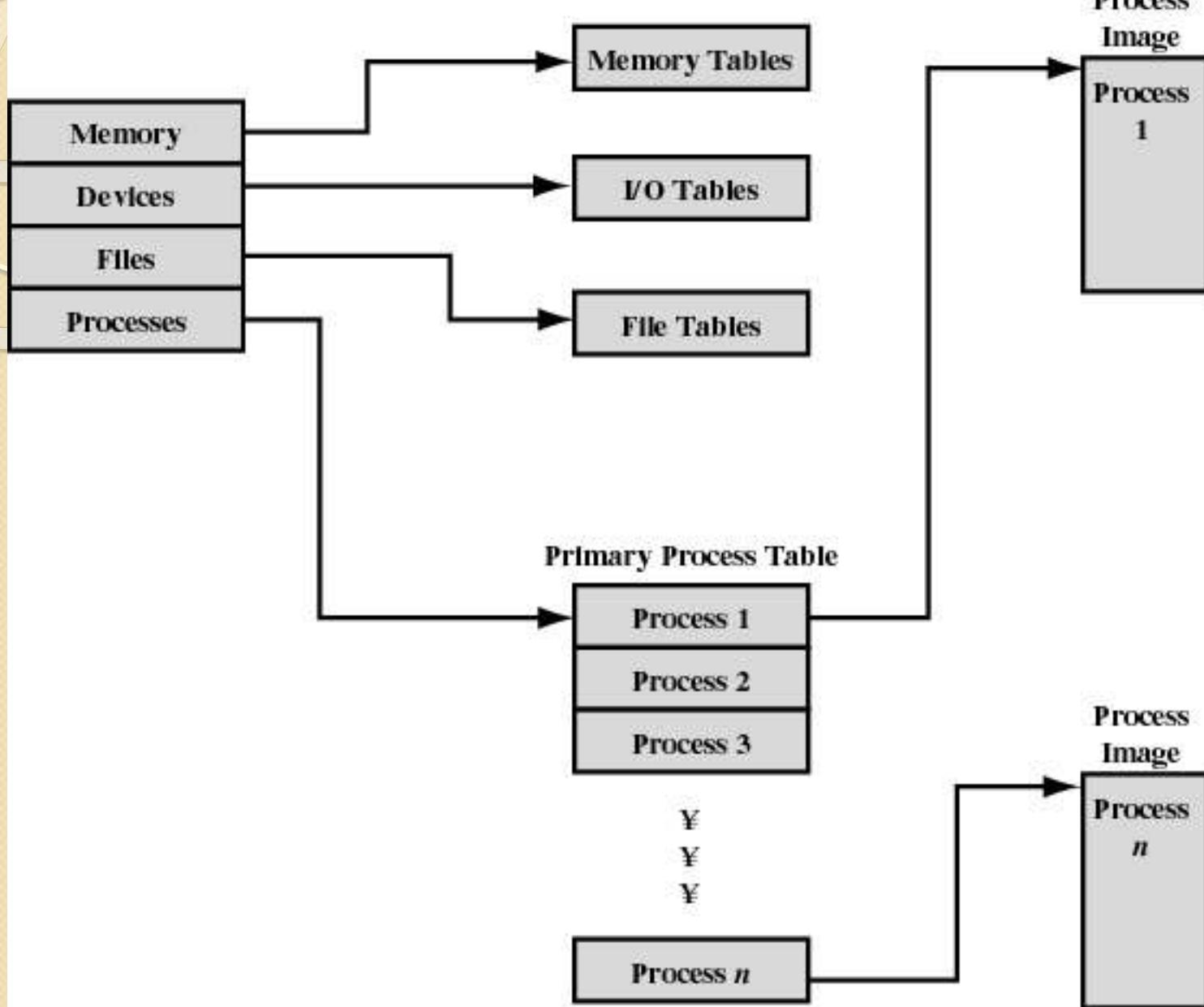


Figure 3.10 General Structure of Operating System Control Tables

# Process Control Block

## I. Process identification (Identifiers)

- Numeric identifiers that may be stored with the process control block include
  - Identifier of this process
  - Identifier of the process that created this process (parent process)
  - User identifier

# **Process Control Block**

## **2. Processor State Information**

### **2.1 User-Visible Registers**

- A user-visible register is one that may be referenced by means of the machine language that the processor executes.
- Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.

# Process Control Block

## Processor State Information (contd..)

### 2.2 Control and Status Registers

These are a variety of processor registers that are employed to control the operation of the processor.

These include

- **Program counter:** Contains the address of the next instruction to be fetched
- **Condition codes:** Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
- **Status information:** Includes interrupt enabled/disabled flags, execution mode

# Process Control Block

## Processor State Information (contd.....)

- **Stack Pointers**
  - Each process has one or more last-in-first-out (LIFO) system stacks associated with it.
  - A stack is used to store parameters and calling addresses for procedure and system calls.
  - The stack pointer points to the top of the stack.

# Process Control Block

## 3. Process Control Information

### 3.I Scheduling and State Information

This is information that is needed by the operating system to perform its scheduling function.

**Typical items of information:**

- **Process state:** defines the readiness of the process to be scheduled for execution (e.g., running, ready, blocked, halted, exit).
- **Priority:** One or more fields may be used to describe the scheduling priority of the process.

# Process Control Block

## Scheduling and State Information (contd...)

***Scheduling-related information:*** This will depend on the scheduling algorithm used.

Examples:- the amount of time that the process has been waiting and

the amount of time that the process executed the last time it was running.

***Event:*** Identity of event the process is awaiting before it can be resumed

# Process Control Block

## Process Control Information (contd...)

- **Data Structuring**

- A process may be linked to other process in a queue, ring, or some other structure.
- For example, all processes in a waiting state for a particular priority level may be linked in a queue.
- A process may exhibit a parent-child (creator-created) relationship with another process.
- The process control block may contain pointers to other processes to support these structures.

# **Process Control Block**

## **Process Control Information (contd.....)**

- **Inter-process Communication**
  - Various flags, signals, and messages may be associated with communication between two independent processes.
  - Some or all of this information may be maintained in the process control block.
- **Process Privileges**
  - Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed.
  - In addition, privileges may apply to the use of system utilities and services.

# **Process Control Block**

## **Process Control Information (contd....)**

- **Memory Management**
  - This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.
- **Resource Ownership and Utilization**
  - Resources controlled by the process may be indicated, such as opened files.
  - A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

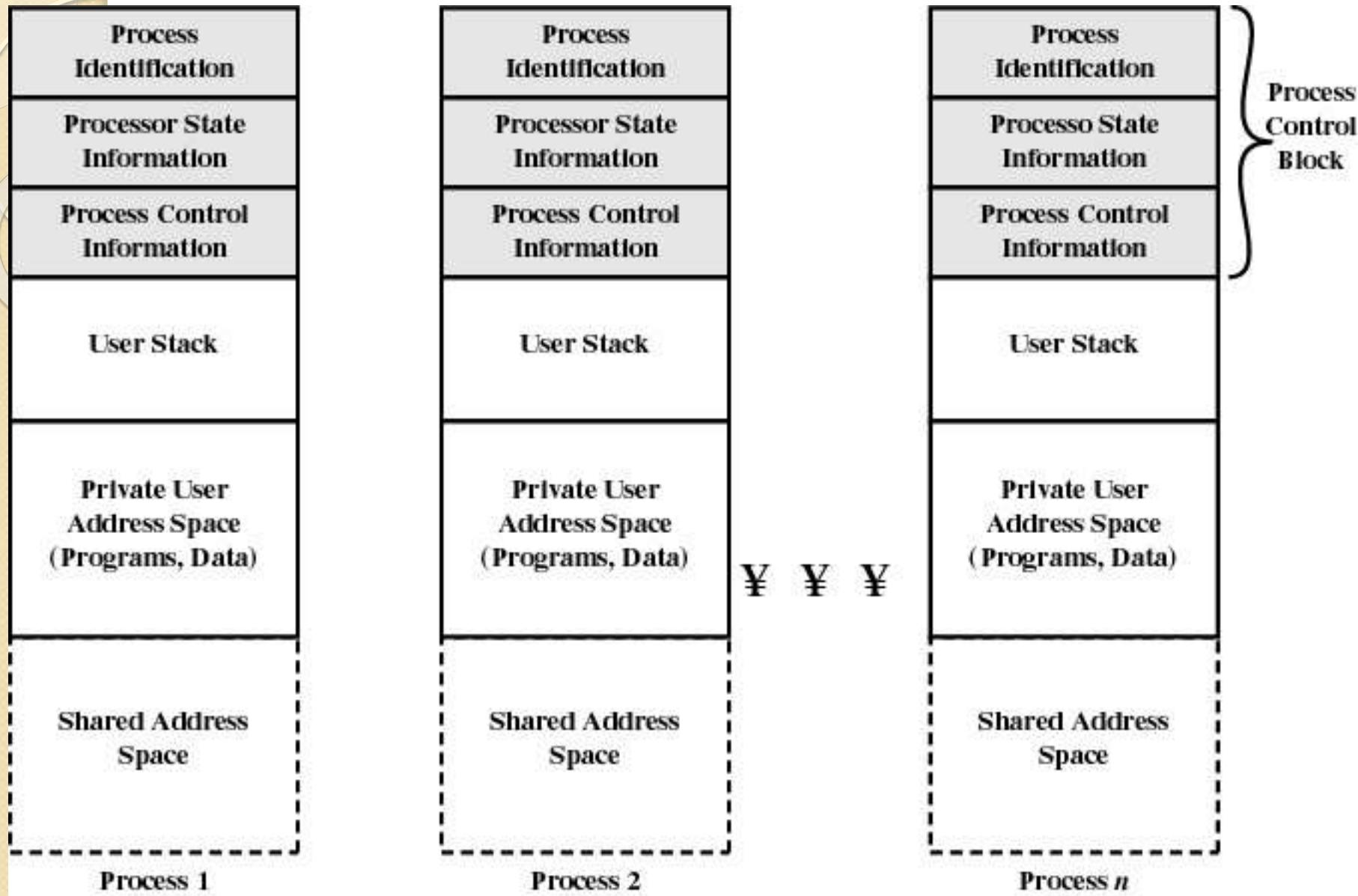


Figure 3.12 User Processes in Virtual Memory

# Processor State Information

- Contents of processor registers
  - User-visible registers
  - Control and status registers
  - Stack pointers
- Program status word (PSW)
  - contains status information
  - Example: the EFLAGS register on Pentium machines

# Modes of Execution

- User mode
  - Less-privileged mode
  - User programs typically execute in this mode
- System mode, control mode, or kernel mode
  - More-privileged mode
  - Kernel of the operating system

# Process Creation

- Assign a unique process identifier
- Allocate space for the process
- Initialize process control block
- Set up appropriate linkages
  - Ex: add new process to linked list used for scheduling queue
- Create or expand other data structures
  - Ex: maintain an accounting file

# When to Switch a Process ?

- Clock interrupt
  - process has executed for the maximum allowable time slice
- I/O interrupt
- Memory fault
  - memory address is in virtual memory so it must be brought into main memory

# When to Switch a Process

- Trap
  - error occurred
  - may cause process to be moved to Exit state
- Supervisor call
  - such as file open

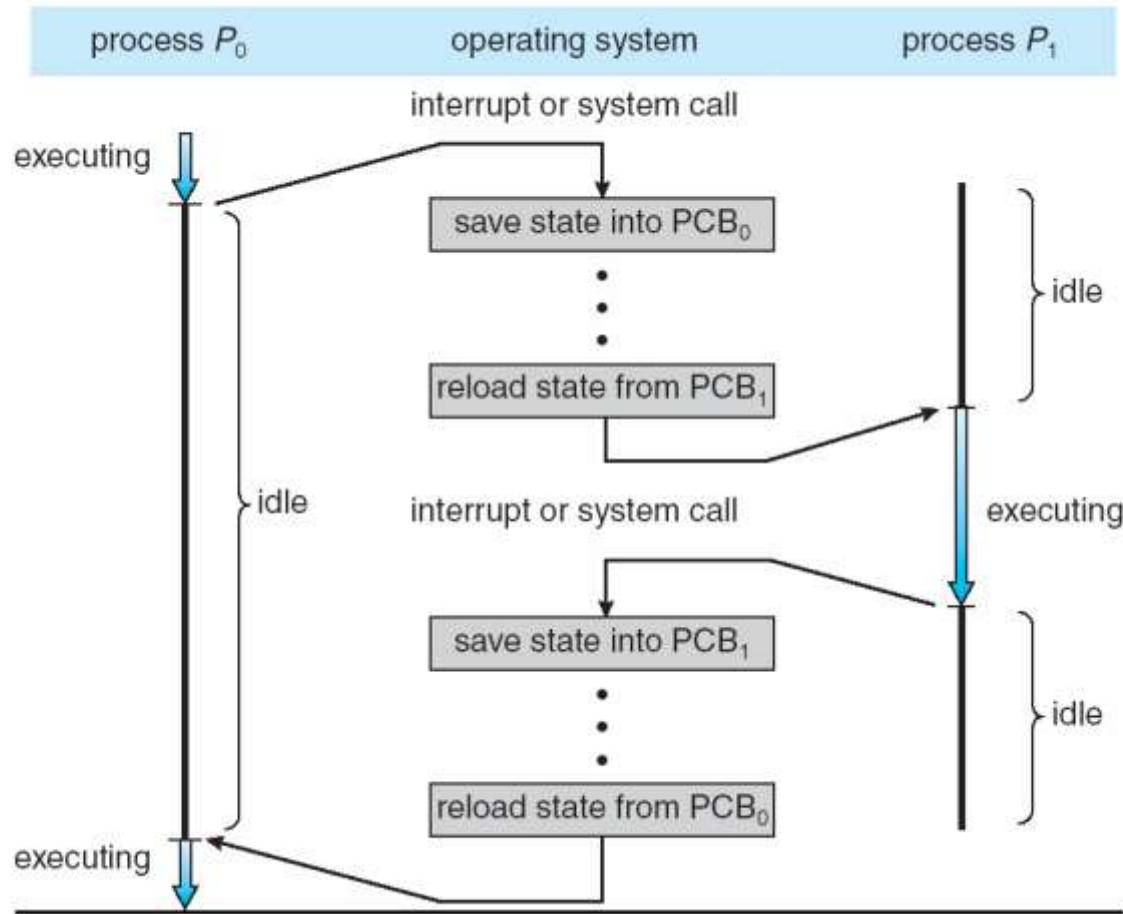
# Change of Process State

- Save context of processor including program counter and other registers
- Update the process control block of the process that is currently running
- Move process control block to appropriate queue - ready, blocked
- Select another process for execution

# **Change of Process State**

- Update the process control block of the process selected
- Update memory-management data structures
- Restore context of the selected process

# Switching between the Processes (context switch)

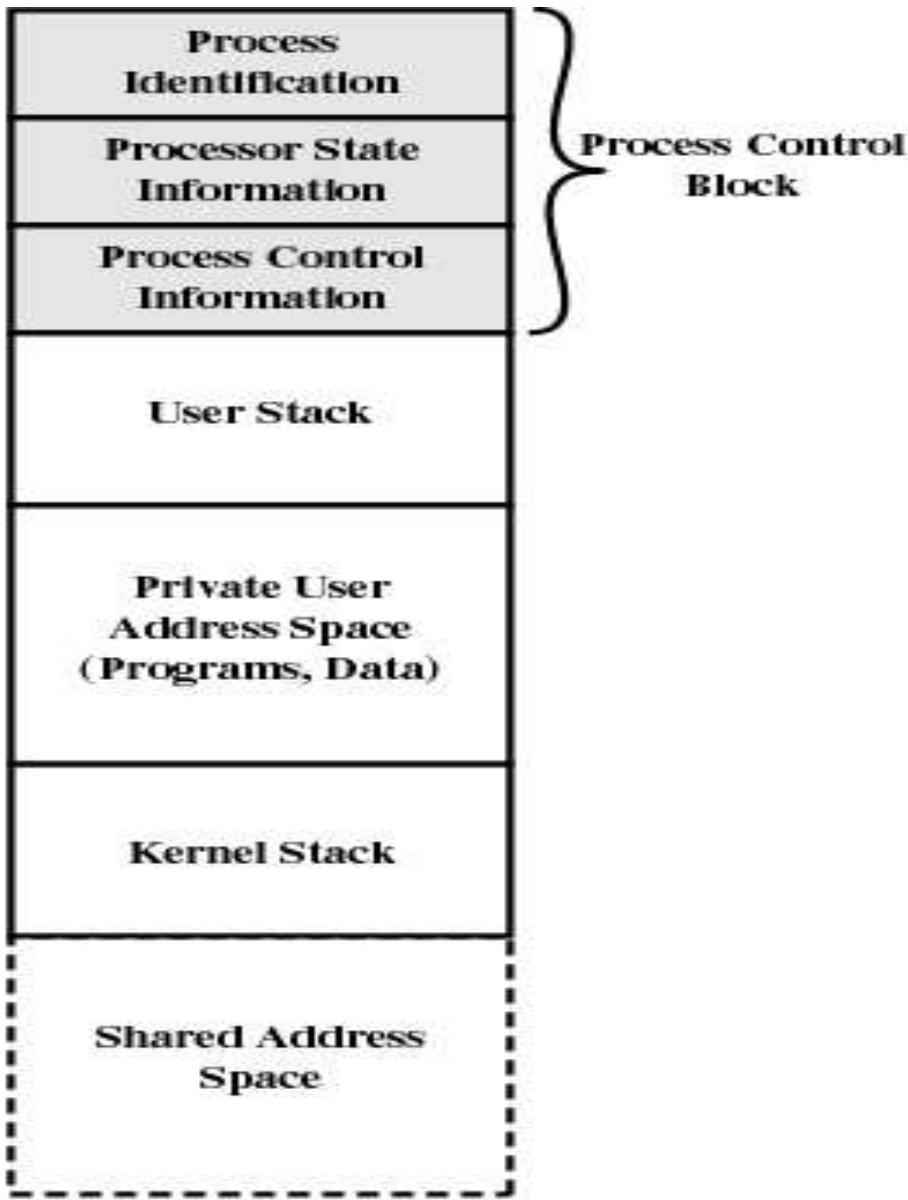


# Execution of the Operating System

- Non-process Kernel
  - execute kernel outside of any process
  - operating system code is executed as a separate entity that operates in privileged mode
- Execution Within User Processes
  - operating system software within context of a user process
  - process executes in privileged mode when executing operating system code

# Execution of the Operating System

- Process-Based Operating System
  - major kernel functions are separate processes
  - Useful in multi-processor or multi-computer environment



**Figure 3.15 Process Image: Operating System Executes Within User Space**

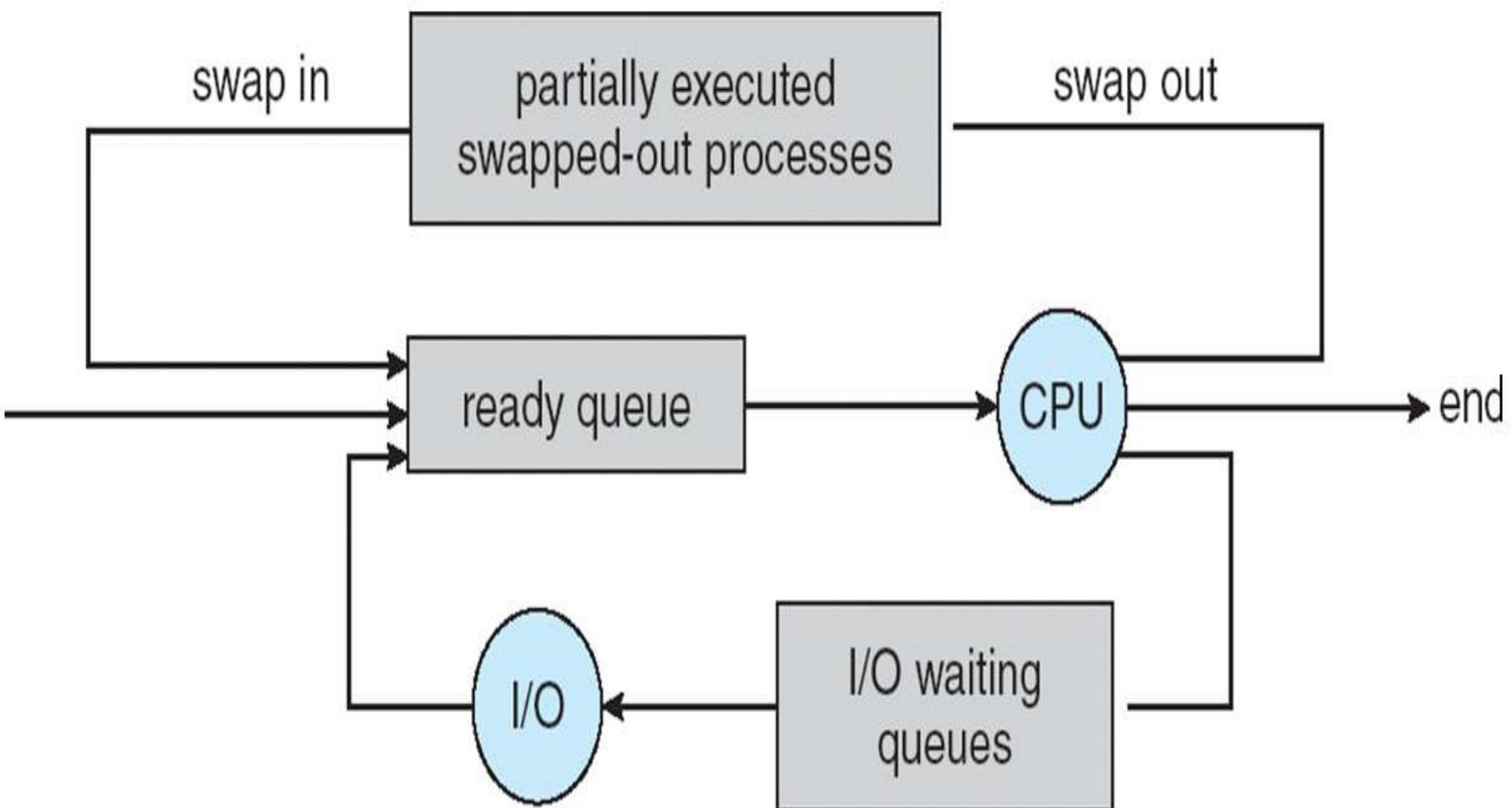
# III. Schedulers

- **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

# Schedulers (Cont)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (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

# Addition of Medium Term Scheduling



# Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation

A.

# Scheduling Objectives

- To introduce process scheduling, which is the basis for multi-programmed operating systems
- To describe various process-scheduling algorithms
- To discuss evaluation criteria for selecting a process-scheduling algorithm for a particular system

## B.

# Basic Concepts

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

•  
•  
•

**load store  
add store  
read from file**

*wait for I/O*

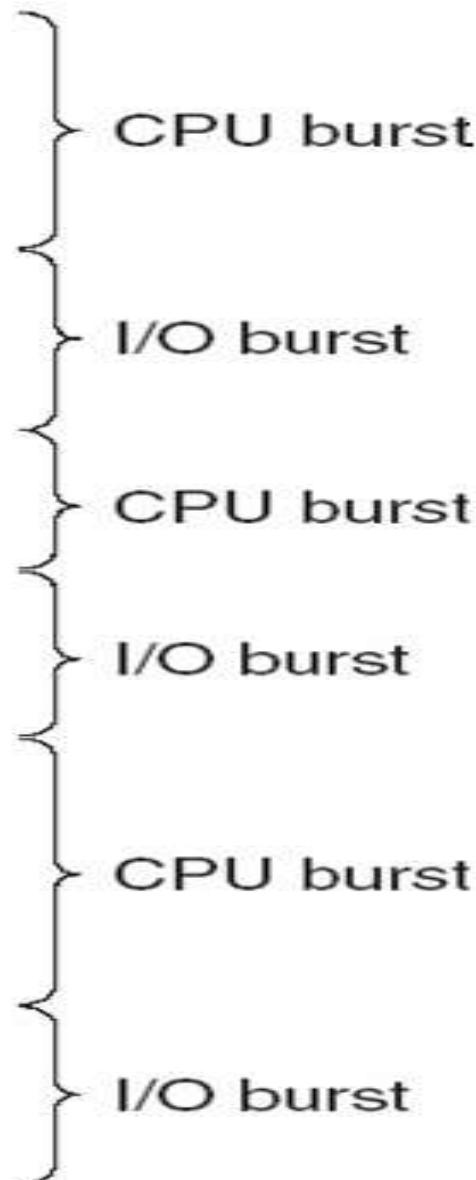
**store increment  
index  
write to file**

*wait for I/O*

**load store  
add store  
read from file**

*wait for I/O*

•  
•  
•



## C. 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 **non-preemptive**
- All other scheduling is **preemptive**

# Dispatching

- Dispatcher module gives control of the CPU to the process selected
  - 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 execution of another

## D. Scheduling Criteria

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

## E.

# Scheduling Algorithm Optimization Criteria

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

I.

## First-Come, First-Served (FCFS) Scheduling

Process    Burst Time

$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt Chart for the schedule is:



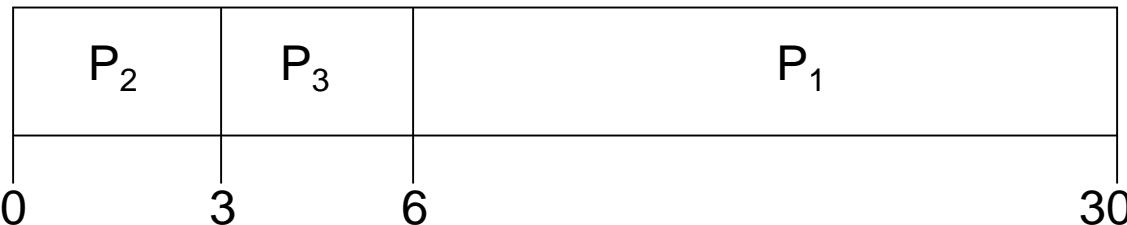
- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

# FCFS Scheduling (Cont)

Suppose that the processes arrive in the order

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for  $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time:  $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- Convoy effect short process behind long process*

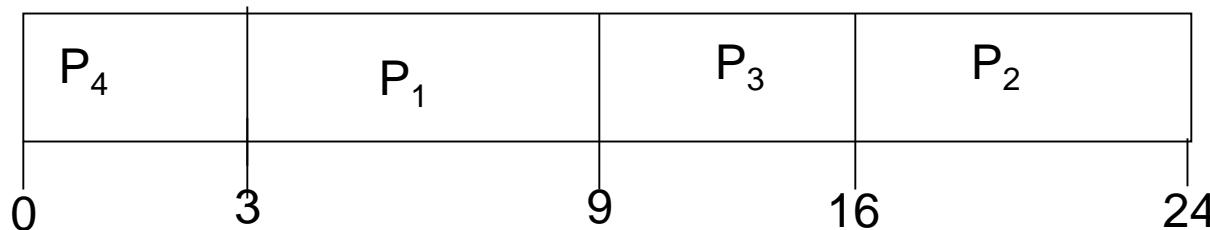
# Shortest-Job-First (SJF) Scheduling

- 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

# Example of SJF

Process	Arrival time	Burst time
$P_1$	0.0	6
$P_2$	2.0	8
$P_3$	4.0	7
$P_4$	5.0	3

- SJF scheduling chart



- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

### 3.

# 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
  - Non-preemptive
- 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

## 4.

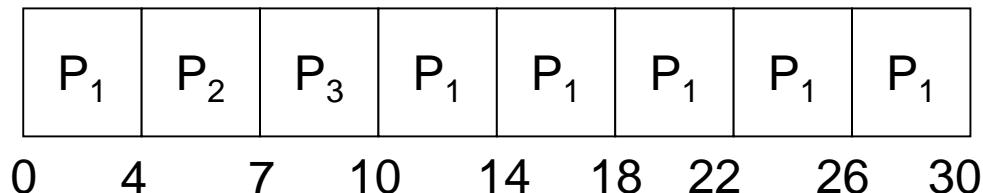
# Round Robin (RR)

- 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

# Example of RR with Time Quantum = 4

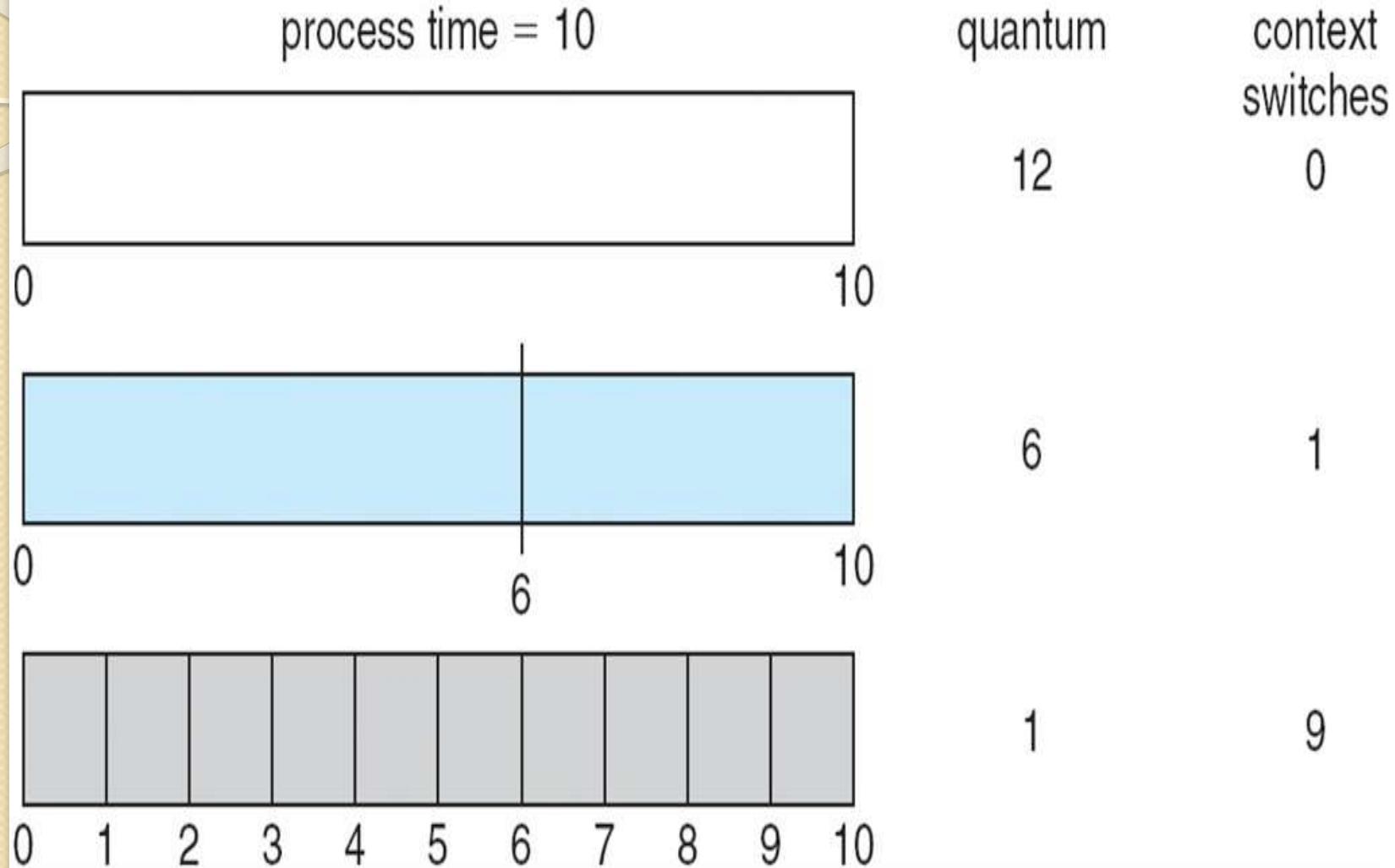
<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:

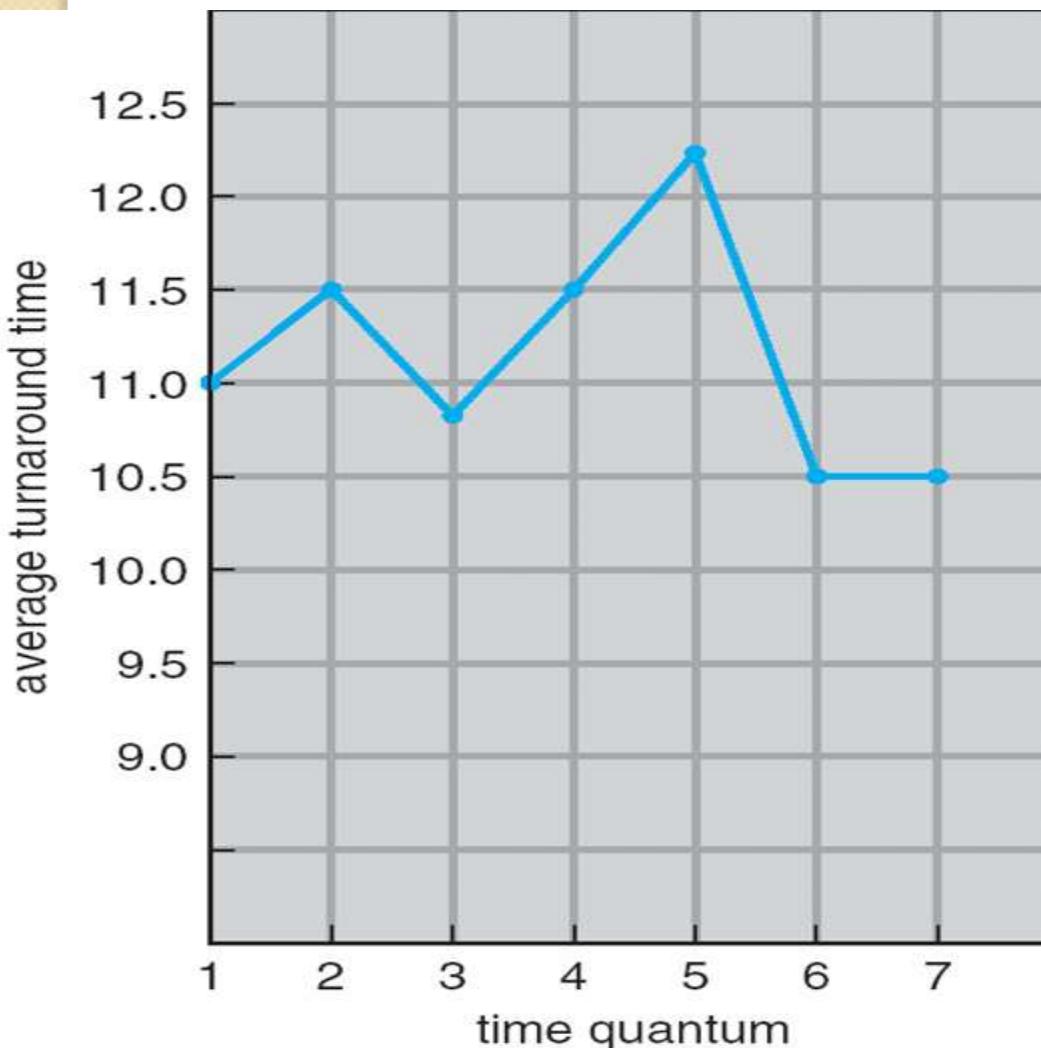


- Typically, higher average turnaround than SJF, but better response

# Time Quantum and Context Switch Time



# Turnaround Time Varies With The Time Quantum



process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

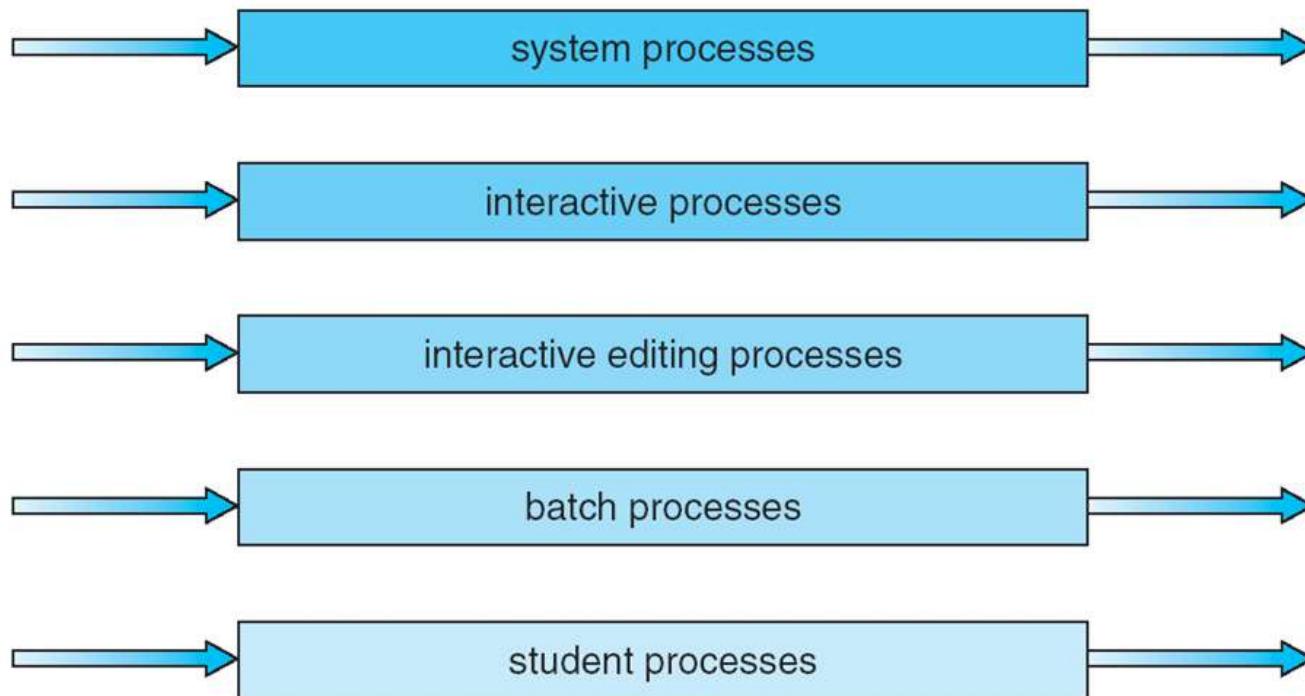
## 5.

# Multilevel Queue

- 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 Queue Scheduling

highest priority



lowest priority

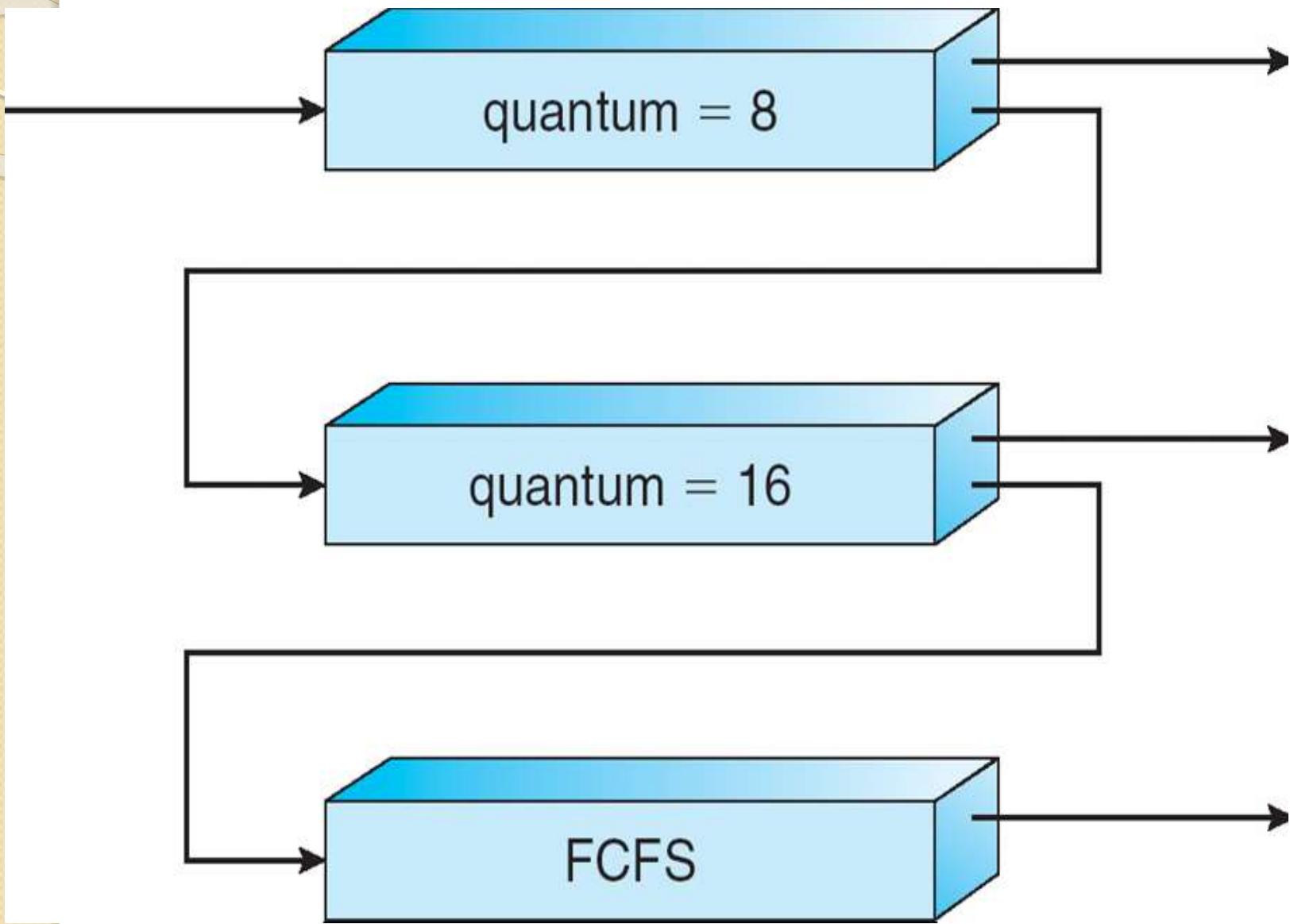
# Multilevel Feedback Queue

- 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

# Example:

- Three queues:
  - $Q_0$  – RR with time quantum 8 milliseconds
  - $Q_1$  – RR time quantum 16 milliseconds
  - $Q_2$  – FCFS
- Scheduling
  - A new job enters queue  $Q_0$  which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue  $Q_1$ .
  - At  $Q_1$  job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue  $Q_2$ .

# Multilevel Feedback Queues





## Unit 3

# Process Concurrency (Inter-process Communication)

## Mutual Exclusion and Synchronization



# Outline

- Principles of Concurrency
- Mutual Exclusion : Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

# Multiple Processes

- Central to the design of modern Operating Systems is managing multiple processes
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
- Big Issue is Concurrency
  - Managing the interaction of all of these processes

# Concurrency

Concurrency arises in:

- Multiple applications
  - Sharing time
- Structured applications
  - Extension of modular design
- Operating system structure
  - OS themselves implemented as a set of processes or threads

# Interleaving and Overlapping Processes

- Earlier we saw that processes may be interleaved on uniprocessors

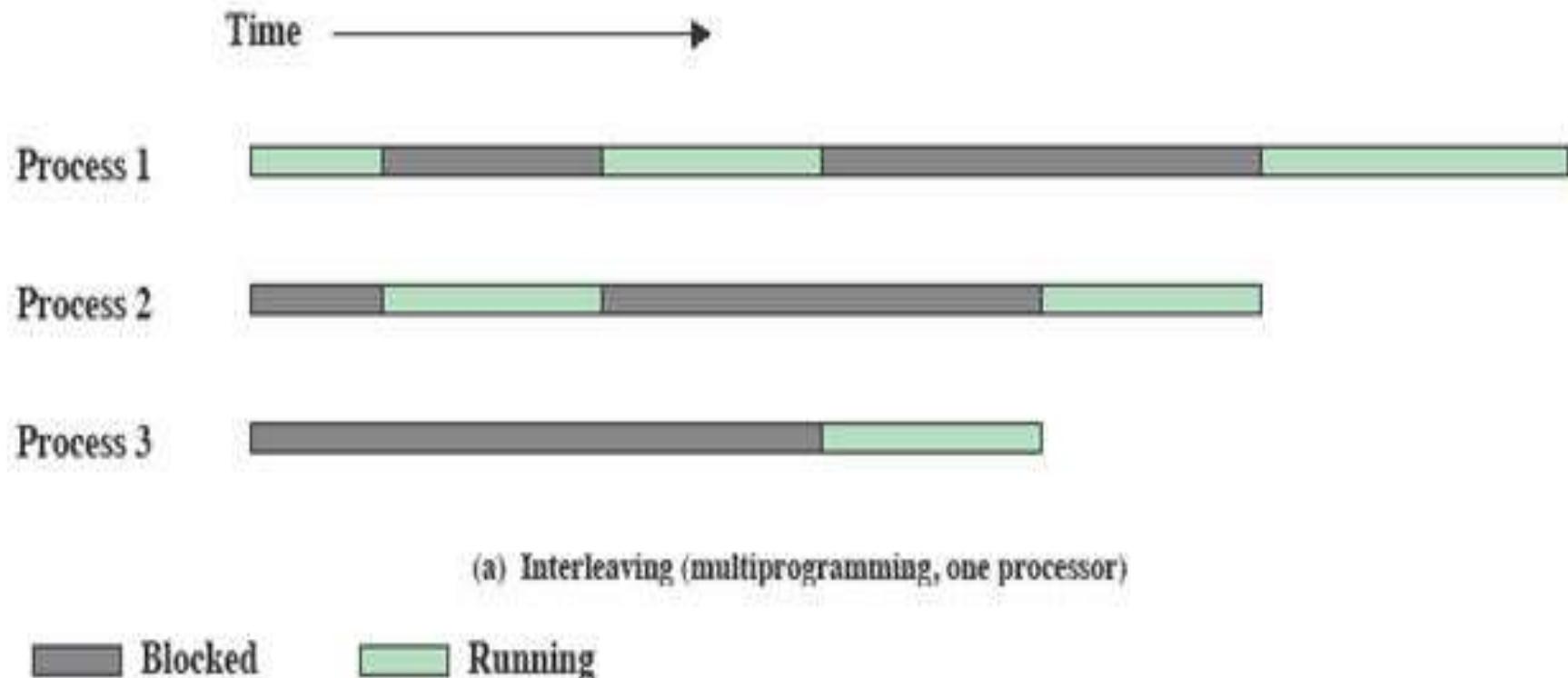


Figure 2.12 Multiprogramming and Multiprocessing

# Interleaving and Overlapping Processes

- And not only interleaved but overlapped on multi-processors

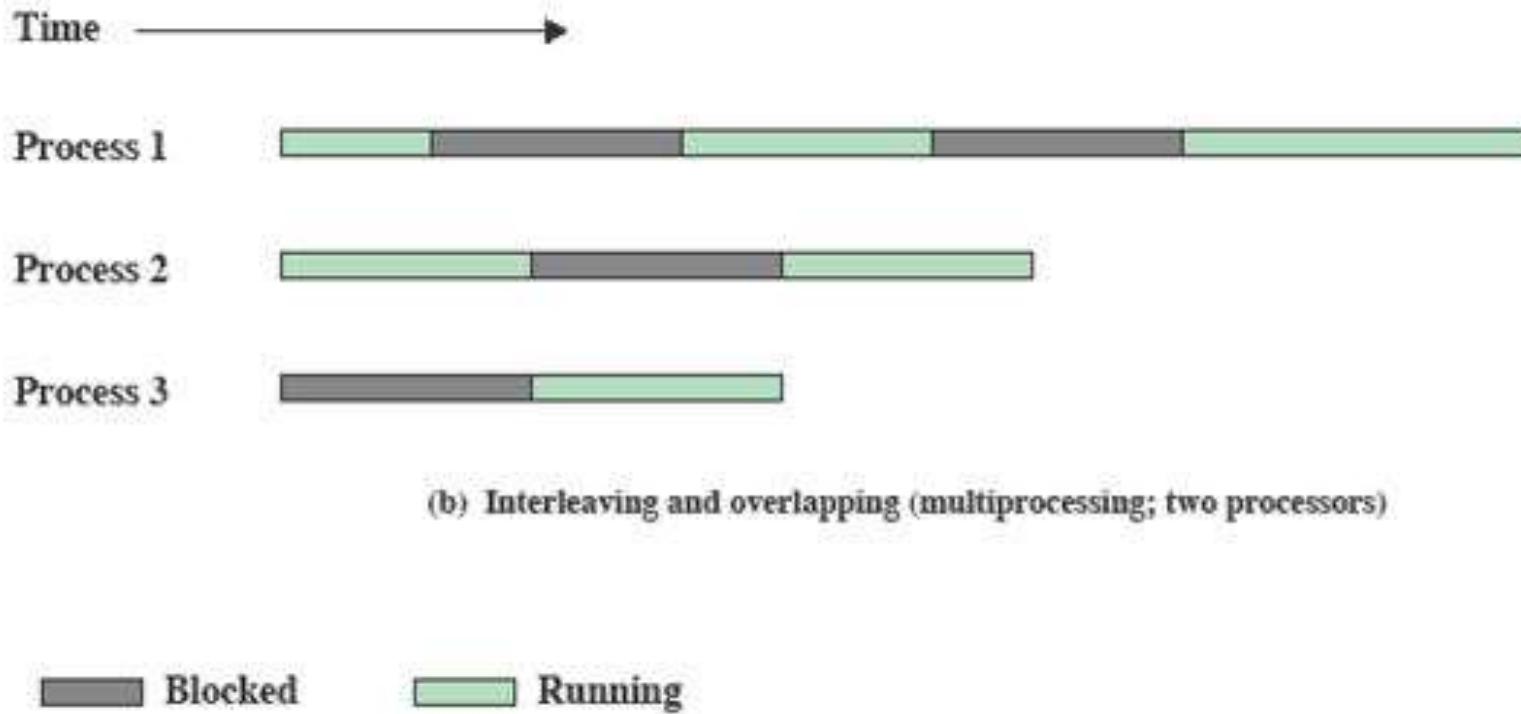


Figure 2.12 Multiprogramming and Multiprocessing

### 3. Difficulties of Concurrency

- Sharing of global resources  
(maintain the consistency)
- Optimally managing the allocation  
of resources (resource blocked)
- Difficult to locate programming  
errors ( running infinite loop)

# A Simple Example : Concurrency

```
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

# An example : Call of a function by two Processes

Process P1

```
{  
.  
.  
.  
echo(); // critical section  
.  
.  
.  
}
```

Process P2

```
{  
.  
.  
.  
echo(); // critical section  
.  
.  
.  
}
```

# An example : On a Multiprocessor system

Process P1

chin = getchar();

chout = chin;

putchar(chout);

Process P2

chin = getchar();

chout = chin;

putchar(chout);

# Solution: Enforce Single Access

- If we enforce a rule that only one process may enter the function at a time then :

## Scenario

- P1 & P2 run on separate processors
- P1 enters echo first,
  - P2 tries to enter but is blocked – P2 suspends
- P1 completes execution
  - P2 resumes and executes echo

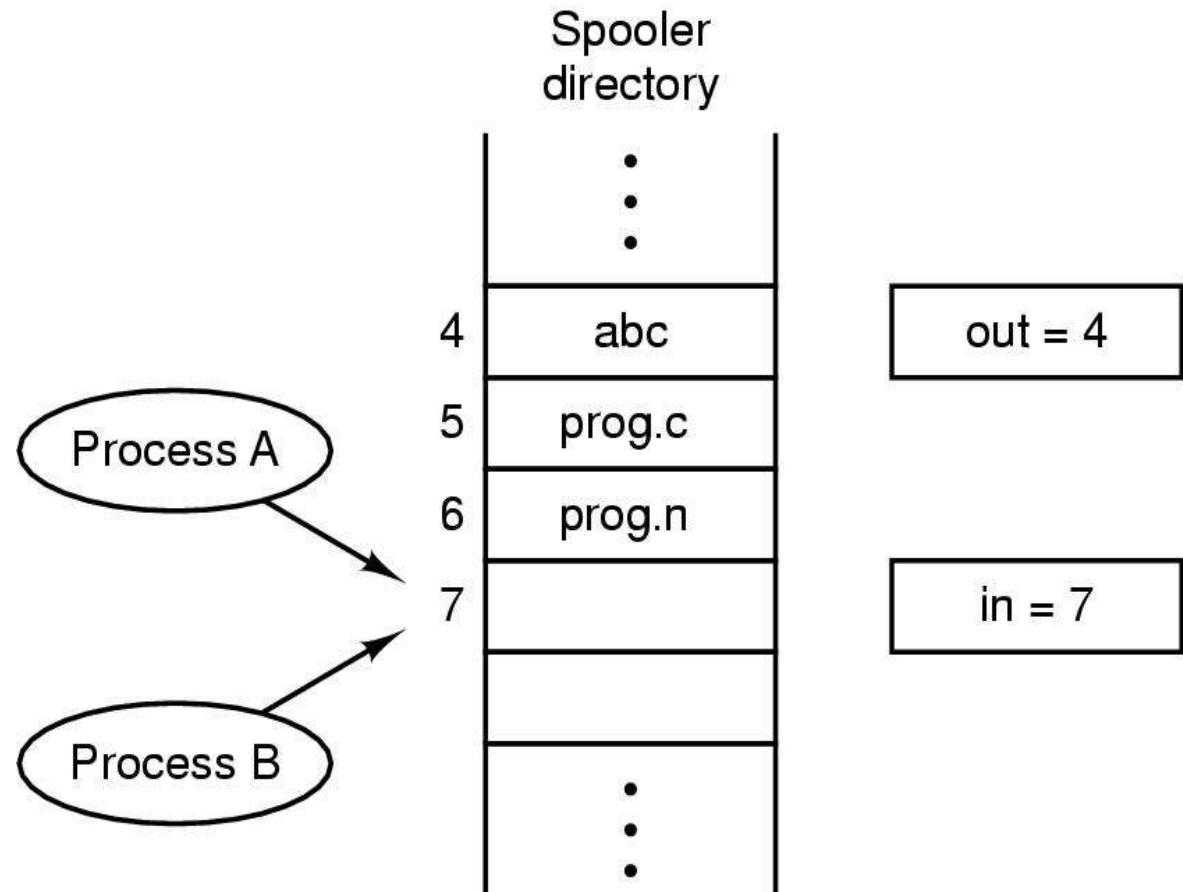
# Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write data items (Global resources)
  - Final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.

# IPC: Race Condition

## Race Condition

The situation where 2 or more processes are reading or writing some shared data is called race condition



Two processes want to access shared memory at same time

# Operating System Concerns

- What design and management issues are raised by the existence of concurrency?
- The OS must
  - Keep track of various processes
  - Allocate and de-allocate resources
  - Protect the data and resources against interference by other processes.
  - Ensure that the processes and outputs are independent of the processing speed

# Process Interaction

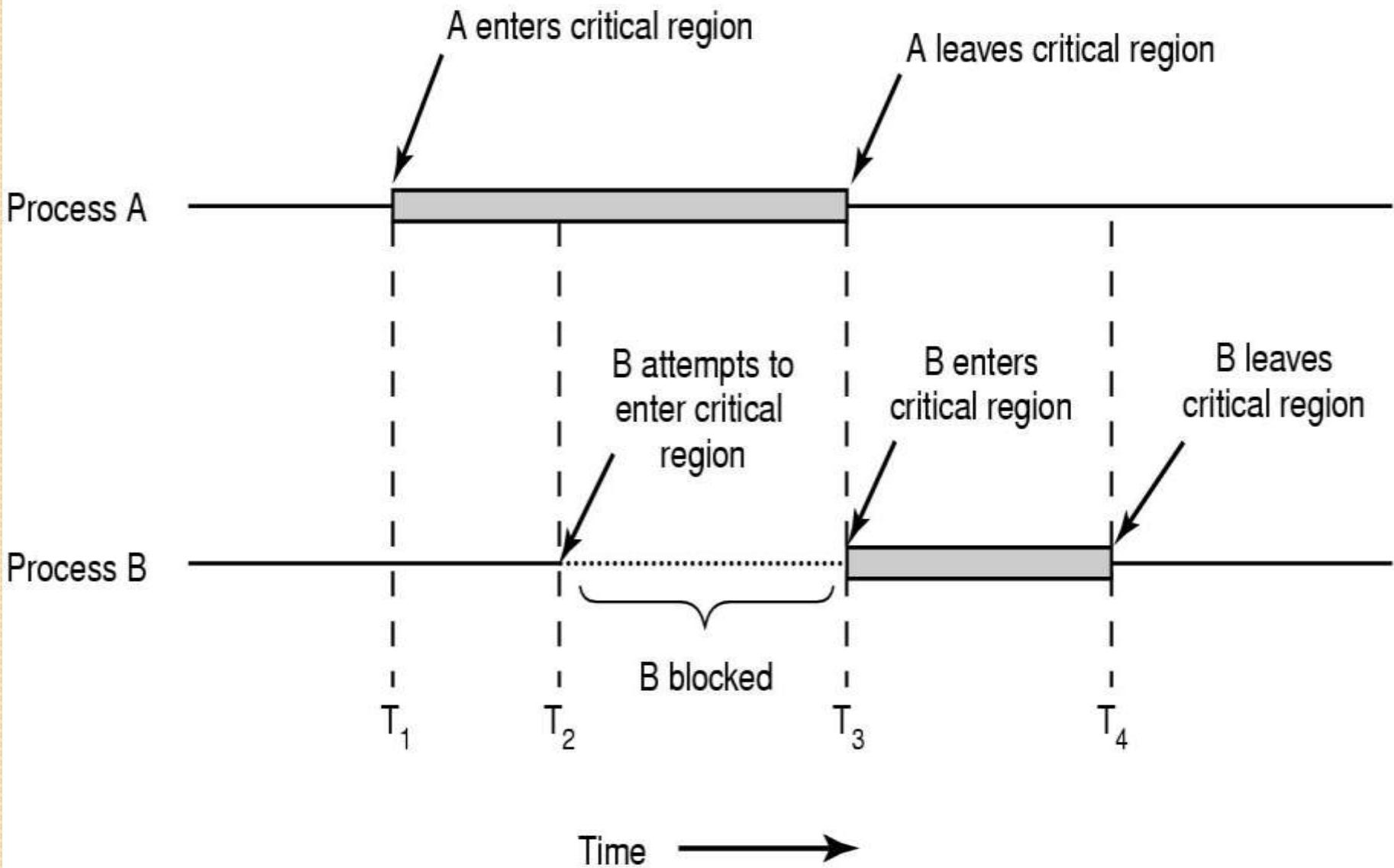
**Table 5.2** Process Interaction

Degree of Awareness	Relationship	Influence That One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	<ul style="list-style-type: none"><li>• Results of one process independent of the action of others</li><li>• Timing of process may be affected</li></ul>	<ul style="list-style-type: none"><li>• Mutual exclusion</li><li>• Deadlock (renewable resource)</li><li>• Starvation</li></ul>
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	<ul style="list-style-type: none"><li>• Results of one process may depend on information obtained from others</li><li>• Timing of process may be affected</li></ul>	<ul style="list-style-type: none"><li>• Mutual exclusion</li><li>• Deadlock (renewable resource)</li><li>• Starvation</li><li>• Data coherence</li></ul>
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	<ul style="list-style-type: none"><li>• Results of one process may depend on information obtained from others</li><li>• Timing of process may be affected</li></ul>	<ul style="list-style-type: none"><li>• Deadlock (consumable resource)</li><li>• Starvation</li></ul>

# Mutual Exclusion : Requirements

- Only one process at a time is allowed in the critical section for a resource
- A process that executes in its noncritical section must not interfere with other processes
- No deadlock or starvation
- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

# Mutual exclusion using Critical Regions





# Outline

- Principles of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

# I.

## Disabling Interrupts

- Uniprocessors only allow interleaving
- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Will not work in multiprocessor architecture

# Pseudo-Code

```
while (true)
{
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

# Synchronization Hardware : Problems

- Many systems provide hardware support for critical section code
- Uniprocessor – could disable interrupts
  - Currently running code would execute without preemption
  - Not supporting in multiprogramming environment
- Multiprocessors -
  - Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable

# Machine Instructions

- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - **Either test memory word and set value**
  - **Or Swap contents of two memory words**

# Mutual Exclusion: Hardware Support

- **Test and Set Instruction**

```
boolean TestAndSet (int lock)
{
    if (lock == 0)
    {
        lock = 1;
        return true;
    }
    else
    {
        return false;
    }
}
```

# Mutual Exclusion: Hardware Support

- Exchange Instruction

```
void Swap(int register,  
          int memory)  
{  
    int temp;  
    temp = memory;  
    memory = register;  
    register = temp;  
}
```

# Solution using TestAndSet

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

```
boolean TestAndSet (int lock) {  
    if (lock == 0)  
    {  
        lock = 1;  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}
```

Process - 1

```
do {  
    while ( TestAndSet (&lock )  
           ; // do nothing  
  
           // critical section  
  
           lock = FALSE;  
  
           // remainder section  
  
    } while (TRUE);
```

Process - 2

```
do {  
    while ( TestAndSet (&lock )  
           ; // do nothing  
  
           // critical section  
  
           lock = FALSE;  
  
           // remainder section  
  
    } while (TRUE);
```

# Solution using Swap

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

```
void Swap(int register,  
int memory)  
{  
int temp;  
temp = memory;  
memory = register;  
register = temp;  
}
```

## Solution:

```
Process - I  
do {  
    key = TRUE;  
    while ( key == TRUE && lock == FALSE)  
        Swap (&lock, &key );  
  
        // critical section  
  
    lock = FALSE;  
  
        // remainder section  
  
} while (TRUE);
```

# Mutual Exclusion Machine Instructions

- **Advantages**

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify
- It can be used to support multiple critical sections

# Mutual Exclusion Machine Instructions

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one processes are waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

# Outline

- Principles of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

# Semaphore

- Semaphore:
  - An integer value used for signalling among processes.
- Only three operations may be performed on a semaphore, all of which are atomic:
  - Initialize,
  - Decrement (semWait)
  - Increment. (semSignal)

# Semaphore Primitives

```
struct semaphore {
    int count;
    queueType queue;
};

void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignal(semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Figure 5.3 A Definition of Semaphore Primitives

# Binary Semaphore Primitives

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

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

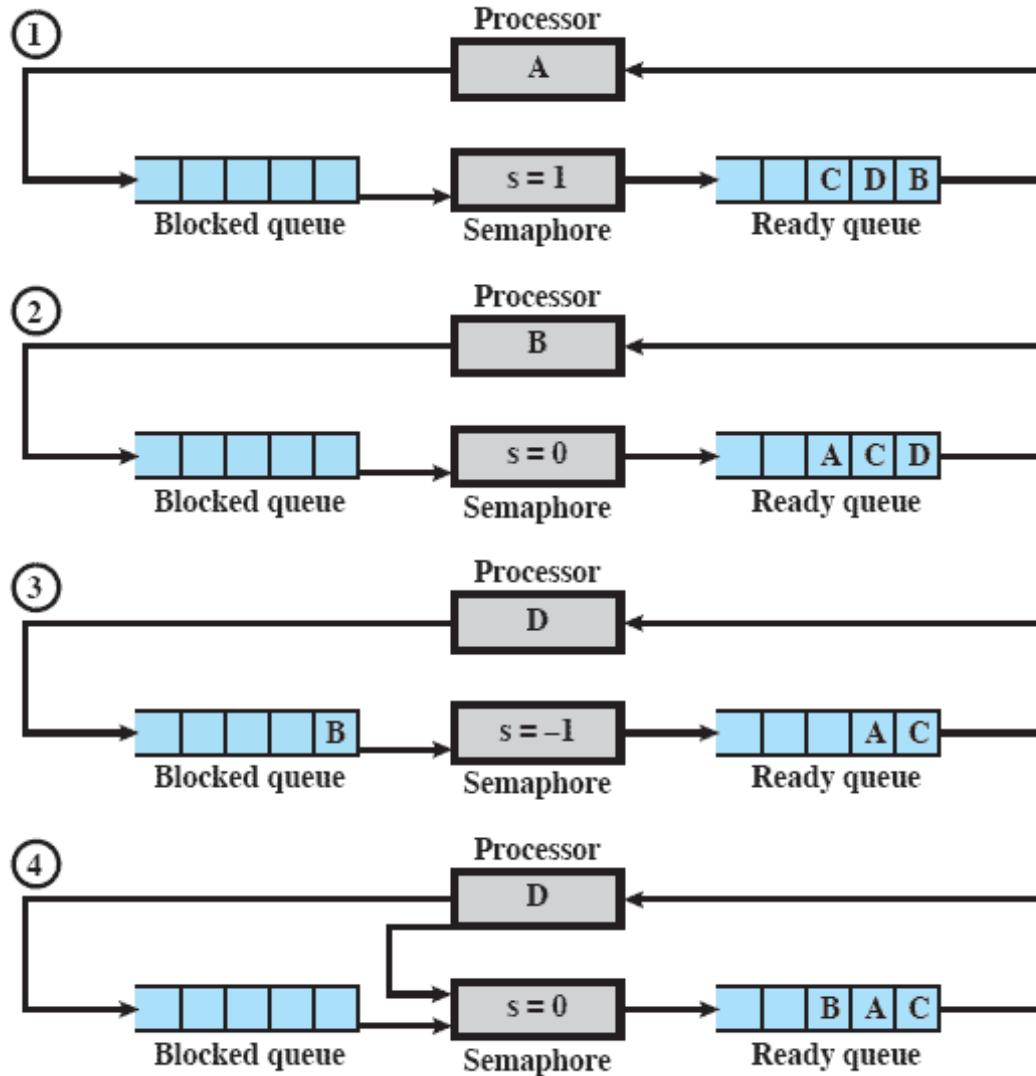
void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

# Strong/Weak Semaphore

- A queue is used to hold processes waiting on the semaphore
  - In what order are processes removed from the queue?
- **Strong Semaphores** use FIFO
- **Weak Semaphores** don't specify the order of removal from the queue

# Example of Strong Semaphore Mechanism



# Semaphore Primitives (Repeated)

```
struct semaphore {
    int count;
    queueType queue;
};

void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignal(semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Figure 5.3 A Definition of Semaphore Primitives

# Example of Semaphore Mechanism

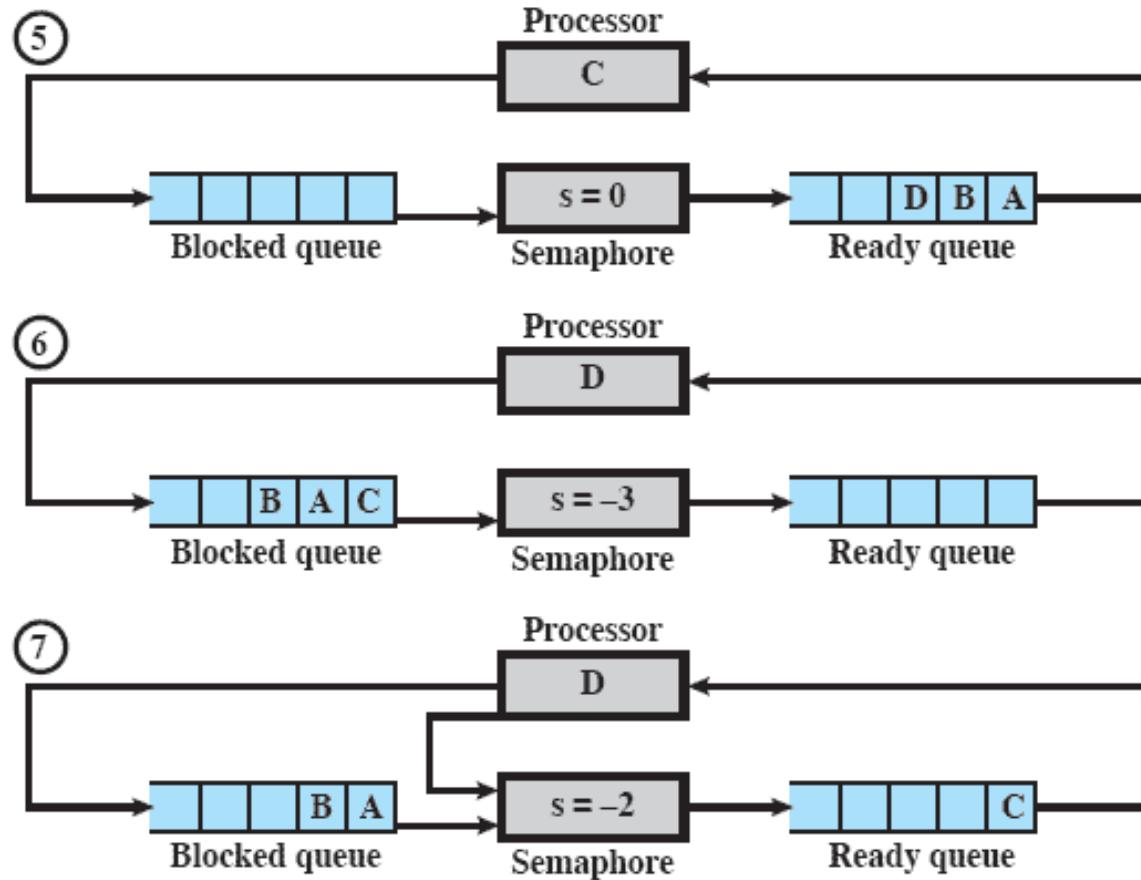


Figure 5.5 Example of Semaphore Mechanism

# Mutual Exclusion Using Semaphores

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true) {
        semWait(s);
        /* critical section */
        semSignal(s);
        /* remainder */
    }
}
void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores

# Semaphore Primitives (Repeated)

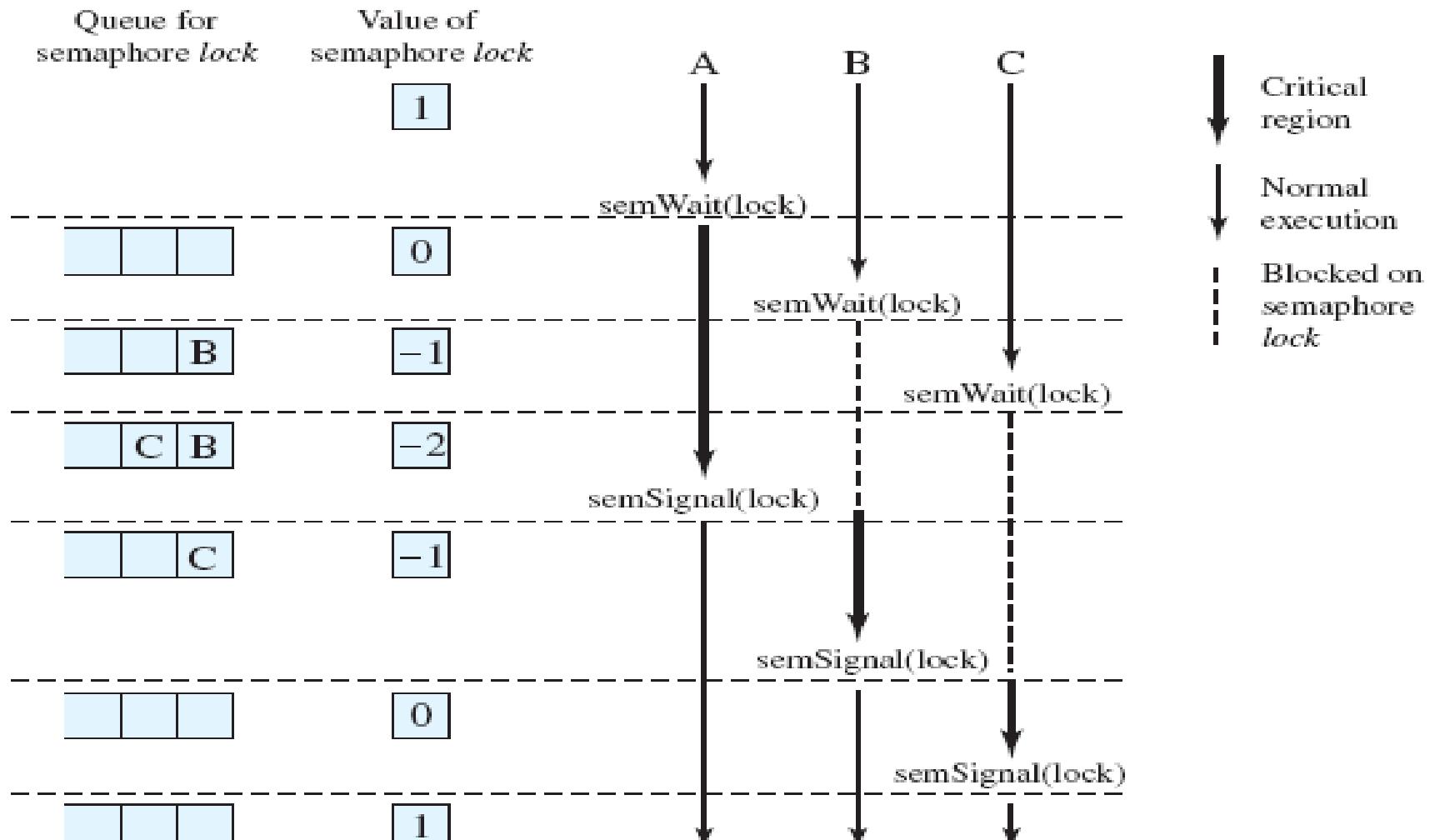
```
struct semaphore {
    int count;
    queueType queue;
};

void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignal(semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Figure 5.3 A Definition of Semaphore Primitives

# Processes Using Semaphore



*Note that normal execution can proceed in parallel but that critical regions are serialized.*

Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

# Producer/Consumer Problem

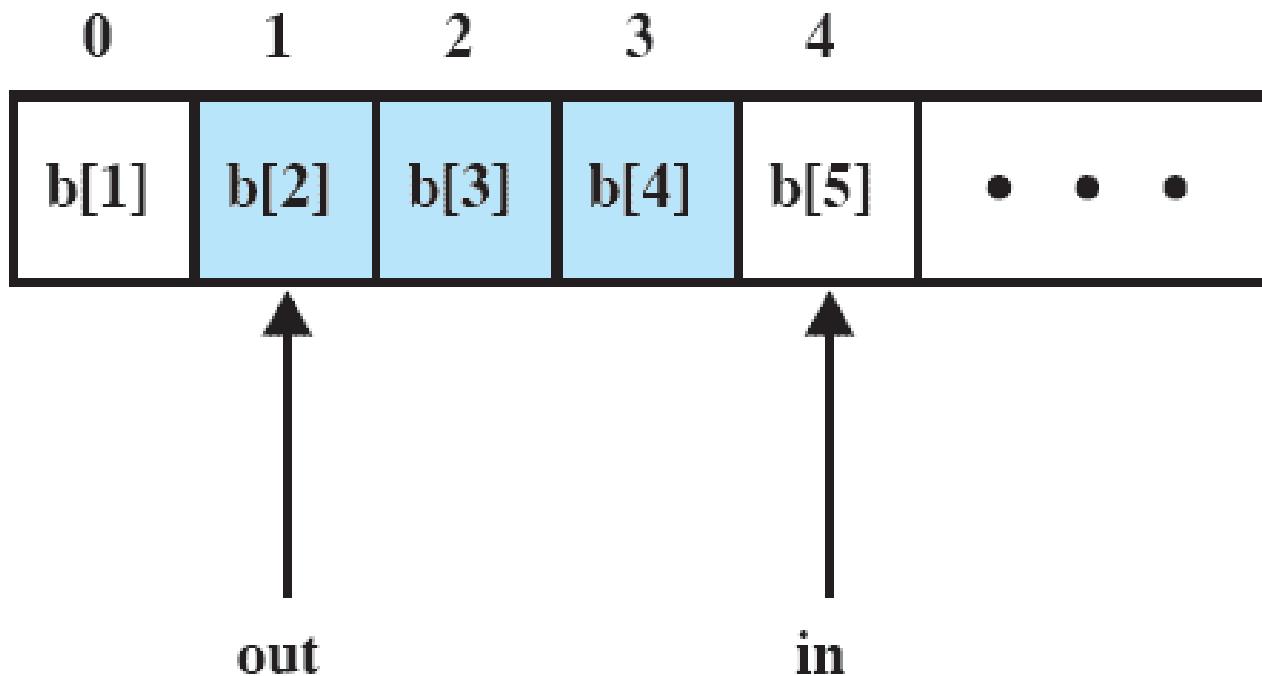
- General Situation:
  1. One or more producers are generating data and placing these in a buffer
  2. A single consumer is taking items out of the buffer one at time
  3. Only one producer or consumer may access the buffer at any one time
- **The Problem:**
  - Ensure that the Producer can't add data into full buffer and consumer can't remove data from empty buffer

# Functions

- Assume an infinite buffer  $b$  with a linear array of elements

Producer	Consumer
<pre>while (true) {     /* produce item v      */     b[in] = v;     in++; }</pre>	<pre>while (true) {     while (in &lt;= out)         /*do nothing */;     w = b[out];     out++;     /* consume item w      */ }</pre>

# Buffer



Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

# Incorrect Solution

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

# Possible Scenario

**Table 5.4** Possible Scenario for the Program of Figure 5.9

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	<b>if</b> (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	<b>if</b> (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		<b>if</b> (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		<b>if</b> (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	-1	0
21		semSignalB(s)	1	-1	0

*NOTE:* White areas represent the critical section controlled by semaphore s.

# Correct Solution

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

# Semaphores

```
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Figure 5.11 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores



# outline

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

# Monitors

- The monitor is a programming-language construct
- It provides equivalent functionality to that of semaphores
- It is easier to control.
- Implemented in a number of programming languages, including
  - Concurrent Pascal, Pascal-Plus,
  - Modula-2, Modula-3, and Java.

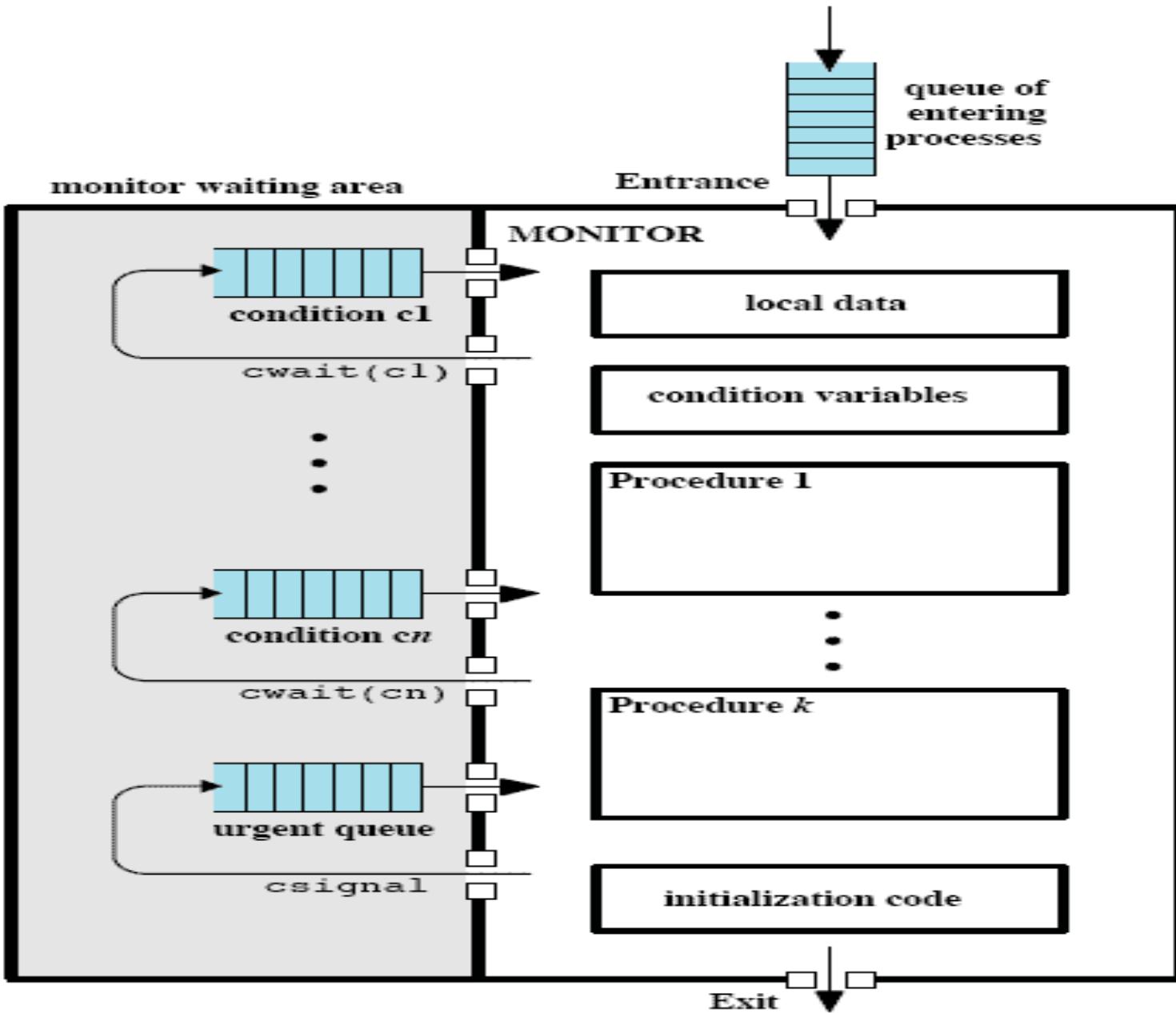
# Main characteristics

- Local data variables are accessible only by the monitor
- Process enters monitor by invoking one of its procedures
- Only one process may be executing in the monitor at a time

# Synchronization

- Synchronisation achieved by **condition variables** within a monitor
  - only accessible by the monitor.
- Monitor Functions:
  - **Cwait(c)**: Suspend execution of the calling process on condition c
  - **Csignal(c)** Resume execution of some process blocked after a cwait on the same condition

# Structure of a Monitor



# Monitors

- One have to be very careful while using semaphore
- Monitor is like class where only one procedure is active at one time

It is sufficient to put only the critical regions into monitor procedures as no two processes will ever execute their critical regions at the same time

**monitor** *example*

**integer** *i*;

**condition** *c*;

**procedure** *producer()*;

.

.

.

**end**;

**procedure** *consumer()*;

.

.

.

**end**;

**end monitor**;

# Bounded Buffer Solution Using Monitor

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                /* space for N items */
int nextin, nextout;                            /* buffer pointers */
int count;                                      /* number of items in buffer */
cond notfull, notempty;                         /* condition variables for synchronization */

void append (char x)
{
    if (count == N) cwait(notfull);           /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);                      /* resume any waiting consumer */
}

void take (char x)
{
    if (count == 0) cwait(notempty);           /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                                 /* one fewer item in buffer */
    csignal(notfull);                        /* resume any waiting producer */
}

                                         /* monitor body */
nextin = 0; nextout = 0; count = 0;             /* buffer initially empty */
}
```

# Bounded Buffer Monitor

```
void append (char x)
{
    while(count == N) cwait(notfull); /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;                                /* one more item in buffer */
    cnotify(notempty);                      /* notify any waiting consumer */
}

void take (char x)
{
    while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                                /* one fewer item in buffer */
    cnotify(notfull);                      /* notify any waiting producer */
}
```

Figure 5.17 Bounded Buffer Monitor Code for Mesa Monitor

# outline

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

# Process Interaction

- When processes interact with one another, two fundamental requirements must be satisfied:
  - synchronization and
  - communication.
- Message Passing is one solution to the second requirement
  - Added bonus: It works with shared memory and with distributed systems

# Message Passing

- The actual function of message passing is normally provided in the form of a pair of primitives:
  - send (destination, message)
  - receive (source, message)

# Synchronization

- Communication requires synchronization
  - Sender must send before receiver can receive
- What happens to a process after it issues a send or receive primitive?
  - Sender and receiver may or may not be blocking (waiting for message)

# Blocking send, Blocking receive

- Both sender and receiver are blocked until message is delivered
- Allows for tight synchronization between processes.

# Non-blocking Send

- More natural for many concurrent programming tasks.
- Nonblocking send, blocking receive
  - Sender continues on
  - Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - Neither party is required to wait

# Addressing

- Sendin process need to be able to specify which process should receive the message
  - Direct addressing
  - Indirect Addressing

# Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive could know ahead of time which process a message is expected
- Receive primitive could use source parameter to return a value when the receive operation has been performed

# Indirect addressing

- Messages are sent to a shared data structure consisting of queues
- Queues are called *mailboxes*
- One process sends a message to the mailbox and the other process picks up the message from the mailbox

# Indirect Process Communication

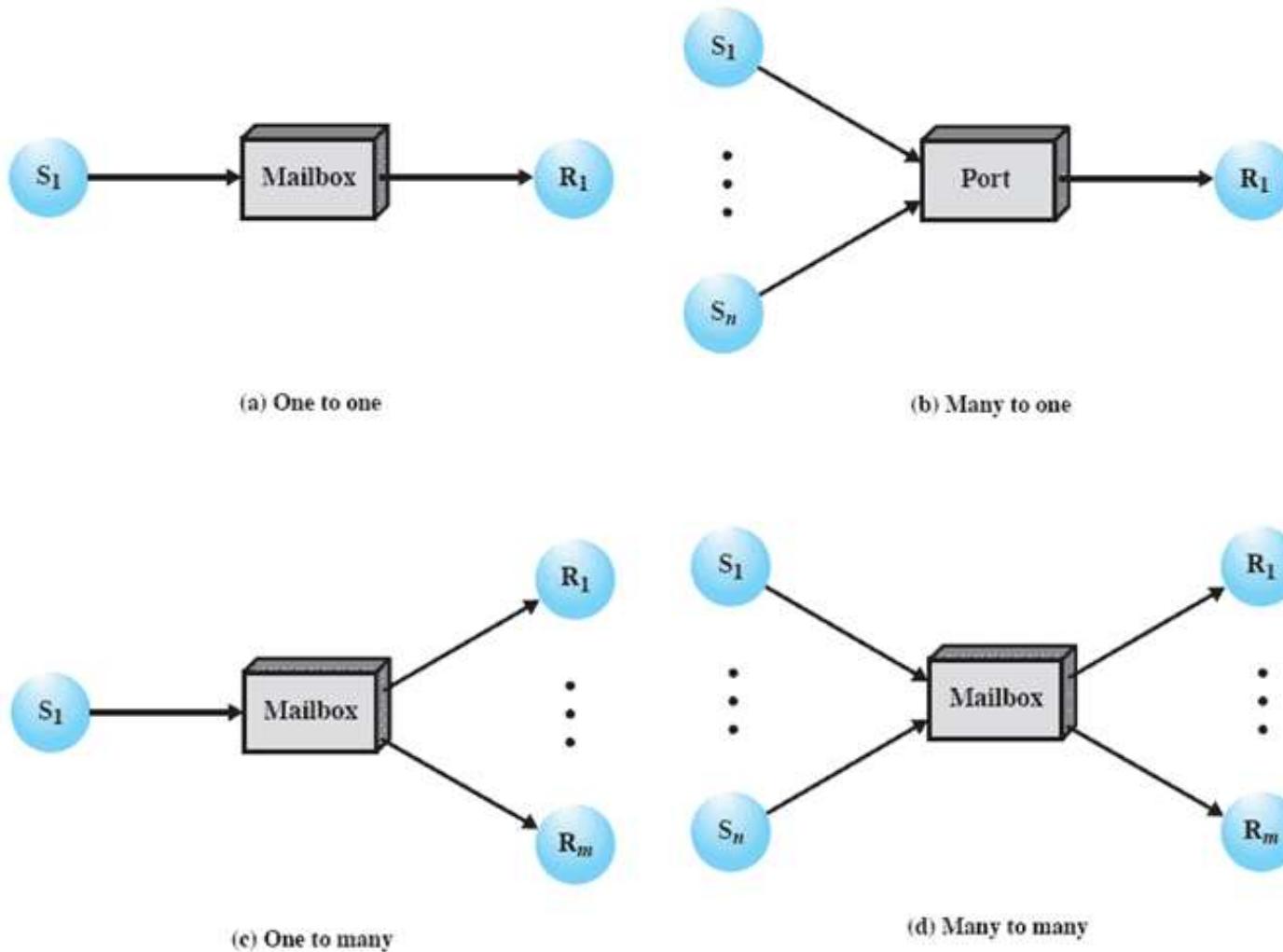


Figure 5.18 Indirect Process Communication

# General Message Format

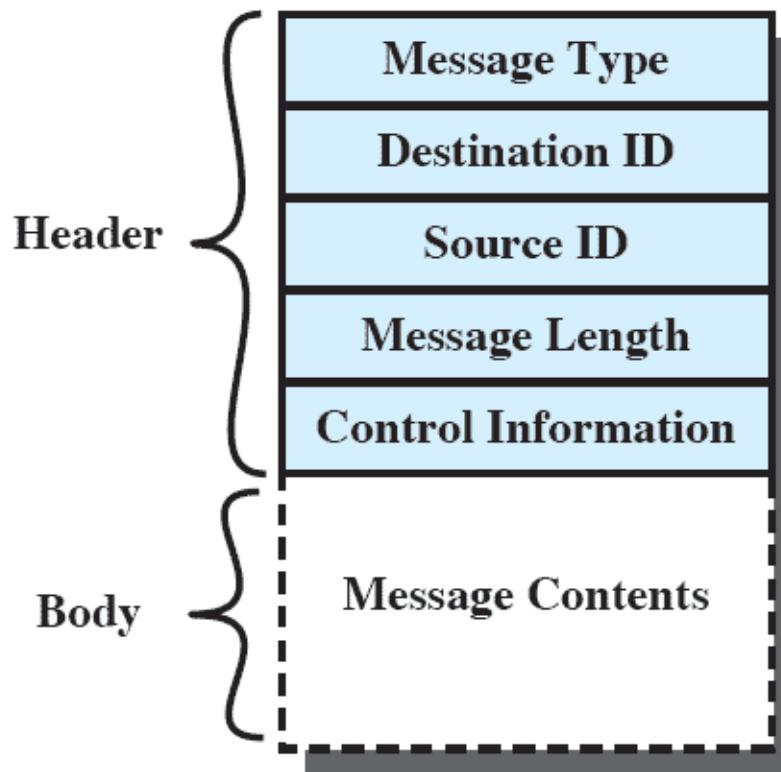


Figure 5.19 General Message Format

# Mutual Exclusion Using Messages

```
/* program mutual exclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
        /* critical section */
        send (box, msg);
        /* remainder */
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . . , P(n));
}
```

Figure 5.20 Mutual Exclusion Using Messages



# outline

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

# Readers/Writers Problem

- A data area is shared among many processes
  - Some processes only read the data area, some only write to the area
- Conditions to satisfy:
  1. Multiple readers may read the file at once.
  2. Only one writer at a time may write
  3. If a writer is writing to the file, no reader may read it.
- Priority
  1. Readers have priority
  2. Writers have priority

# Reader have priority : Regulations

- No reader will be kept waiting unless writer has already obtained the permission to write
- No reader should wait for other reader to finish simply because writer is waiting
- **Problem :** In this case there is possibility of starvation for the writers

# Writer have priority : Regulations

- Once the writer is ready that writer performs its write as soon as possible
- Writer is waiting to write, new reader will not perform read operation.
- **Problem :** In this case there is a possibility of starvation for readers

# Readers have Priority

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer()
{
    while (true) {
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
    }
}
void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

# Writers have Priority

```
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
```

# Writers have Priority

```
void writer ()
{
    while (true) {
        semWait (y);
        writecount++;
        if (writecount == 1) semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}

void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

# Unit – 4

# Deadlock

# Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes

# Deadlock

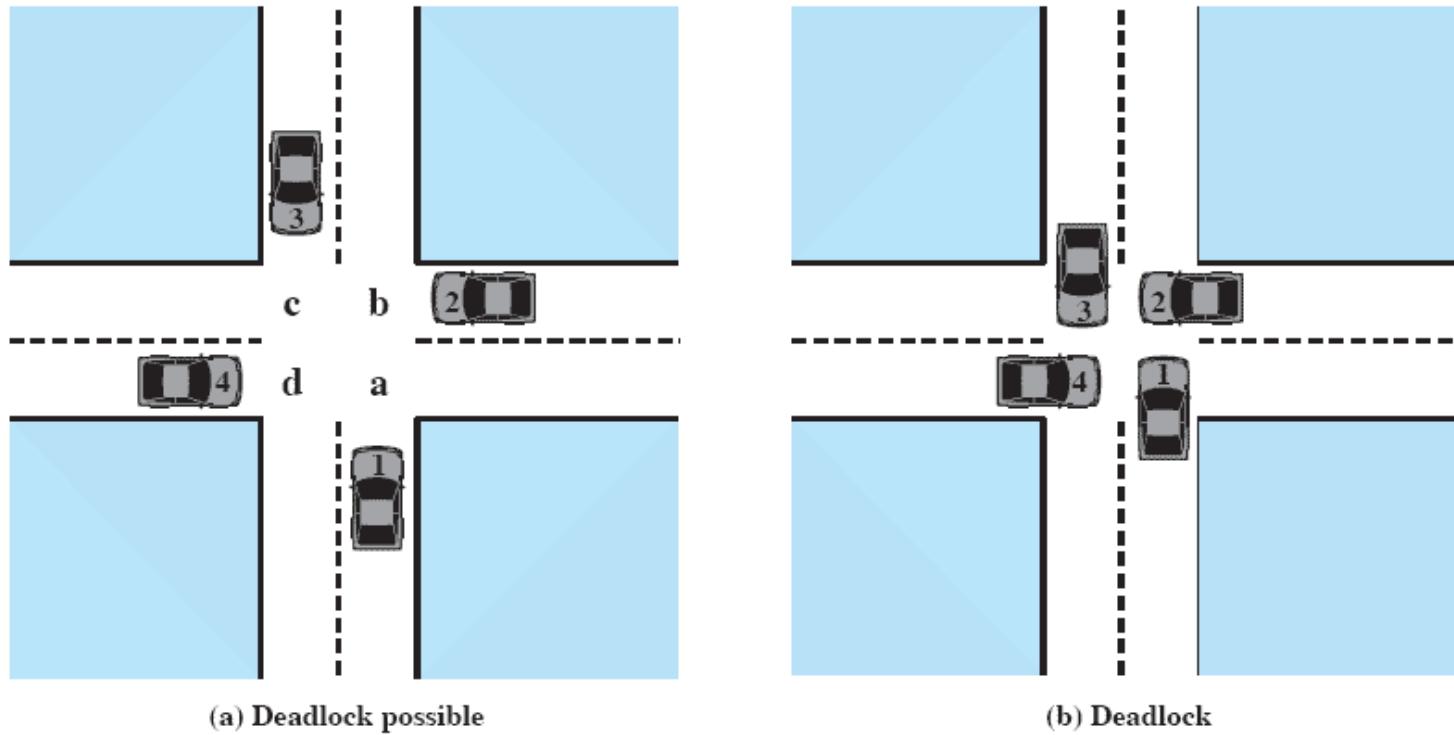


Figure 6.1 Illustration of Deadlock

# Reusable Resources

- Used by only one process at a time and not exhausted by that use.
- Processes obtain resources that they later release for reuse by other processes.
- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other

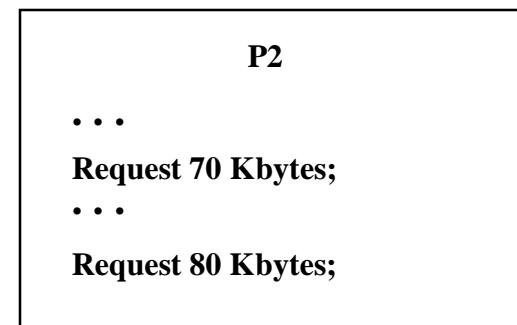
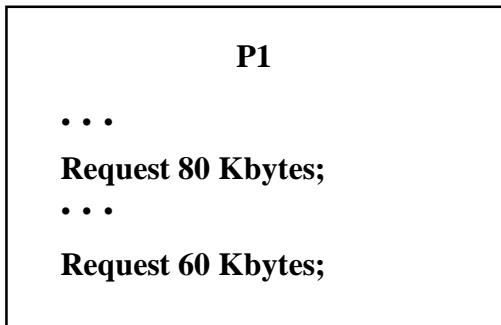
# Reusable Resources

Process P		Process Q	
Step	Action	Step	Action
p <sub>0</sub>	Request (D)	q <sub>0</sub>	Request (T)
p <sub>1</sub>	Lock (D)	q <sub>1</sub>	Lock (T)
p <sub>2</sub>	Request (T)	q <sub>2</sub>	Request (D)
p <sub>3</sub>	Lock (T)	q <sub>3</sub>	Lock (D)
p <sub>4</sub>	Perform function	q <sub>4</sub>	Perform function
p <sub>5</sub>	Unlock (D)	q <sub>5</sub>	Unlock (T)
p <sub>6</sub>	Unlock (T)	q <sub>6</sub>	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

# Reusable Resources

- Space is available for allocation of 200Kbytes, and the following sequence of events occur



- Deadlock occurs if both processes progress to their second request

# Consumable Resources

- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

# Resource Allocation Graphs

- Directed graph that depicts a state of the system of resources and processes



(a) Resource is requested

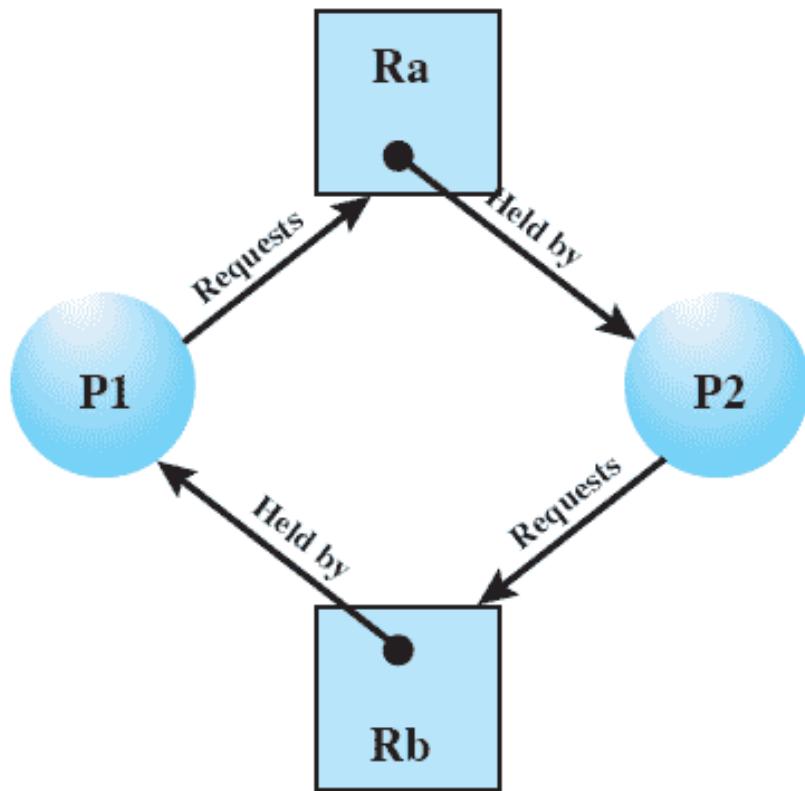


(b) Resource is held

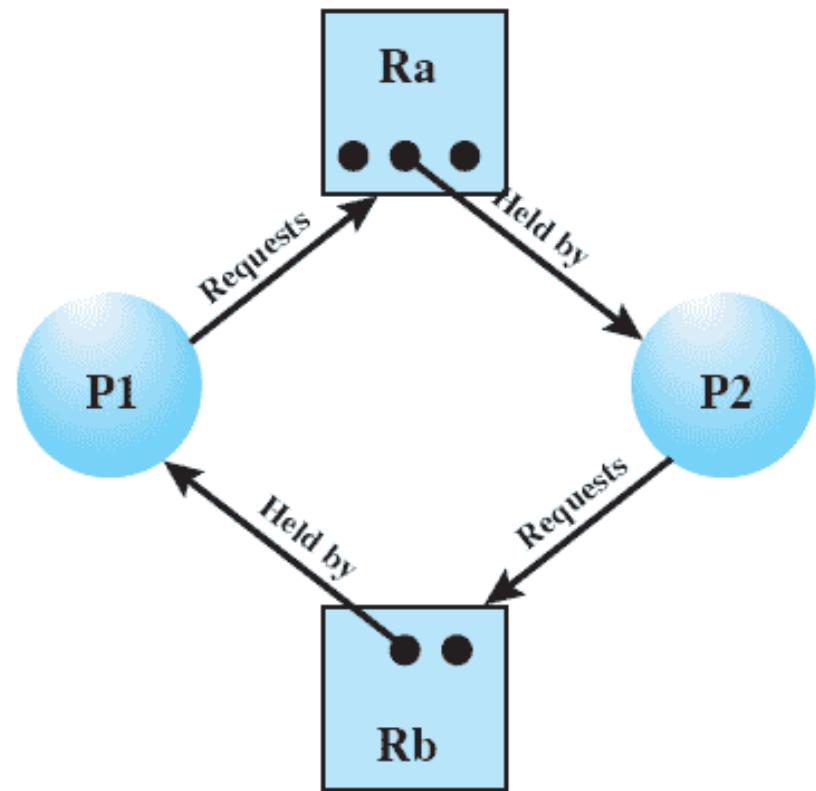
# Conditions for Deadlock

- Mutual exclusion
  - Only one process may use a resource at a time
- Hold-and-wait
  - A process may hold allocated resources while awaiting assignment of others
- No preemption
  - No resource can be forcibly removed from a process holding it
- Circular wait
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

# Resource Allocation Graphs



(c) Circular wait



(d) No deadlock

# Resource Allocation Graphs

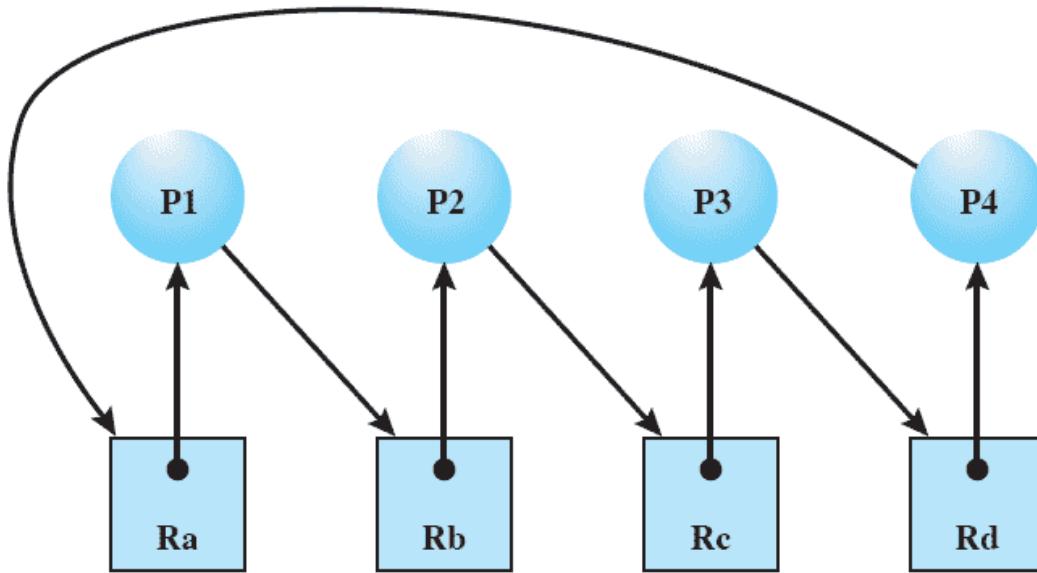


Figure 6.6 Resource Allocation Graph for Figure 6.1b

# Possibility of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait

# Existence of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait
- Circular wait

# Deadlock Prevention

Restrain the ways request can be made

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

# Deadlock Prevention (Cont.)

- **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

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process requests

# Two Approaches to Deadlock Avoidance

- Do not start a process if its demands might lead to deadlock
- Do not grant an incremental resource request to a process if this allocation might lead to deadlock

# Resource Allocation Denial (Banker's Algorithm)

- Referred to as the banker's algorithm
- State of the system is the current allocation of resources to process
- Safe state is where there is at least one sequence that does not result in deadlock
- Unsafe state is a state that is not safe

# Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

$C - A$

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	0	1	1

Available vector V

(a) Initial state

# Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	6	2	3

Available vector V

(b) P2 runs to completion

# Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	7	2	3

Available vector V

(c) P1 runs to completion

# Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

$C - A$

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	9	3	4

Available vector V

(d) P3 runs to completion

# Determination of an Unsafe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix  $\mathbf{C}$

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix  $\mathbf{A}$

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

$\mathbf{C} - \mathbf{A}$

	R1	R2	R3
	9	3	6

Resource vector  $\mathbf{R}$

	R1	R2	R3
	1	1	2

Available vector  $\mathbf{V}$

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix  $\mathbf{C}$

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix  $\mathbf{A}$

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

$\mathbf{C} - \mathbf{A}$

	R1	R2	R3
	9	3	6

Resource vector  $\mathbf{R}$

	R1	R2	R3
	0	1	1

Available vector  $\mathbf{V}$

(b) P1 requests one unit each of R1 and R3

# Deadlock Avoidance

- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent; no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

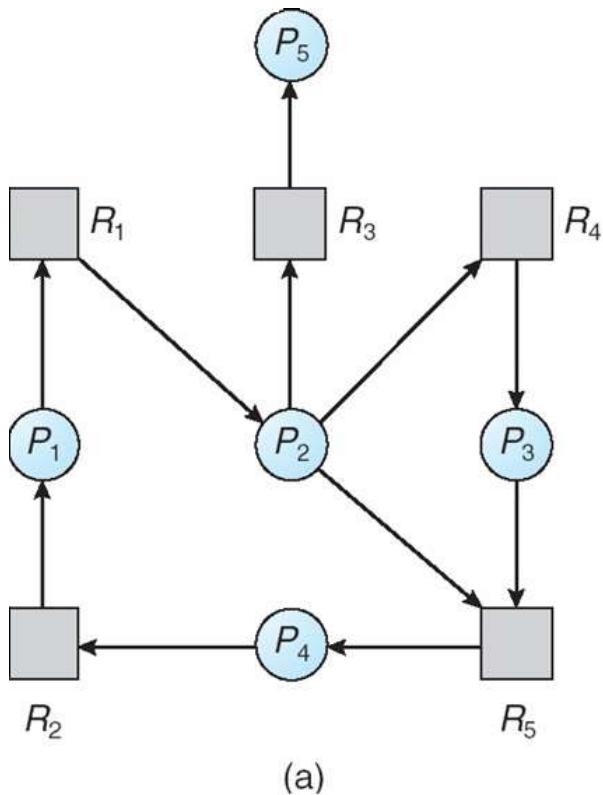
# Deadlock Detection

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

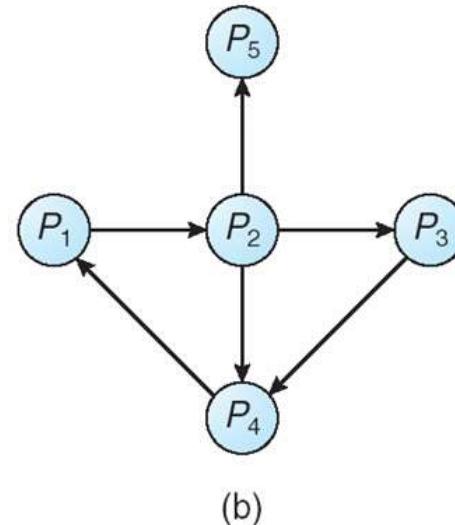
# Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes
  - $P_i \rightarrow P_j$  if  $P_i$  is waiting for  $P_j$
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of  $n^2$  operations, where  $n$  is the number of vertices in the graph

# Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

# Several Instances of a Resource Type

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

# Detection Algorithm

1. Let  $Work$  and  $Finish$  be vectors of length  $m$  and  $n$ , respectively Initialize:
  - (a)  $Work = Available$
  - (b) For  $i = 1, 2, \dots, n$ , if  $Allocation_i \neq 0$ , then  
 $Finish[i] = \text{false}$ ; otherwise,  $Finish[i] = \text{true}$
2. Find an index  $i$  such that both:
  - (a)  $Finish[i] == \text{false}$
  - (b)  $Request_i \leq Work$If no such  $i$  exists, go to step 4

# Detection Algorithm (Cont.)

3.  $Work = Work + Allocation_i$ ,  
 $Finish[i] = true$   
go to step 2
4. If  $Finish[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then  
the system is in deadlock state. Moreover, if  
 $Finish[i] == \text{false}$ , then  $P_i$  is deadlocked

# Example of Detection Algorithm

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

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	0	0	0	0	0	0
$P_1$	2	0	0	2	0	2			
$P_2$	3	0	3	0	0	0			
$P_3$	2	1	1	1	0	0			
$P_4$	0	0	2	0	0	2			

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

# Example (Cont.)

- $P_2$  requests an additional instance of type C

*Request*

	A	B	C
$P_0$	0	0	0
$P_1$	2	0	1
$P_2$	0	0	1
$P_3$	1	0	0
$P_4$	0	0	2

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

# Detection-Algorithm Usage

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

# Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - 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
  - Is process interactive or batch?

# Recovery from Deadlock: 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

# Advantages and Disadvantages

**Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]**

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	<ul style="list-style-type: none"> <li>• Works well for processes that perform a single burst of activity</li> <li>• No preemption necessary</li> </ul>	<ul style="list-style-type: none"> <li>• Inefficient</li> <li>• Delays process initiation</li> <li>• Future resource requirements must be known by processes</li> </ul>
		Preemption	<ul style="list-style-type: none"> <li>• Convenient when applied to resources whose state can be saved and restored easily</li> </ul>	<ul style="list-style-type: none"> <li>• Preempts more often than necessary</li> </ul>
		Resource ordering	<ul style="list-style-type: none"> <li>• Feasible to enforce via compile-time checks</li> <li>• Needs no run-time computation since problem is solved in system design</li> </ul>	<ul style="list-style-type: none"> <li>• Disallows incremental resource requests</li> </ul>
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	<ul style="list-style-type: none"> <li>• No preemption necessary</li> </ul>	<ul style="list-style-type: none"> <li>• Future resource requirements must be known by OS</li> <li>• Processes can be blocked for long periods</li> </ul>
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	<ul style="list-style-type: none"> <li>• Never delays process initiation</li> <li>• Facilitates online handling</li> </ul>	<ul style="list-style-type: none"> <li>• Inherent preemption losses</li> </ul>

# **Unit-5**

# **Memory Management**

# Outline

1. Basic requirements of Memory Management
2. Memory Partitioning
3. Basic blocks of Memory Management
  1. Paging
  2. Segmentation

# The need for memory management

- Memory is cheap today, and getting cheaper
  - But applications are demanding more and more memory, there is never end to it.
- Memory Management, involves swapping blocks of data from secondary storage.
- Memory I/O is slow compared to a CPU
  - The OS must cleverly time the swapping to maximise the CPU's efficiency

# Memory Management

*Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time*

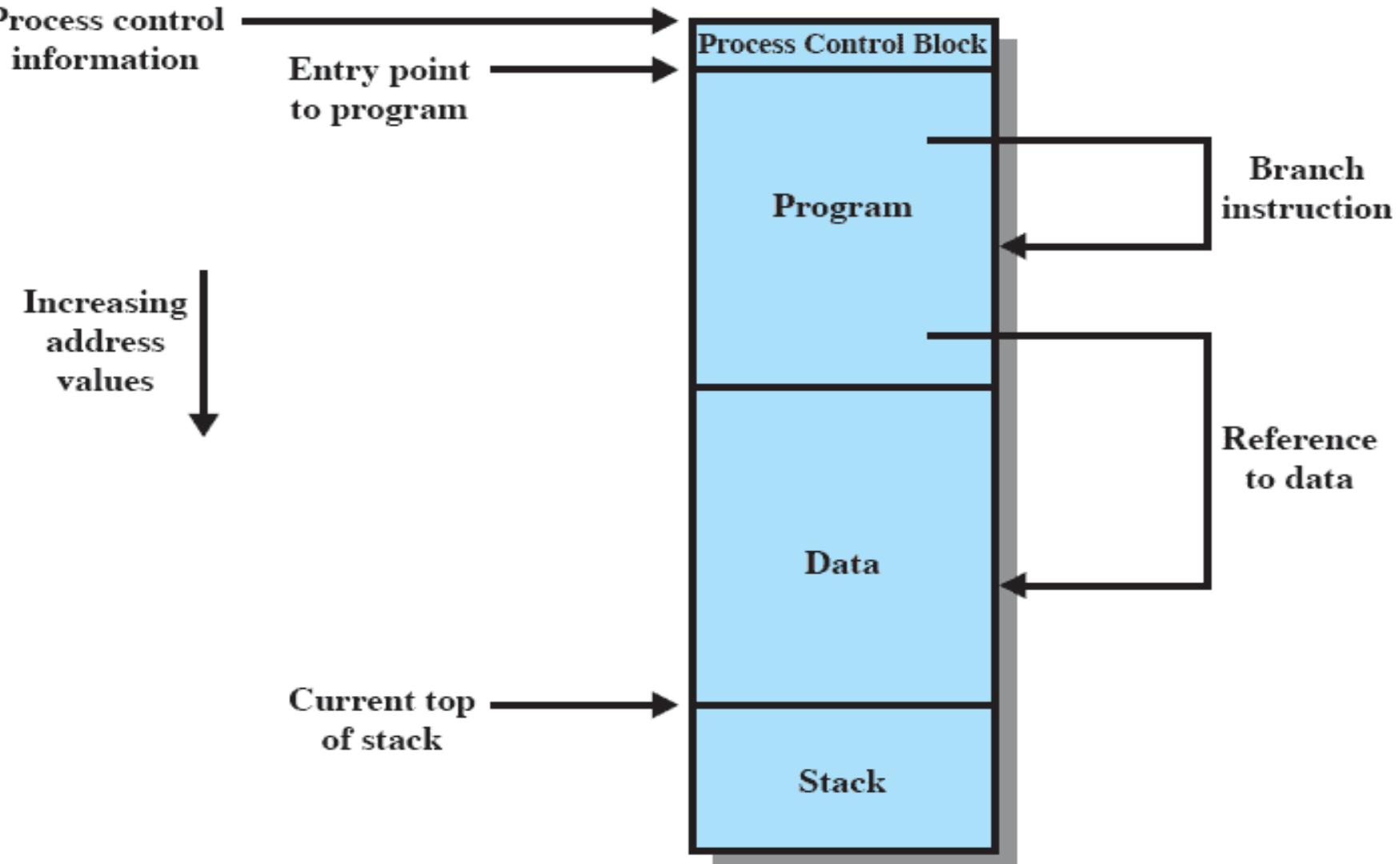
# Memory Management Requirements

- Relocation
- Protection
- Sharing
- Logical organisation
- Physical organisation

# Requirements: Relocation

- The programmer does not know where the program will be placed in memory when it is executed,
  - it may be swapped to disk and return to main memory at a different location (relocated)
- Memory references must be translated to the actual physical memory address

# Addressing



# Requirements: Protection

- Processes should not be able to reference memory locations in another process without permission
- Impossible to check absolute addresses at compile time
- Must be checked at run time

# Requirements: Sharing

- Allow several processes to access the same portion of memory
- Better to allow each process access to the same copy of the program rather than have their own separate copy

# Requirements: Logical Organization

- Memory is organized linearly (usually)
- Programs are written in modules
  - Modules can be written and compiled independently
- Different degrees of protection given to modules (read-only, execute-only)
- Share modules among processes
- Segmentation helps here

# Requirements: Physical Organization

- Cannot leave the programmer with the responsibility to manage memory
- Memory available for a program plus its data may be insufficient
  - **Overlaying (Overlapping)** allows various modules to be assigned the same region of memory but is time consuming to program
- Programmer does not know how much space will be available

## 2.

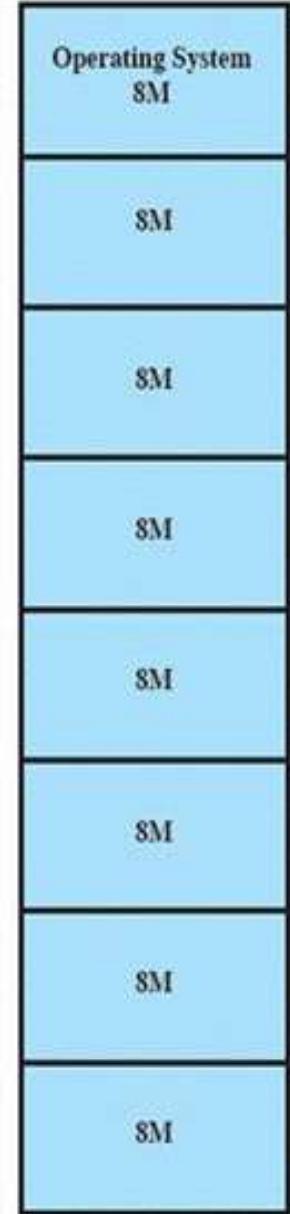
# Memory Partitioning

- A. Fixed Partitioning
- B. Dynamic Partitioning
- C. Simple Paging
- D. Simple Segmentation
- E. Virtual Memory Paging
- F. Virtual Memory Segmentation

A.

# Fixed Partitioning

- Equal-size partitions Any process whose size is less than or equal to the partition size can be loaded into an available partition
- The operating system can swap a process out of a partition
  - If none are in a ready or running state



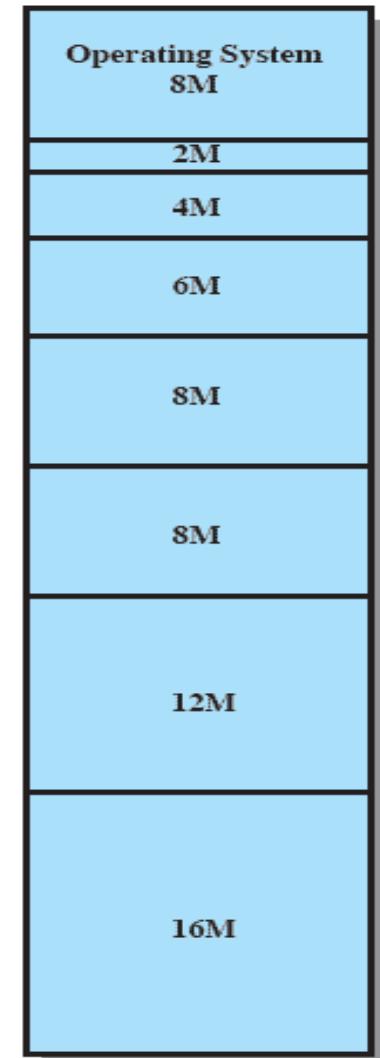
(a) Equal-size partitions

# Fixed Partitioning Problems

- A program may not fit in a partition.
  - The programmer must design the program with overlays
- Main memory use is inefficient.
  - Any program, no matter how small, occupies an entire partition.
  - This results in *internal fragmentation*.

# Solution – Unequal Size Partitions

- But doesn't solve completely
- Programs up to 16M can be accommodated without overlay
- Smaller programs can be placed in smaller partitions, reducing internal fragmentation



(b) Unequal-size partitions

# Placement Algorithm

- Equal-size
  - Placement is trivial (no options)
- Unequal-size
  - Can assign each process to the smallest partition within which it will fit
  - Queue for each partition
  - Processes are assigned in such a way as to minimize wasted memory within a partition

# Fixed Partitioning

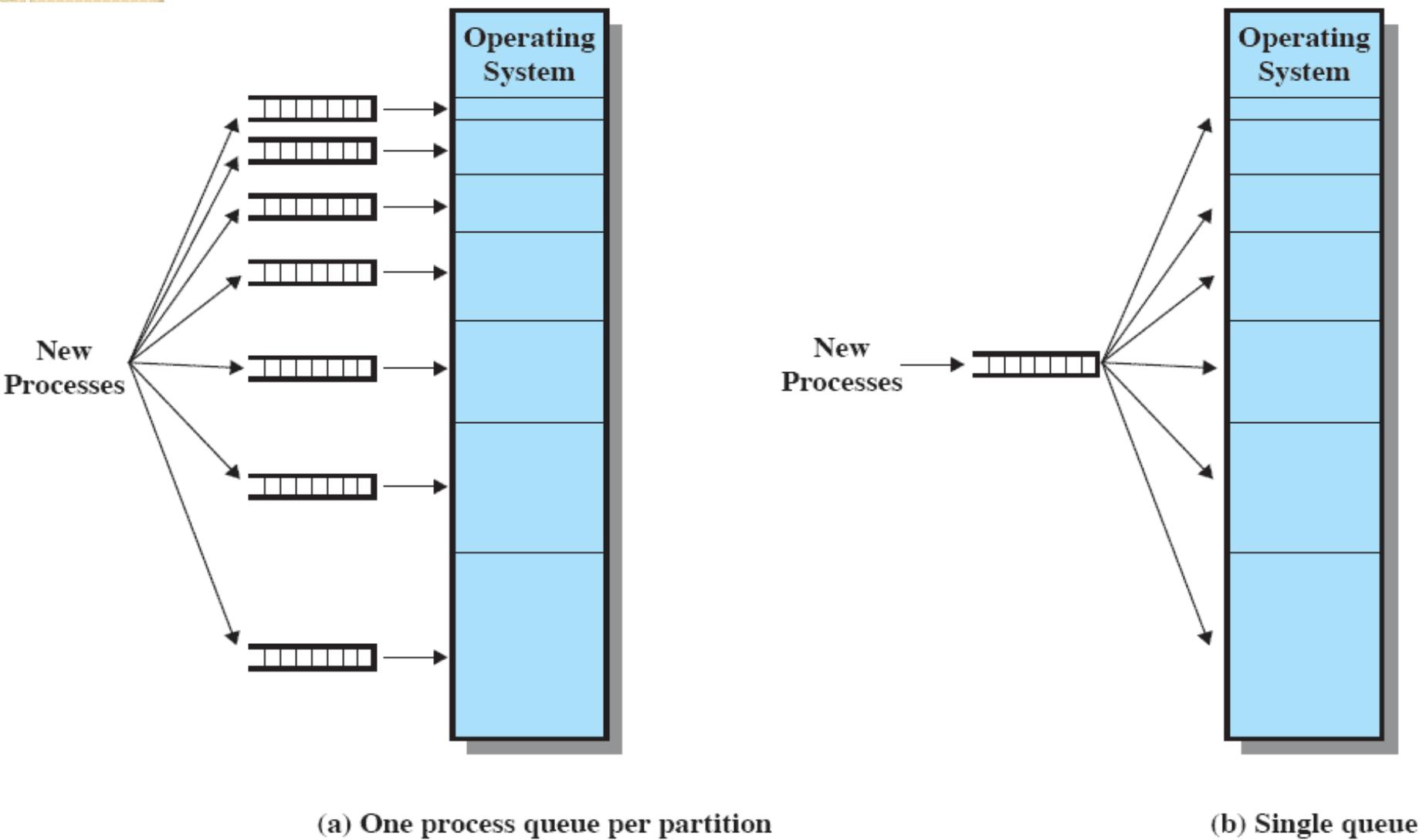


Figure 7.3 Memory Assignment for Fixed Partitioning

# Remaining Problems with Fixed Partitions

- The number of active processes is limited by the system
  - limited by the pre-determined number of partitions
- A large number of very small process will not use the space efficiently
  - In either fixed or variable length partition methods

B.

## Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required

# Dynamic Partitioning Example



- ***External Fragmentation***
- Memory external to all processes is fragmented
- Can resolve using ***compaction***
  - OS moves processes so that they are contiguous
  - Time consuming and wastes CPU time

# Dynamic Partitioning

- Operating system must decide which free block to allocate to a process
- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Since smallest block is found for process, the smallest amount of fragmentation is left
  - Memory compaction must be done more often
  - Worst performer overall

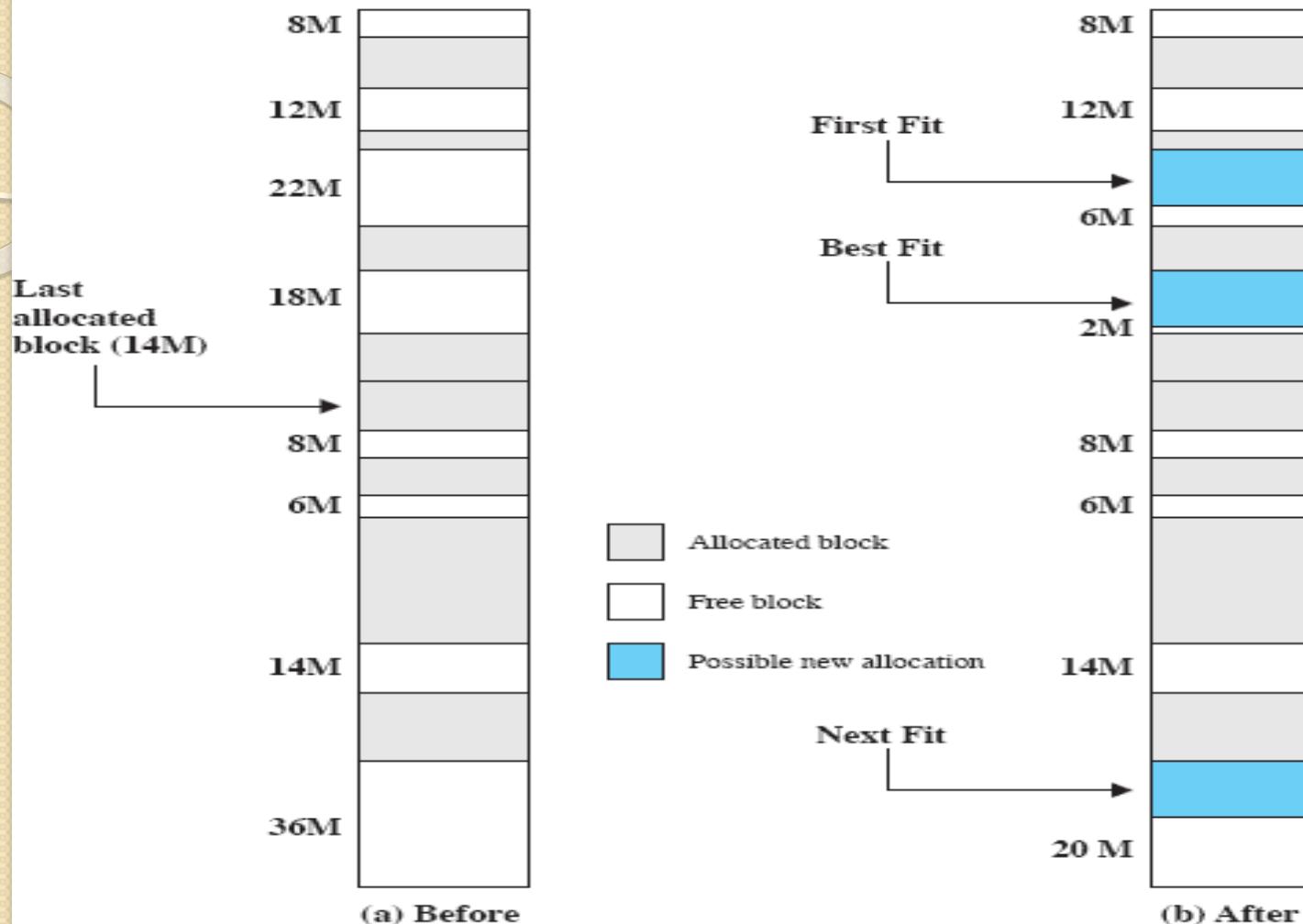
# Dynamic Partitioning

- First-fit algorithm
  - Scans memory from the beginning and chooses the first available block that is large enough
  - Fastest
  - May have many processes loaded in the front end of memory that must be searched over when trying to find a free block

# Dynamic Partitioning

- Next-fit
  - Scans memory from the location of the last placement
  - More often allocate a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory

# Allocation



**Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block**

# Buddy System

- Entire space available is treated as a single block of  $2^N$
- If a request of size  $s$  where  $2^{N-1} < s \leq 2^N$ 
  - entire block is allocated
- Otherwise block is split into two equal buddies
  - Process continues until smallest block greater than or equal to  $s$  is generated

# Example of Buddy System

1 Mbyte block	1 M					
Request 100 K	A = 128K	128K	256K	512K		
Request 240 K	A = 128K	128K	B = 256K	512K		
Request 64 K	A = 128K	C = 64K	64K	B = 256K	512K	
Request 256 K	A = 128K	C = 64K	64K	B = 256K	D = 256K	256K
Release B	A = 128K	C = 64K	64K	256K	D = 256K	256K
Release A	128K	C = 64K	64K	256K	D = 256K	256K
Request 75 K	E = 128K	C = 64K	64K	256K	D = 256K	256K
Release C	E = 128K	128K	256K	D = 256K	256K	
Release E		512K	D = 256K	256K		
Release D			1M			

Figure 7.6 Example of Buddy System

# Tree Representation of Buddy System

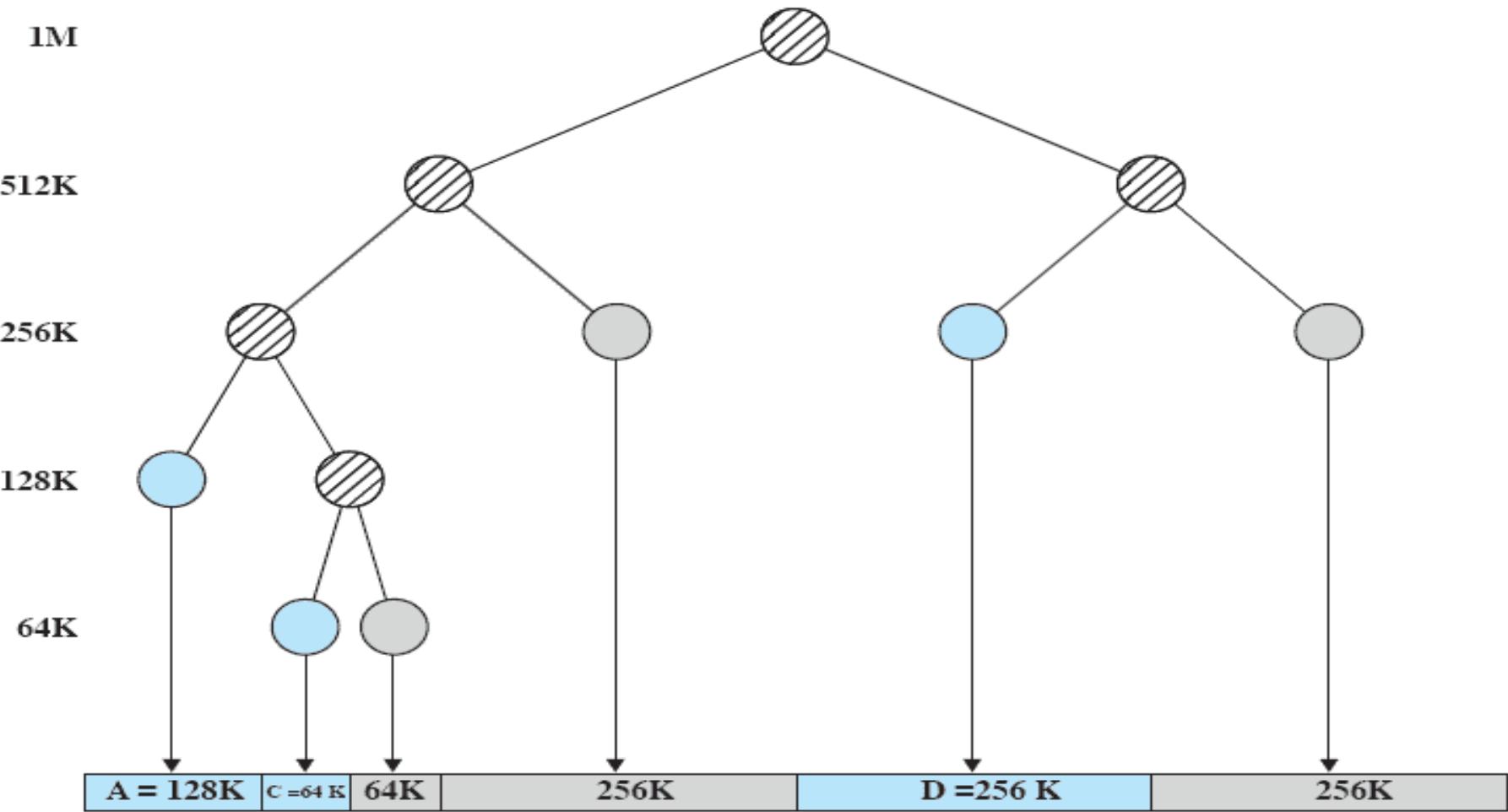


Figure 7.7 Tree Representation of Buddy System

# Relocation

- When program loaded into memory the actual (absolute) memory locations are determined
- A process may occupy different partitions which means different absolute memory locations during execution
  - Swapping
  - Compaction

# Addresses

- Logical
  - Reference to a memory location independent of the current assignment of data to memory.
- Relative
  - Address expressed as a location relative to some known point.
- Physical or Absolute
  - The absolute address or actual location in main memory.

# Relocation

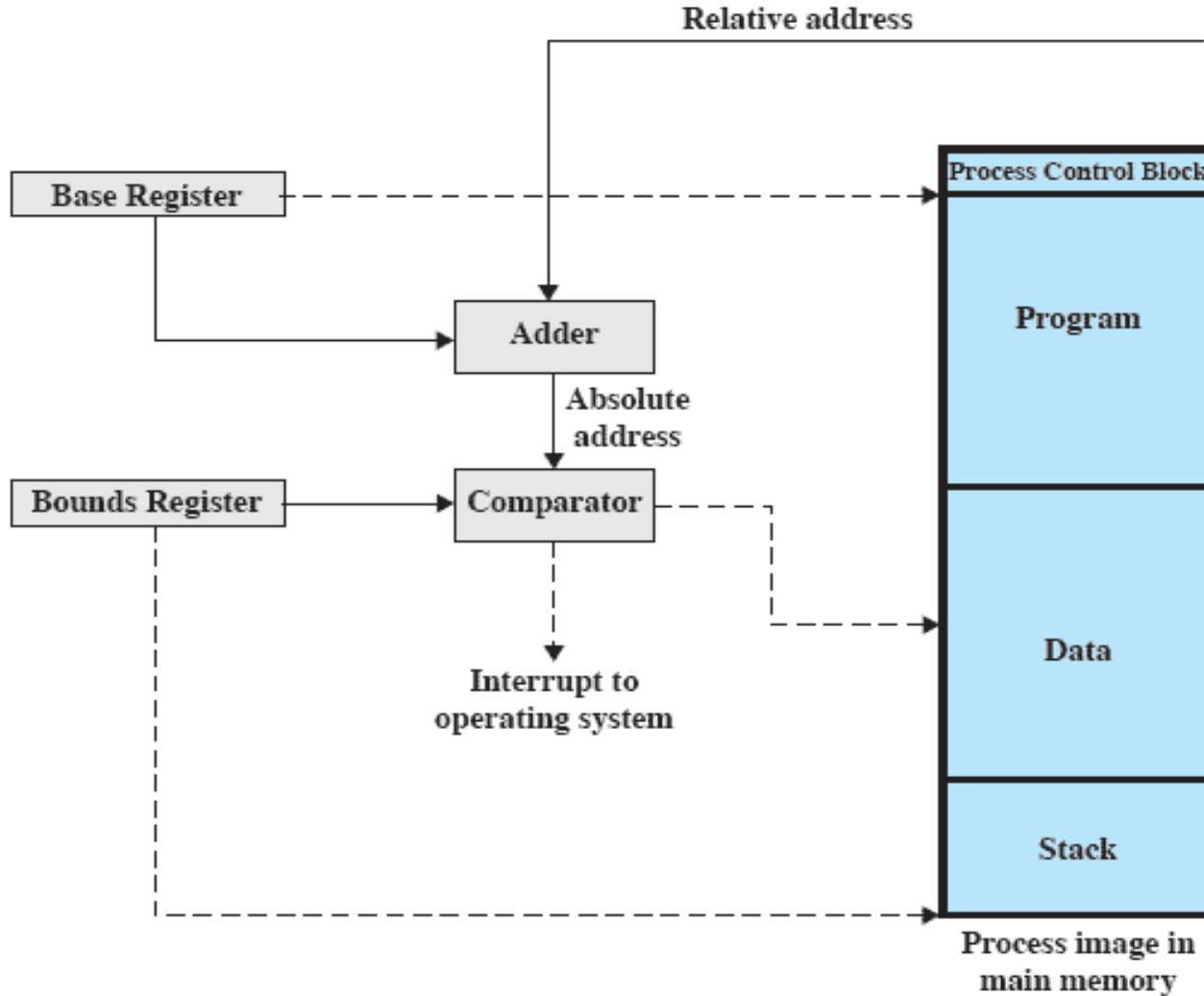


Figure 7.8 Hardware Support for Relocation

# Registers Used during Execution

- Base register
  - Starting address for the process
- Bounds register
  - Ending location of the process
- These values are set when the process is loaded or when the process is swapped in

# Registers Used during Execution

- The value of the base register is added to a relative address to produce an absolute address
- The resulting address is compared with the value in the bounds register
- If the address is not within bounds, an interrupt is generated to the operating system

# Paging

- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks
- The chunks of a process are called *pages*
- The chunks of memory are called *frames*

# Paging

- Operating system maintains a page table for each process
  - Contains the **frame** location for each page in the process
  - Memory address consist of a page number and offset within the page

# Processes and Frames

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

# Page Table

0	0
1	1
2	2
3	3

Process A  
page table

0	—
1	—
2	—

Process B  
page table

0	7
1	8
2	9
3	10

Process C  
page table

0	4
1	5
2	6
3	11
4	12

Process D  
page table

13
14

Free frame  
list

Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

# Segmentation

- A program can be subdivided into segments
  - Segments may vary in length
  - There is a maximum segment length
- Addressing consists of two parts
  - a segment number and
  - an offset
- Segmentation is similar to dynamic partitioning

# Logical Addresses

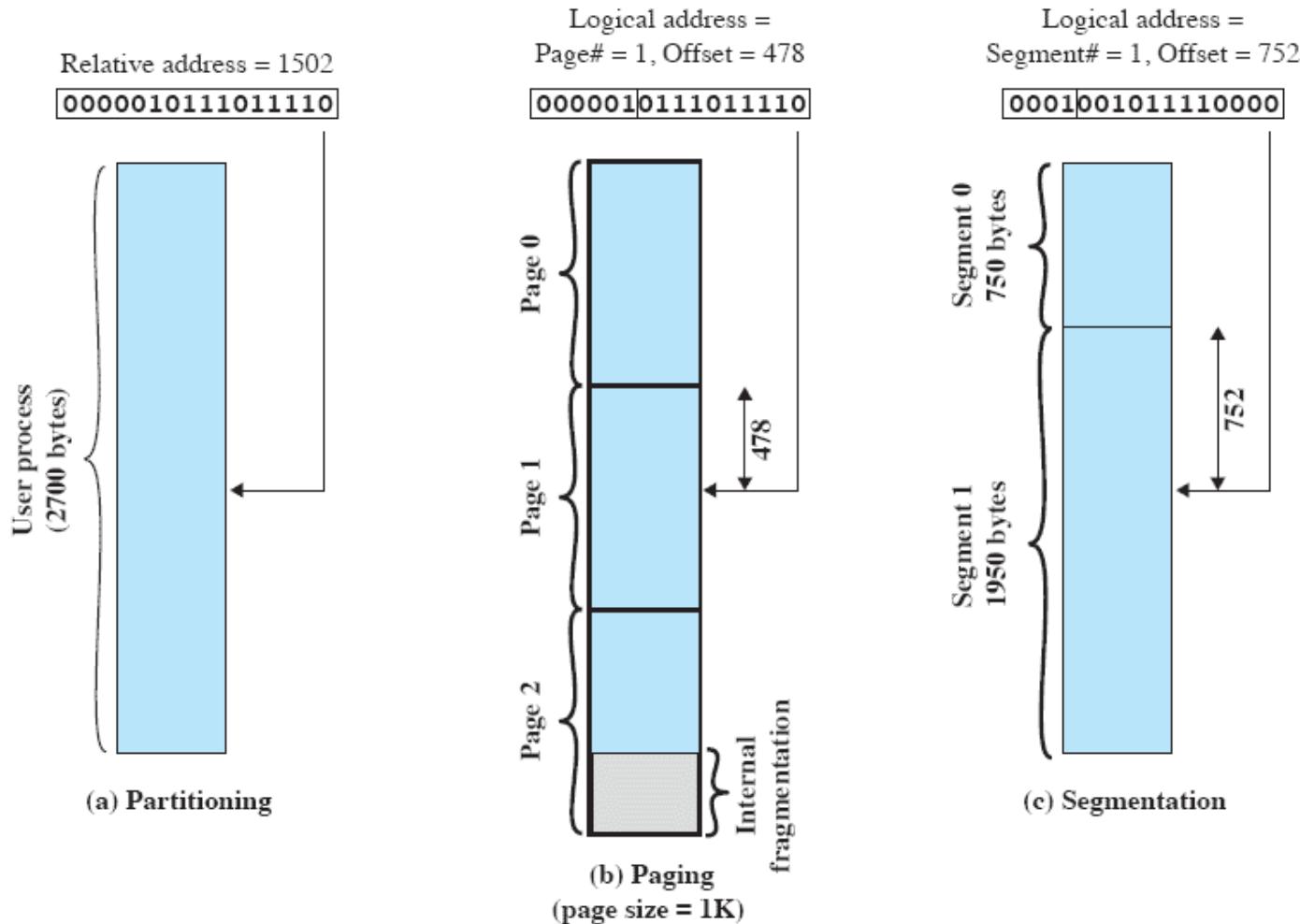
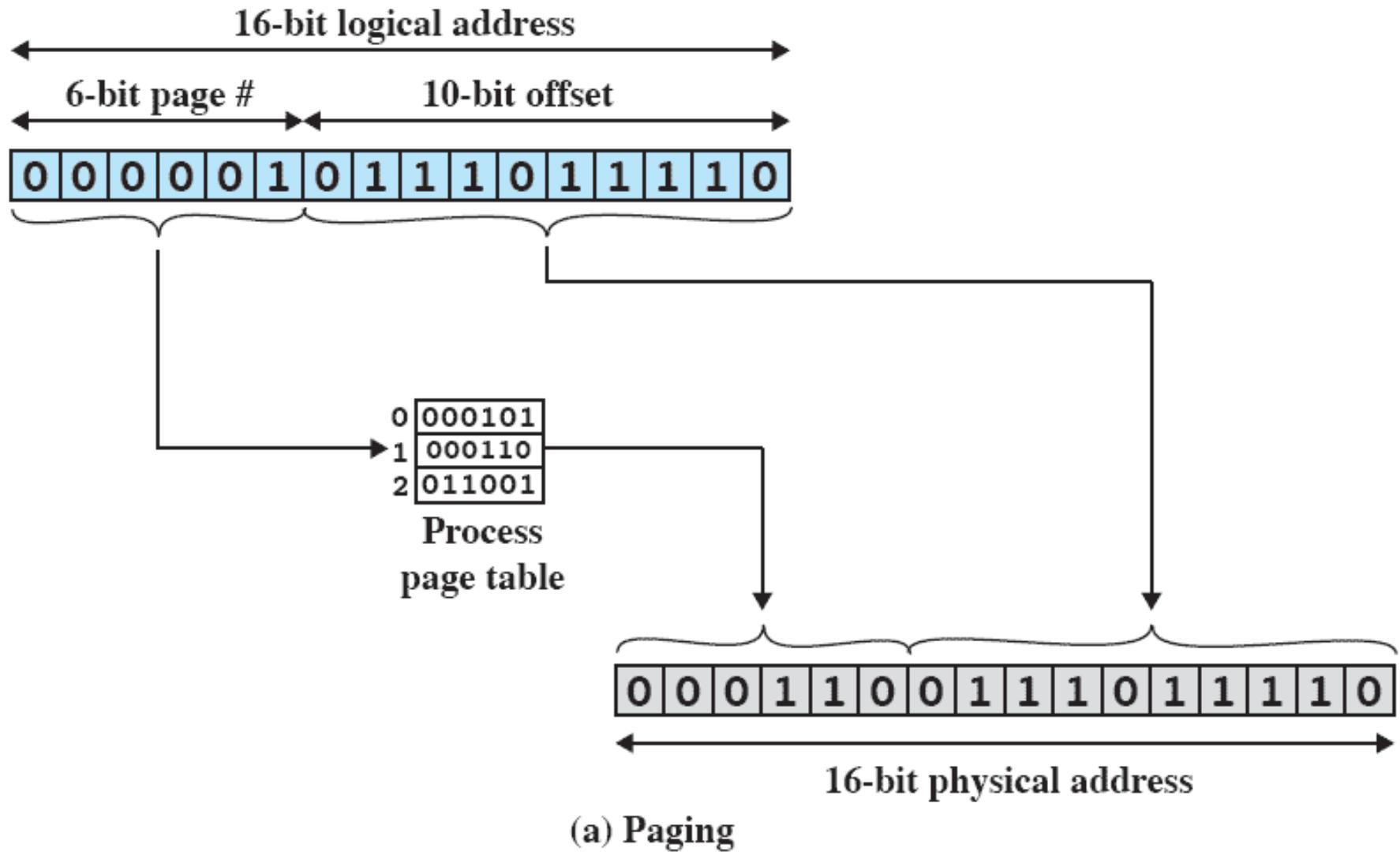


Figure 7.11 Logical Addresses

# Paging



# Segmentation

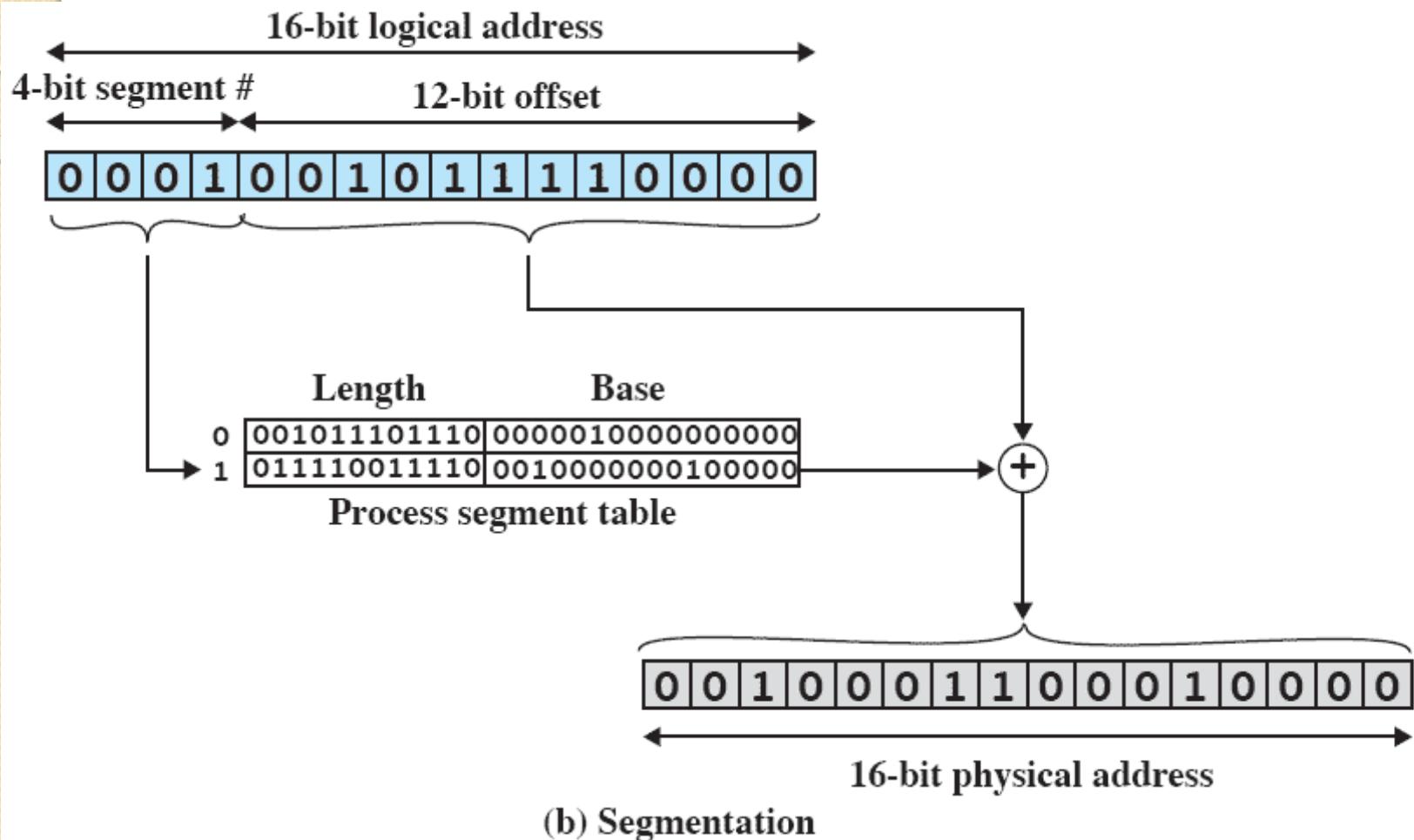


Figure 7.12 Examples of Logical-to-Physical Address Translation

# Virtual Memory Management and Page Replacement Algorithms

**Table 8.1** Virtual Memory Terminology

<b>Virtual memory</b>	A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.
<b>Virtual address</b>	The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory.
<b>Virtual address space</b>	The virtual storage assigned to a process.
<b>Address space</b>	The range of memory addresses available to a process.
<b>Real address</b>	The address of a storage location in main memory.

# Key points in Memory Management

- 1) Memory references are logical addresses dynamically translated into physical addresses at run time
  - A process may be swapped in and out of main memory occupying different regions at different times during execution
- 2) A process may be broken up into pieces that do not need to located contiguously in main memory

# Breakthrough in Memory Management

- If **both** of those two characteristics are present,
  - then it is not necessary that all of the pages or all of the segments of a process be in main memory during execution.
- If the next instruction, and the next data location are in memory then execution can proceed
  - at least for a time

# Execution of a Process

- Operating system brings into main memory a few pieces of the program
- Resident set - portion of process that is in main memory
- An interrupt is generated when an address is needed that is not in main memory
- Operating system places the process in a blocking state

# Execution of a Process

- Piece of process that contains the logical address is brought into main memory
  - Operating system issues a disk I/O Read request
  - Another process is dispatched to run while the disk I/O takes place
  - An interrupt is issued when disk I/O complete which causes the operating system to place the affected process in the Ready state

# Implications of this new strategy

- More processes may be maintained in main memory
  - Only load in some of the pieces of each process
  - With so many processes in main memory, it is very likely a process will be in the Ready state at any particular time
- A process may be larger than all of main memory

# Real and Virtual Memory

- Real memory
  - Main memory, the actual RAM
- Virtual memory
  - Memory on disk
  - Allows for effective multiprogramming and relieves the user of tight constraints of main memory

# Support Needed for Virtual Memory

- Hardware must support paging and segmentation
- Operating system must be able to manage the movement of pages and/or segments between secondary memory and main memory

# Paging

- Each process has its own page table
- Each page table entry contains the frame number of the corresponding page in main memory
- Two extra bits are needed to indicate:
  - whether the page is in main memory or not
  - Whether the contents of the page has been altered since it was last loaded

# Paging Table

Virtual Address



Page Table Entry



(a) Paging only

# Address Translation

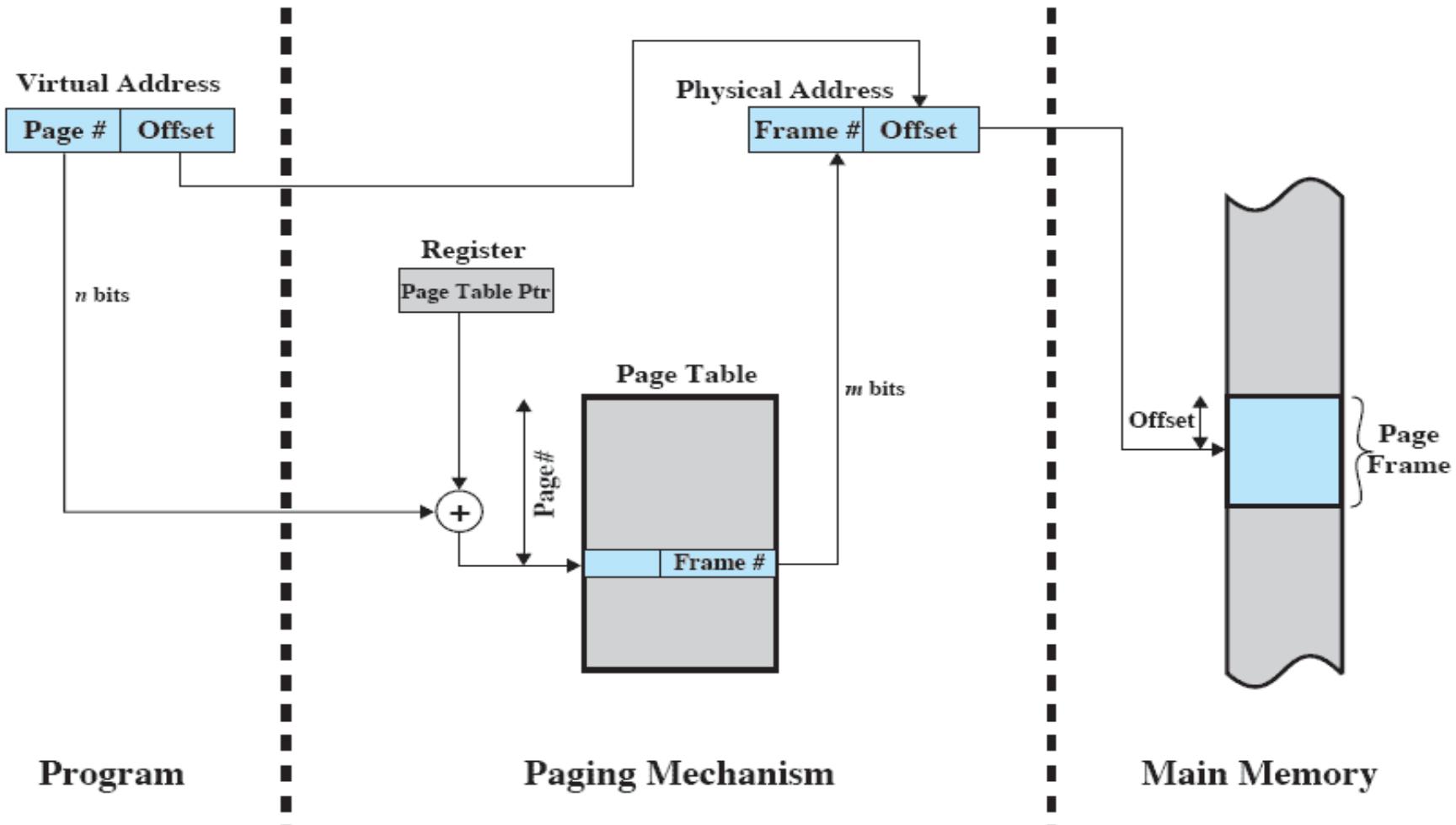
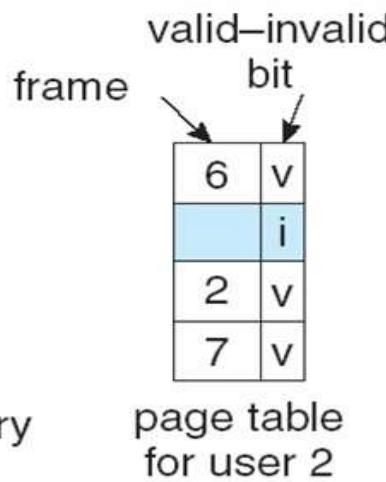
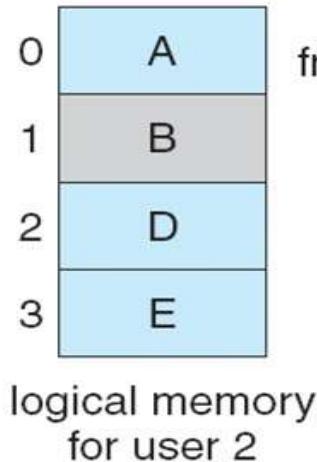
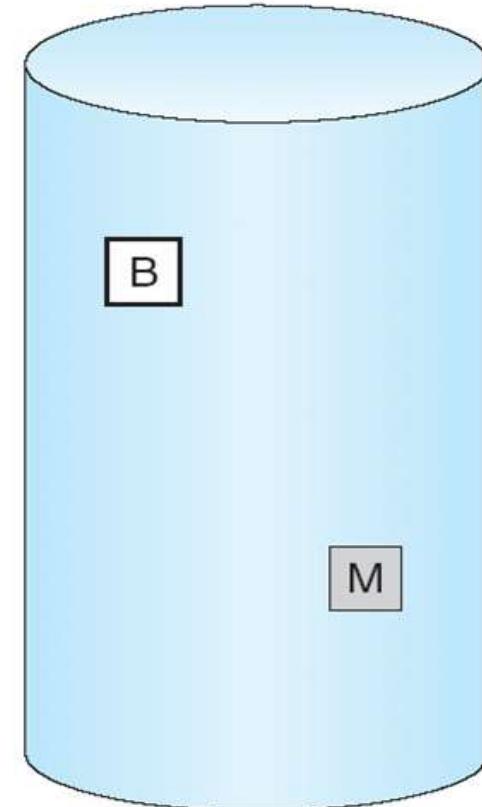
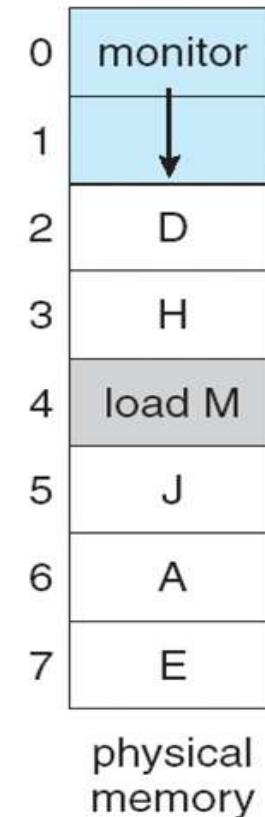
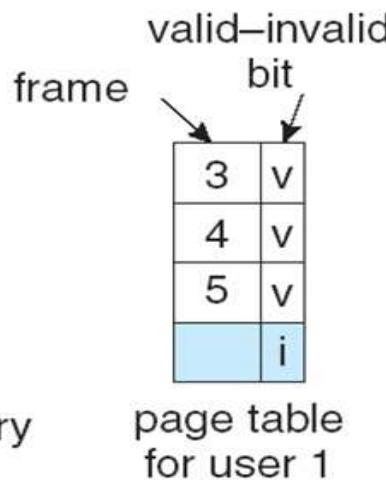
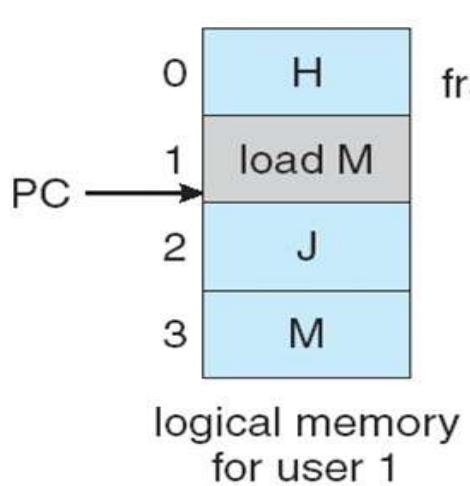


Figure 8.3 Address Translation in a Paging System

# Replacement Policy

- When all of the frames in main memory are occupied and it is necessary to bring in a new page, the replacement policy determines which page currently in memory is to be replaced.

# Need For Page Replacement



# Basic Replacement Algorithms

- There are certain basic algorithms that are used for the selection of a page to replace, they include
  - Optimal
  - Least recently used (LRU)
  - First-in-first-out (FIFO)
- Examples

# Examples

- An example of the implementation of these policies will use a page address stream formed by executing the program is
  - 2 3 2 1 5 2 4 5 3 2 5 2
- Which means that the first page referenced is 2,
  - the second page referenced is 3,
  - And so on.

# Optimal policy

- Selects for replacement that page for which the time to the next reference is the longest
- But Impossible to have perfect knowledge of future events

# Optimal Policy Example

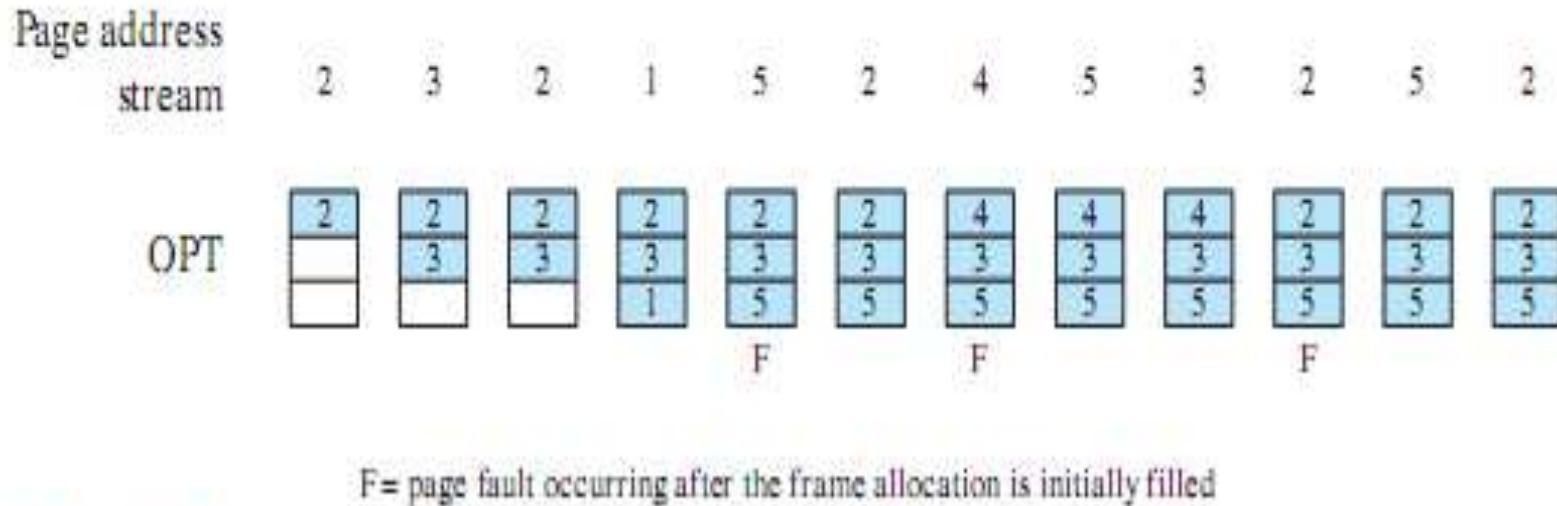


Figure 8.15 Behavior of Four Page Replacement Algorithms

- The optimal policy produces three page faults after the frame allocation has been filled.

# Least Recently Used (LRU)

- Replaces the page that has not been referenced for the longest time
- By the principle of locality, this should be the page least likely to be referenced in the near future
- Difficult to implement
  - One approach is to tag each page with the time of last reference.
  - This requires a great deal of overhead.

# LRU Example

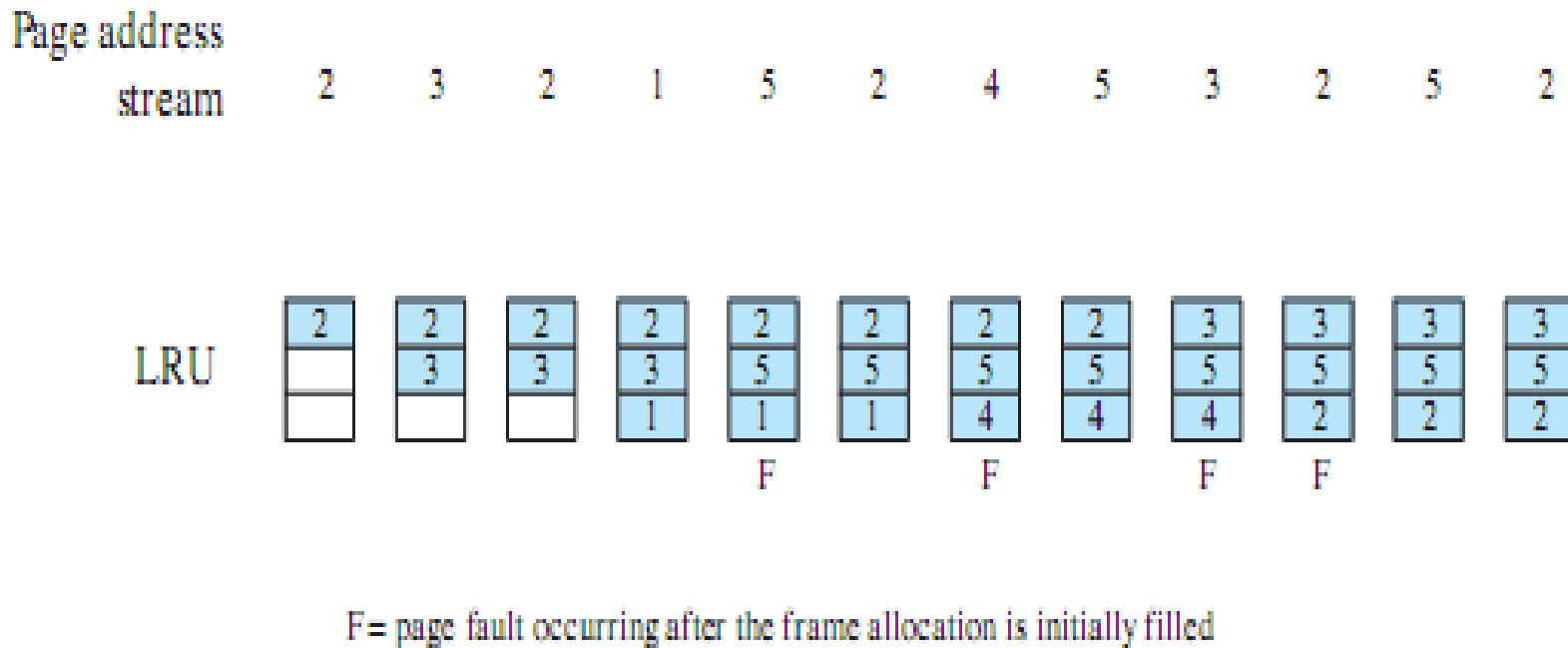


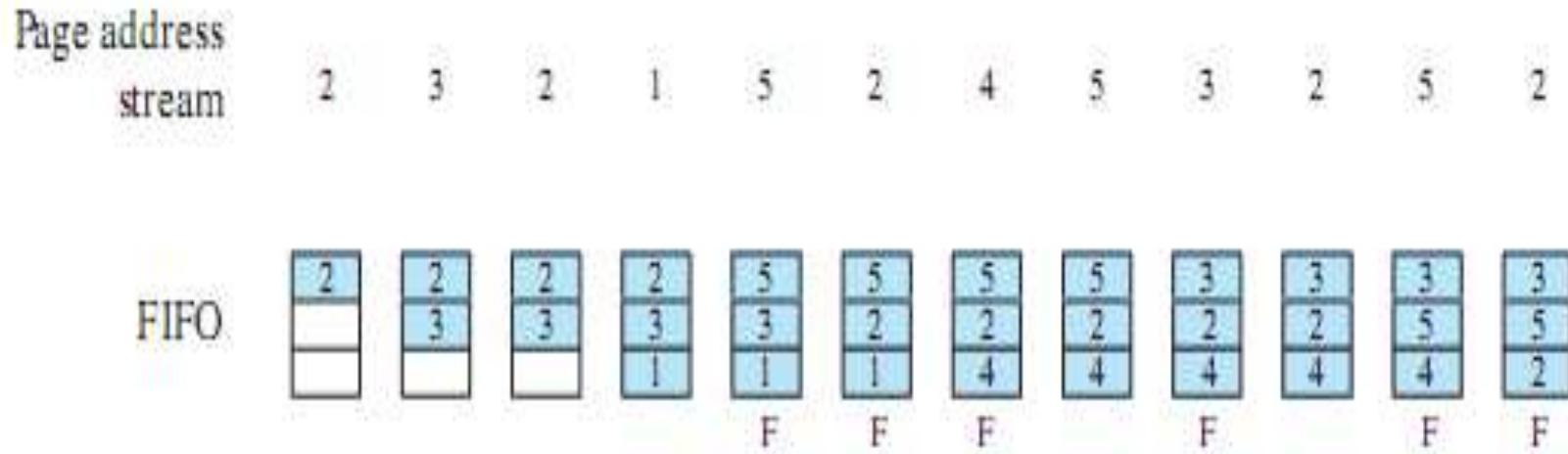
Figure 8.15 Behavior of Four Page Replacement Algorithms

- The LRU policy does nearly as well as the optimal policy.
  - In this example, there are four page faults

# First-in, first-out (FIFO)

- Treats page frames allocated to a process as a circular buffer
- Pages are removed in round-robin style
  - Simplest replacement policy to implement
- Page that has been in memory the longest is replaced
  - But, these pages may be needed again very soon if it hasn't truly fallen out of use

# FIFO Example



F = page fault occurring after the frame allocation is initially filled

Figure 8.15 Behavior of Four Page Replacement Algorithms

- The FIFO policy results in six page faults.
  - Note that LRU recognizes that pages 2 and 5 are referenced more frequently than other pages, whereas FIFO does not.

# Clock Policy

- Uses an additional bit called a “use bit”
- When a page is first loaded in memory or referenced, the use bit is set to 1
- When it is time to replace a page, the OS scans the set flipping all 1's to 0
- The first frame encountered with the use bit already set to 0 is replaced.

# Clock Policy Example

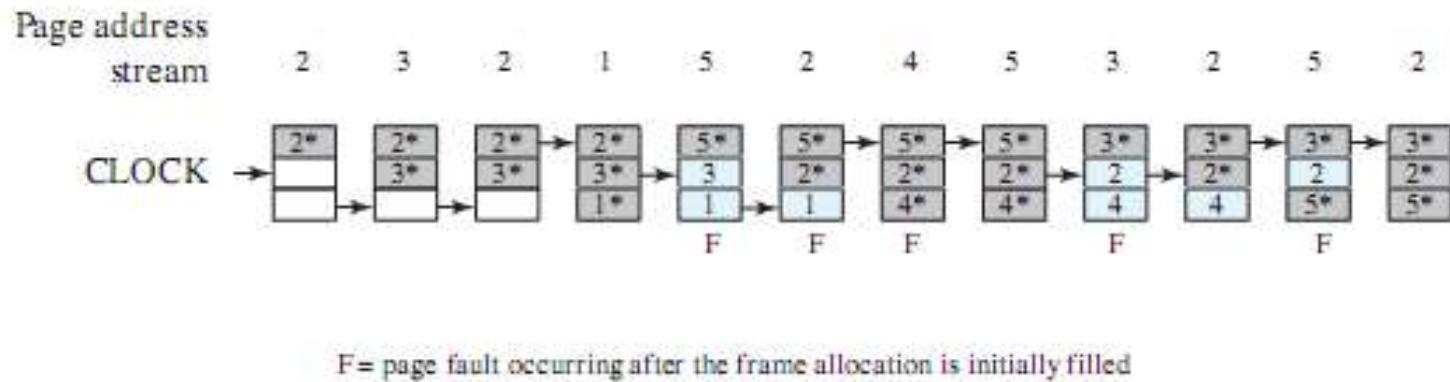


Figure 8.15 Behavior of Four Page Replacement Algorithms

- Note that the clock policy is adept at protecting frames 2 and 5 from replacement.

# Combined Examples

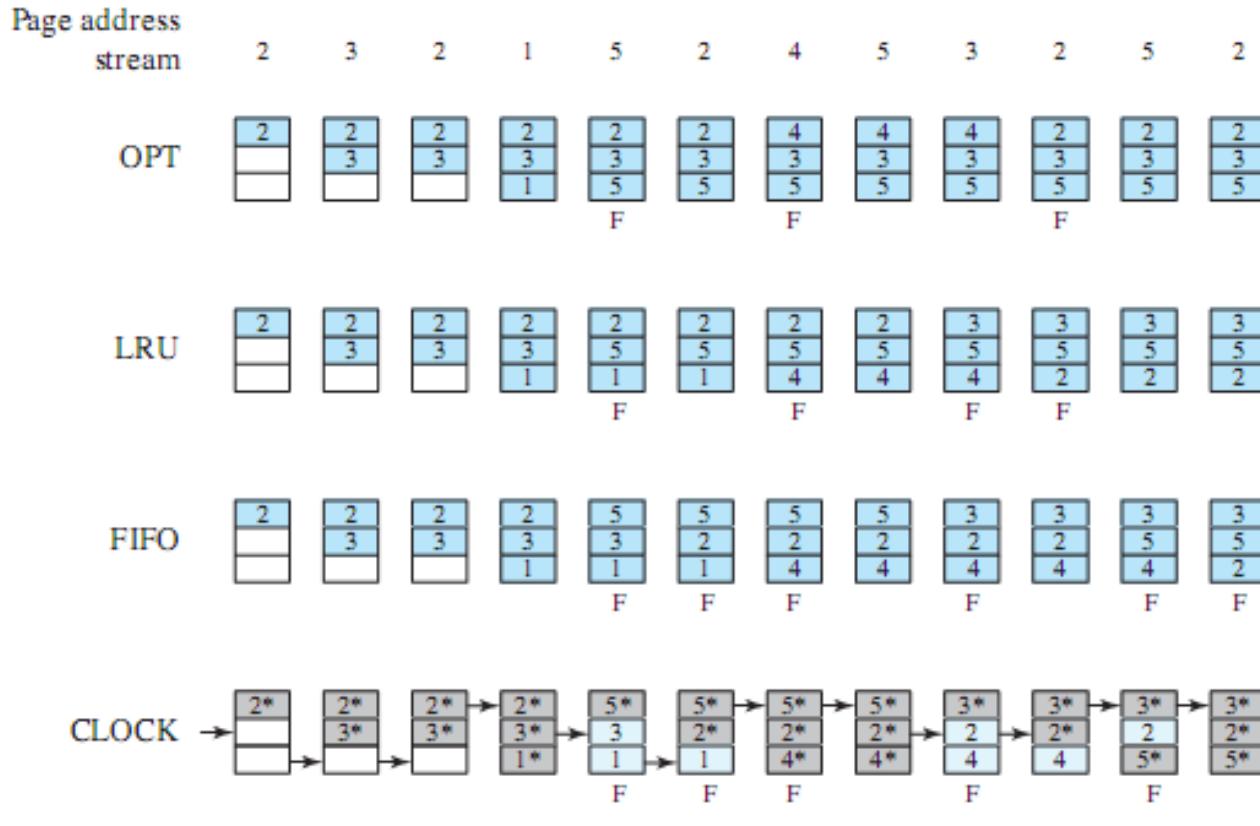


Figure 8.15 Behavior of Four Page Replacement Algorithms

# Comparison

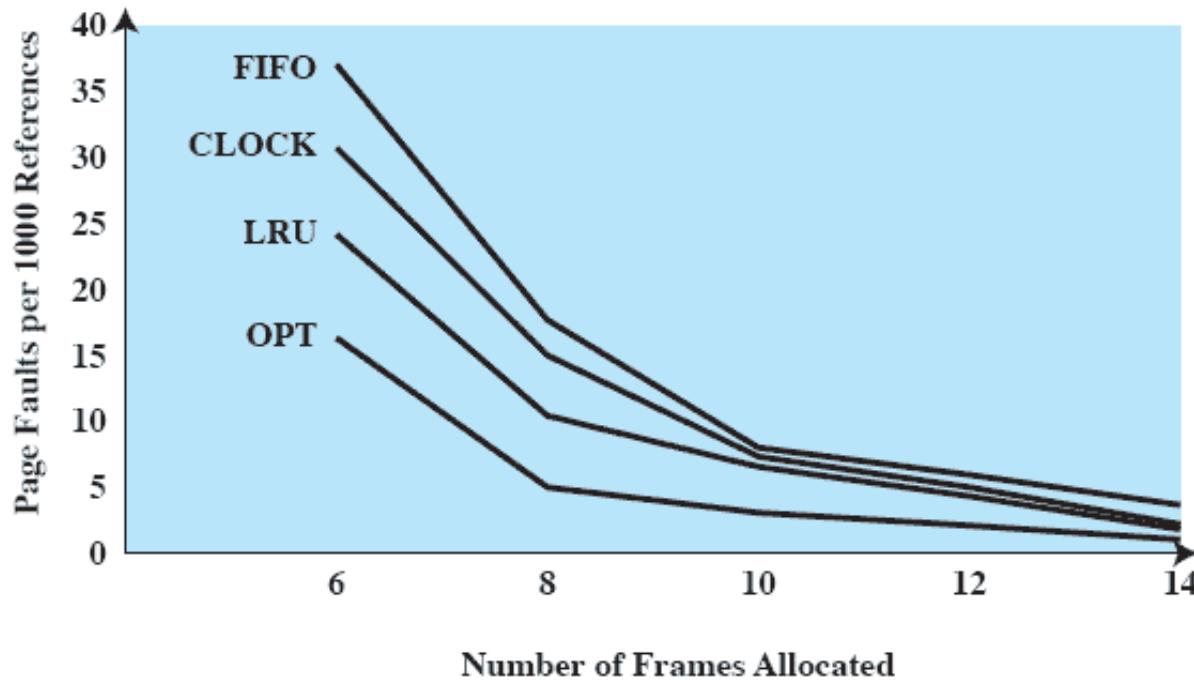
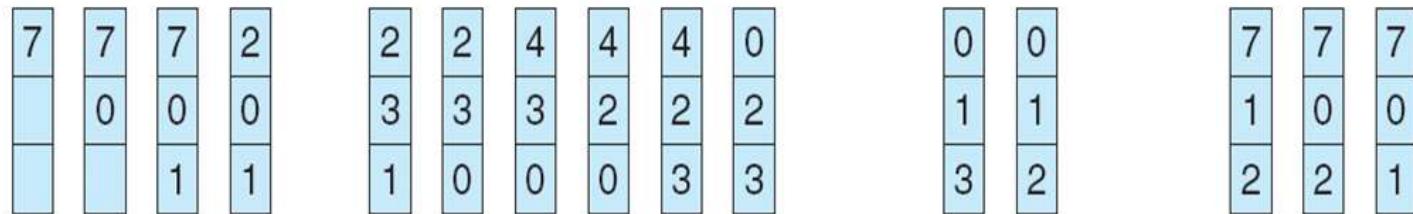


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

# FIFO Page Replacement

reference string

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

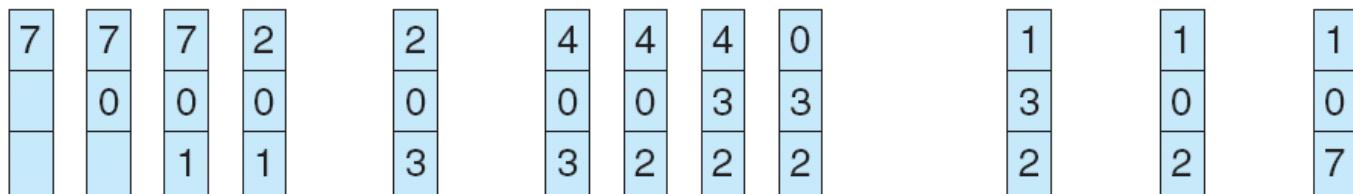


page frames

# LRU Page Replacement

reference string

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



page frames



# Unit-6, I/O Management and Disk Scheduling

# Outline

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- I/O Buffering
- Disk Scheduling

# Categories of I/O Devices

- Difficult area of OS design
  - Difficult to develop a consistent solution due to a wide variety of devices and applications
- Three Categories:
  - a. Human readable
  - b. Machine readable
  - c. Communications

# Human readable

- Devices used to communicate with the user
- Printers and terminals
  - Video display
  - Keyboard
  - Mouse etc

## I.b Machine readable

- Used to communicate with electronic equipment
  - Disk drives
  - USB keys
  - Sensors
  - Controllers
  - Actuators

# Communication

- Used to communicate with remote devices
  - Digital line drivers
  - Modems
  - I/O controllers

## 2.

# Differences in I/O Devices

- Devices differ in a number of areas
  - A. Data Rate
  - B. Application
  - C. Complexity of Control
  - D. Unit of Transfer
  - E. Data Representation
  - F. Error Conditions

## 2.A Data Rate

- May be massive difference between the data transfer rates of devices

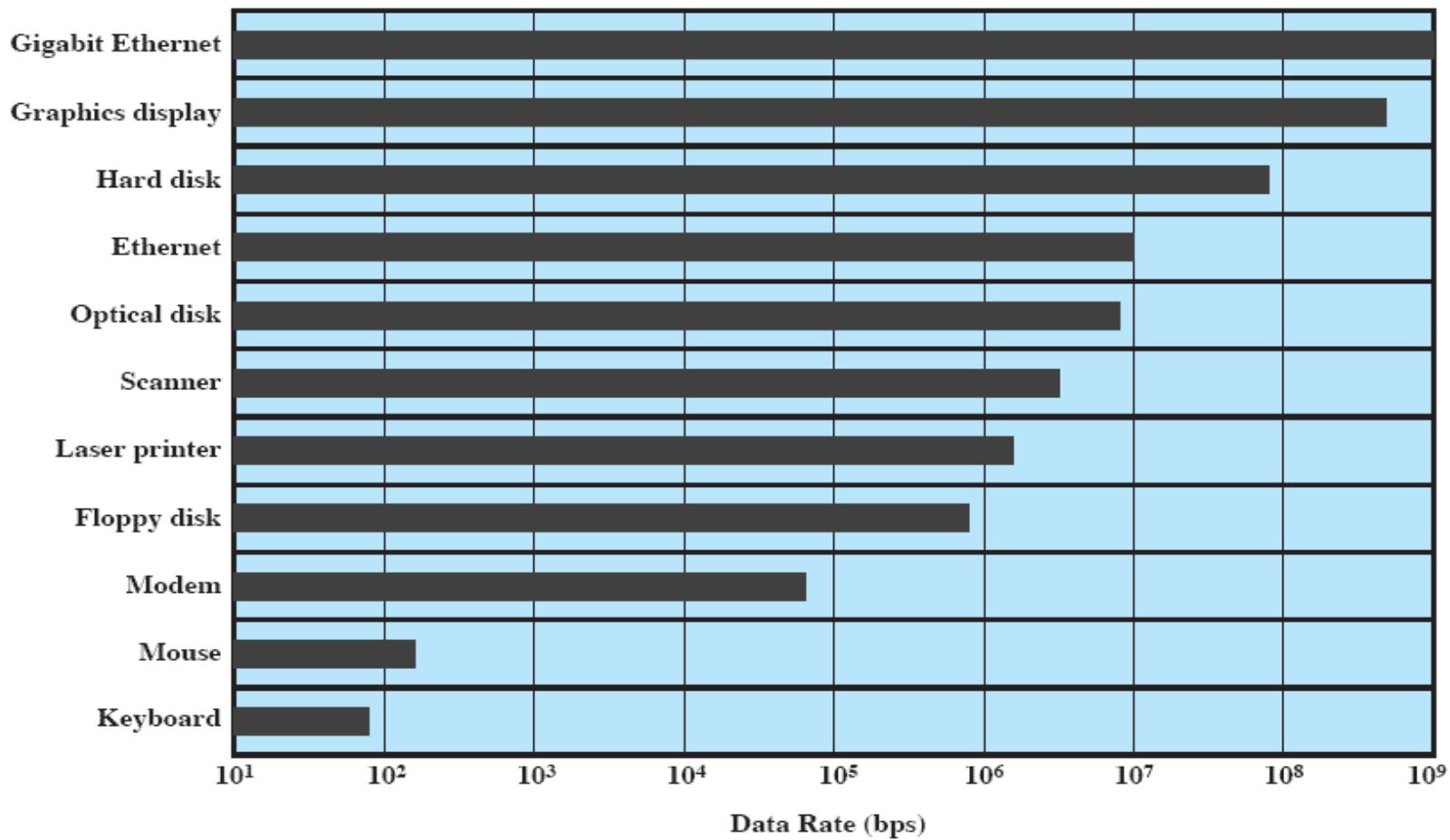


Figure 11.1 Typical I/O Device Data Rates

## 2.B Application

- Disk used to store files requires file management software
- Disk used to store virtual memory pages needs special hardware and software to support it
- Terminal used by system administrator may have a higher priority

## 2.C Complexity of control

- A printer requires a relatively simple control interface.
- A disk is much more complex.
- This complexity is filtered to some extent by the complexity of the I/O module that controls the device.

## 2.D Unit of transfer

- Data may be transferred as
  - a stream of bytes or characters (e.g., terminal I/O)
  - or in larger blocks (e.g., disk I/O).

## Data representation

- Different data encoding schemes are used by different devices,
  - including differences in character code and parity conventions.

## 2.F Error Conditions

- The nature of errors differ widely from one device to another.
- Aspects include:
  - the way in which they are reported,
  - their consequences,
  - the available range of responses

# Outline

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- I/O Buffering
- Disk Scheduling

# Techniques for performing I/O

- Programmed I/O
- Interrupt-driven I/O
- Direct memory access (DMA)

**Table 11.1 I/O Techniques**

	No Interrupts	Use of Interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

# Evolution of the I/O Function

1. Processor directly controls a peripheral device
2. Controller or I/O module is added
  - Processor uses programmed I/O without interrupts
  - Processor does not need to handle details of external devices

# Evolution of the I/O Function cont...

3. Controller or I/O module with interrupts
  - Efficiency improves as processor does not spend time waiting for an I/O operation to be performed
4. Direct Memory Access
  - Blocks of data are moved into memory without involving the processor
  - Processor involved at beginning and end only

# Evolution of the I/O Function cont...

5. I/O module is a separate processor
  - CPU directs the I/O processor to execute an I/O program in main memory.
6. I/O processor
  - I/O module has its own local memory
  - Commonly used to control communications with interactive terminals

## 2.

# Direct Memory Address

- Processor delegates I/O operation to the DMA module
- DMA module transfers data directly to or from memory
- When complete DMA module sends an interrupt signal to the processor

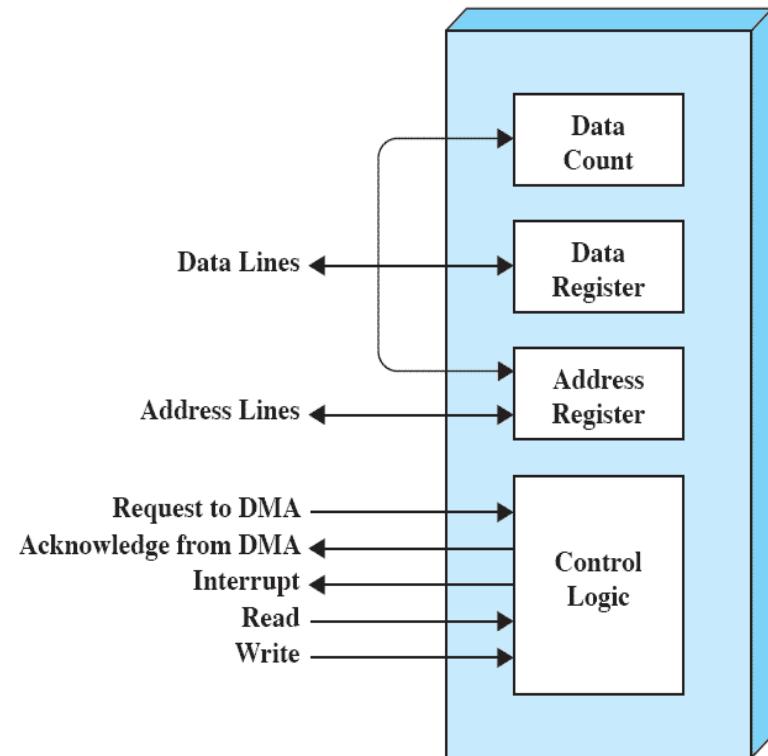
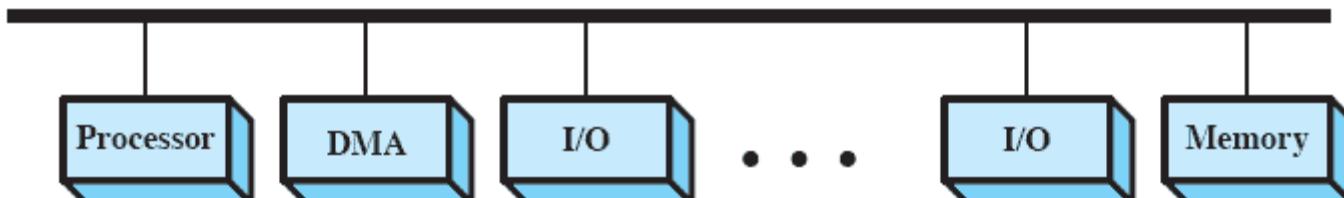


Figure 11.2 Typical DMA Block Diagram

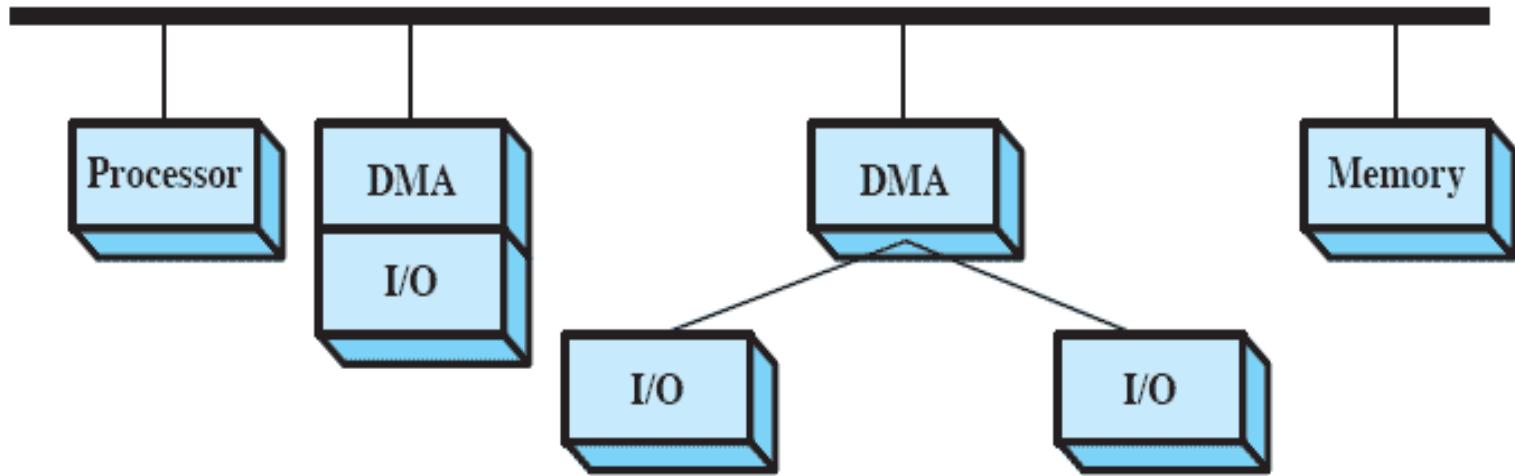
# DMA Configurations: Single Bus



(a) Single-bus, detached DMA

- DMA can be configured in several ways
- Shown here, all modules share the same system bus

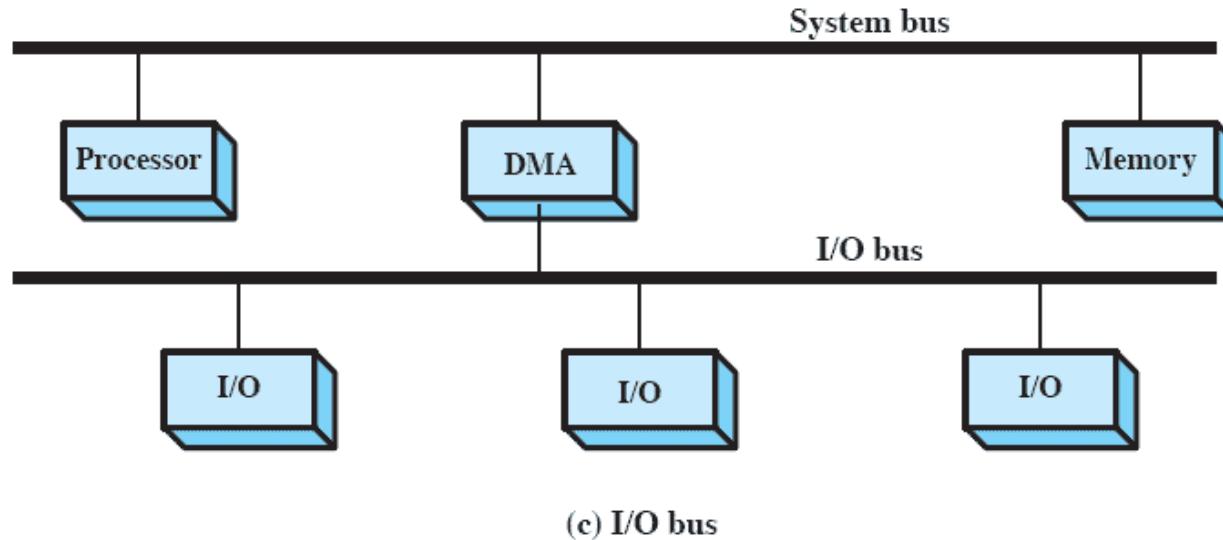
# DMA Configurations: Integrated DMA & I/O



(b) Single-bus, Integrated DMA-I/O

- Direct Path between DMA and I/O modules
- This substantially cuts the required bus cycles

# DMA Configurations: I/O Bus



- Reduces the number of I/O interfaces in the DMA module

# Outline

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- I/O Buffering
- Disk Scheduling

# Goals: Efficiency

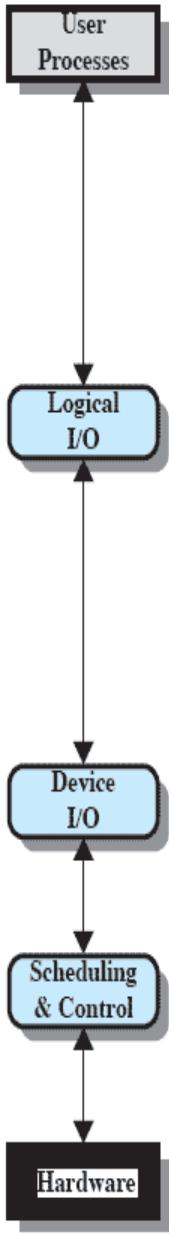
- Most I/O devices extremely slow compared to main memory
- Use of multiprogramming allows for some processes to be waiting on I/O while another process executes
- I/O cannot keep up with processor speed
  - Swapping used to bring in ready processes
  - But this is an I/O operation itself

# Generality

- For simplicity and freedom from error it is desirable to handle all I/O devices in a uniform manner
- Hide most of the details of device I/O in lower-level routines
- Difficult to completely generalize, but can use a hierarchical modular design of I/O functions

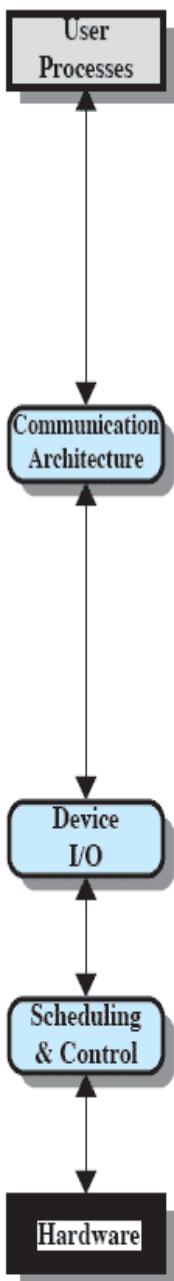
# Hierarchical design

- A hierarchical philosophy leads to organizing an OS into layers
- Each layer relies on the next lower layer to perform more primitive functions
- It provides services to the next higher layer.
- Changes in one layer should not require changes in other layers



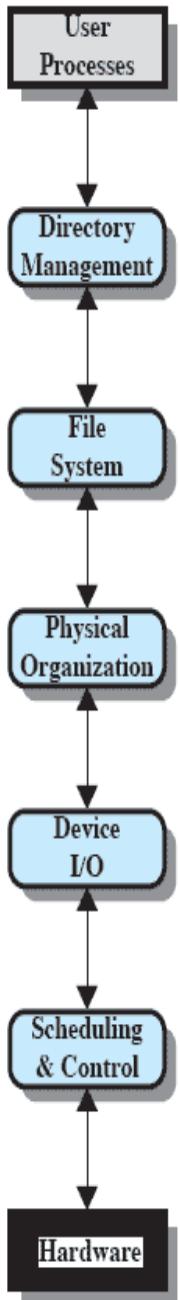
# Local peripheral device

- Logical I/O:
  - Deals with the device as a logical resource
- Device I/O:
  - Converts requested operations into sequence of I/O instructions
- Scheduling and Control
  - Performs actual queuing and control operations



# Communications Port

- Similar to previous but the logical I/O module is replaced by a communications architecture,
  - This consist of a number of layers.
  - An example is TCP/IP,



# File System

- Directory management
  - Concerned with user operations affecting files
- File System
  - Logical structure and operations
- Physical organisation
  - Converts logical names to physical addresses

# Outline

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- I/O Buffering
- Disk Scheduling
- Raid
- Disk Cache

# I/O Buffering

- Processes must wait for I/O to complete before proceeding
  - To avoid deadlock certain pages must remain in main memory during I/O
- It may be more efficient to perform input transfers in advance of requests being made and to perform output transfers some time after the request is made.

# Block-oriented Buffering

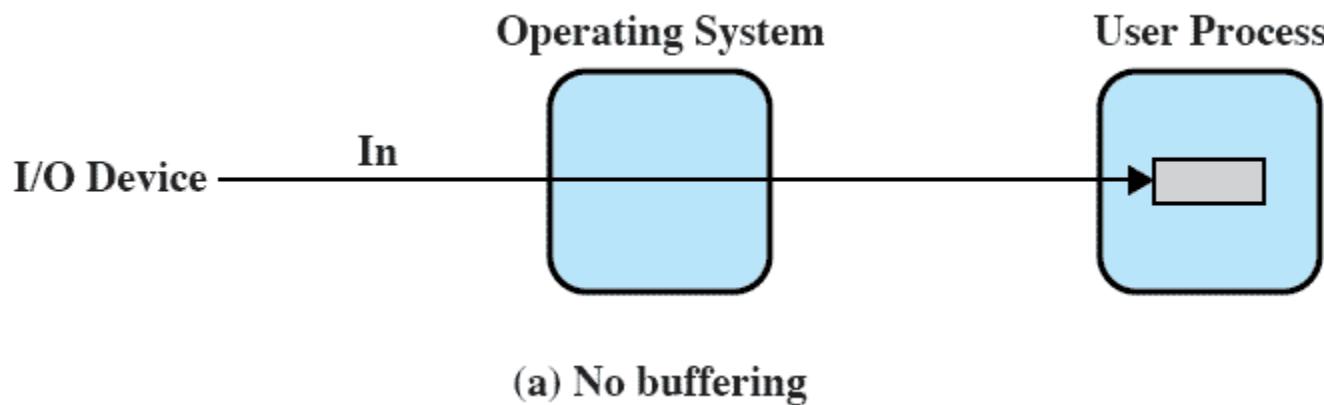
- Information is stored in fixed sized blocks
- Transfers are made a block at a time
  - Can reference data by block number
- Used for disks and USB keys

# Stream-Oriented Buffering

- Transfer information as a stream of bytes
- Used for terminals, printers, communication ports, mouse and other pointing devices, and most other devices that are not secondary storage

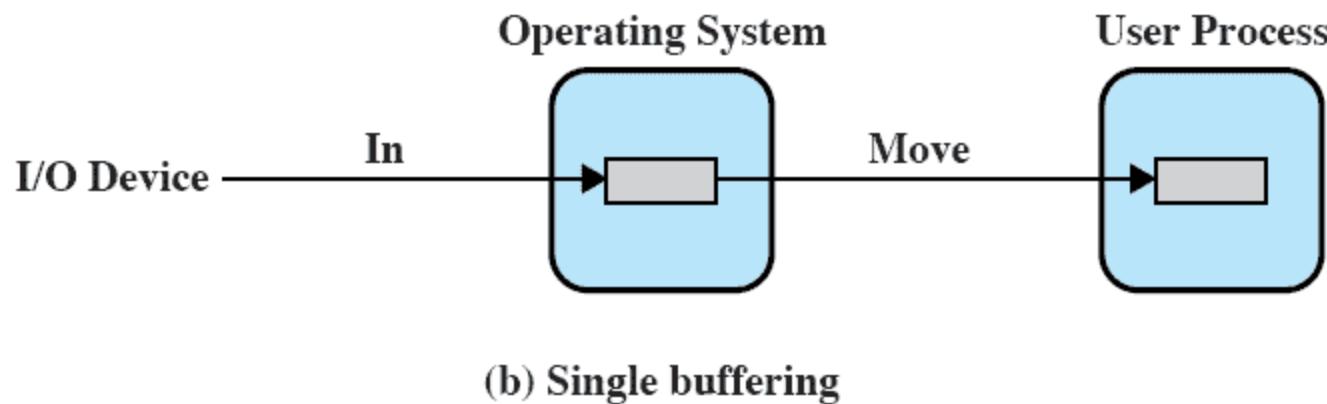
# No Buffer

- Without a buffer, the OS directly access the device as and when it needs



# Single Buffer

- Operating system assigns a buffer in main memory for an I/O request



# Block Oriented Single Buffer

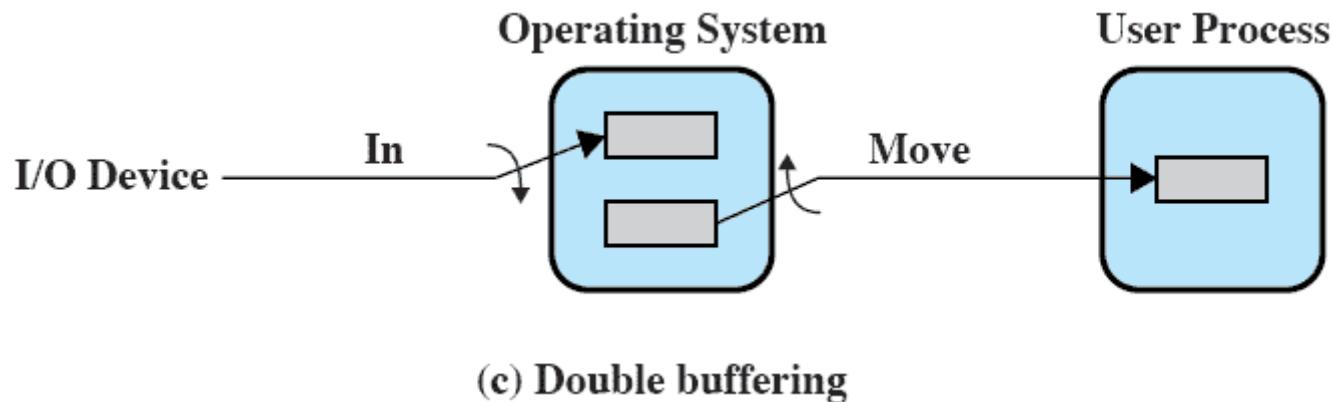
- Input transfers made to buffer
- Block moved to user space when needed
- The next block is moved into the buffer
  - *Read ahead or Anticipated Input*
- Often a reasonable assumption as data is usually accessed sequentially

# Stream-oriented Single Buffer

- Line-at-time or Byte-at-a-time
- Terminals often deal with one line at a time with carriage return signaling the end of the line
- Byte-at-a-time suites devices where a single keystroke may be significant
  - Also sensors and controllers

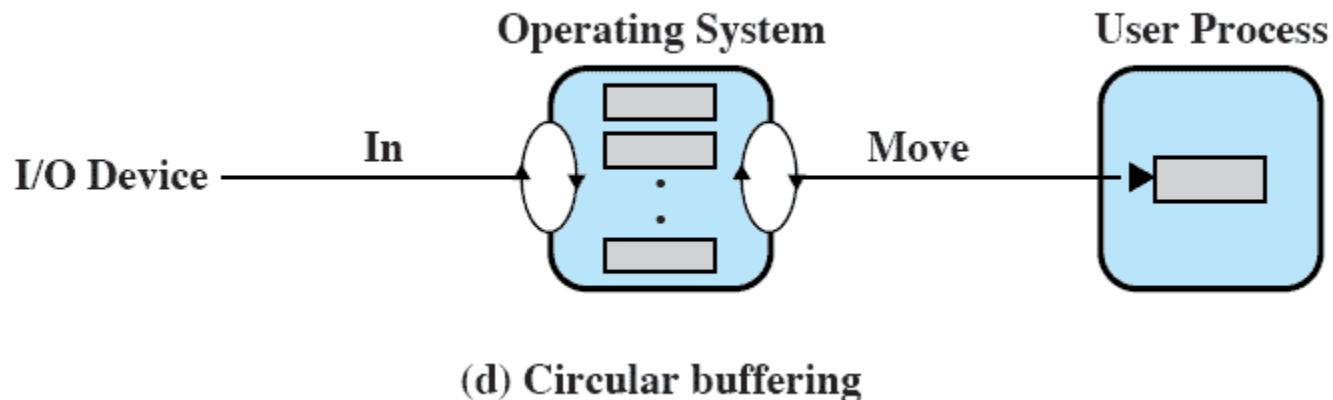
# Double Buffer

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



# Circular Buffer

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



# Buffer Limitations

- Buffering smoothes out peaks in I/O demand.
  - But with enough demand eventually all buffers become full and their advantage is lost
- However, when there is a variety of I/O and process activities to service, buffering can increase the efficiency of the OS and the performance of individual processes.

# Outline

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- I/O Buffering
- ➤ Disk Scheduling

# Disk Performance Parameters

- The actual details of disk I/O operation depend on many things
  - A general timing diagram of disk I/O transfer is shown here.



Figure 11.6 Timing of a Disk I/O Transfer

# Positioning the Read/Write Heads

- When the disk drive is operating, the disk is rotating at constant speed.
- Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system.

# Disk Performance Parameters

- **Access Time** is the sum of:
  - **Seek time:** The time it takes to position the head at the desired track
  - **Rotational delay or rotational latency:** The time its takes for the beginning of the sector to reach the head
- **Transfer Time** is the time taken to transfer the data.

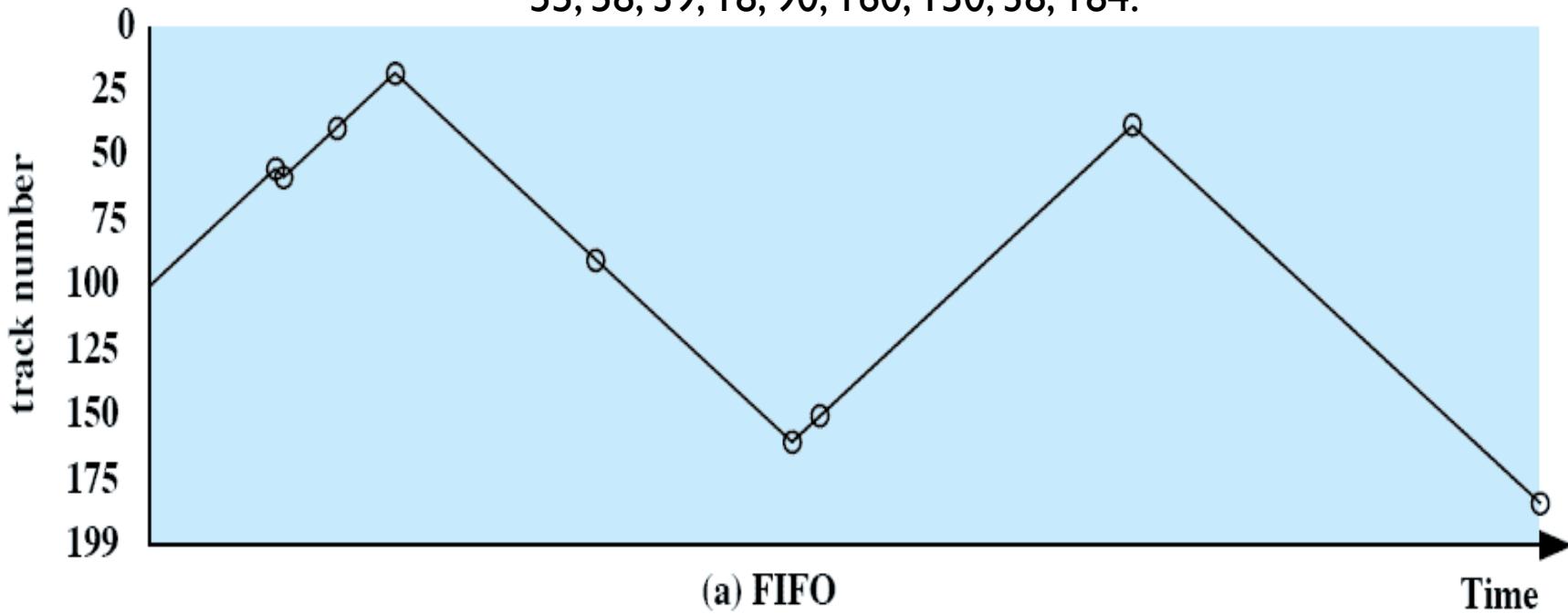
# Disk Scheduling Policies

- To compare various schemes, consider a disk head is initially located at track 100.
  - assume a disk with 200 tracks and that the disk request queue has random requests in it.
- The requested tracks, in the order received by the disk scheduler, are
  - 55, 58, 39, 18, 90, 160, 150, 38, 184.

# First-in, first-out (FIFO)

- Process request sequentially
- Fair to all processes
- Approaches random scheduling in performance if there are many processes

55, 58, 39, 18, 90, 160, 150, 38, 184.



# Priority

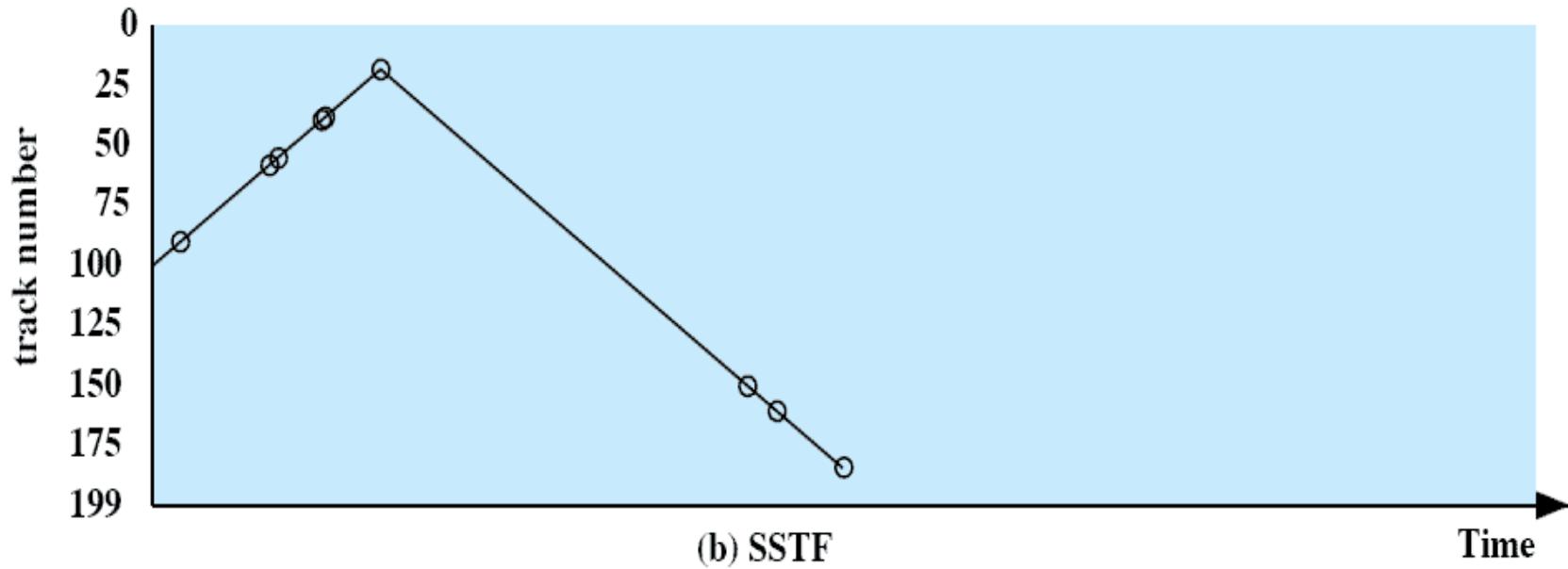
- Goal is not to optimize disk use but to meet other objectives
- Short batch jobs may have higher priority
- Provide good interactive response time
- Longer jobs may have to wait an excessively long time

# Last-in, first-out

- Good for transaction processing systems
  - The device is given to the most recent user so there should be little arm movement

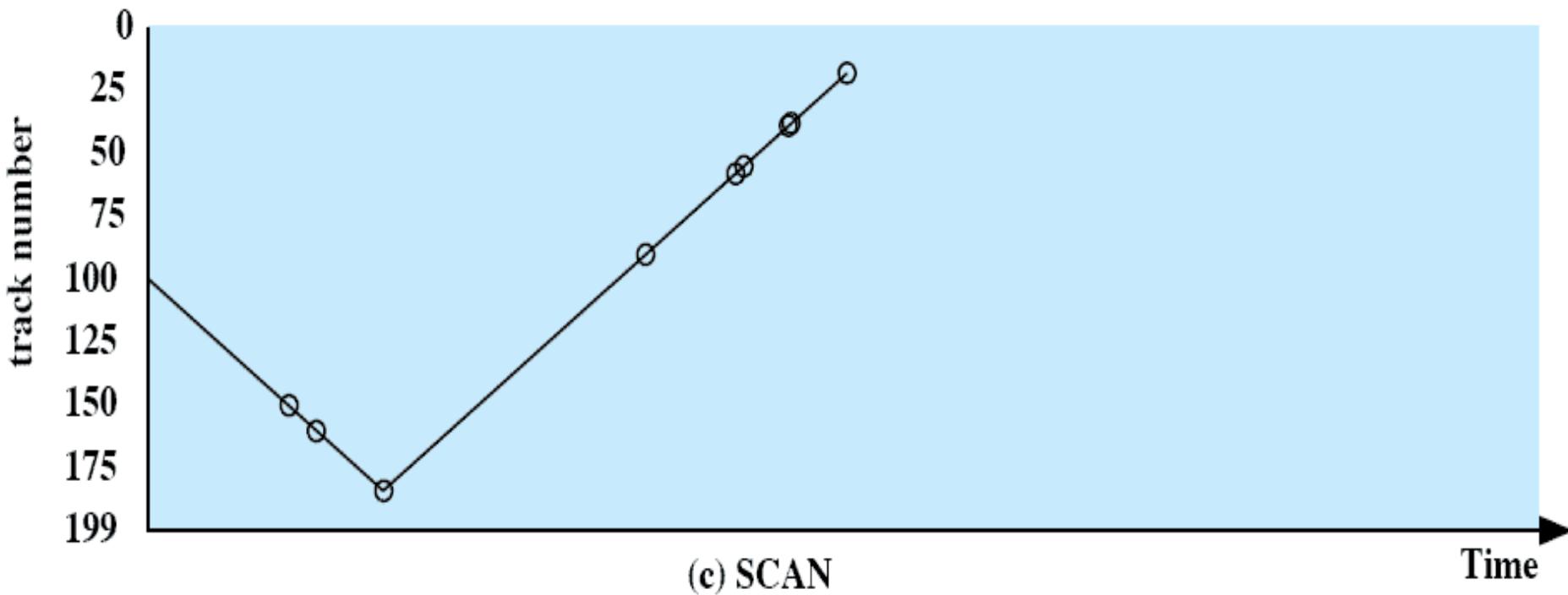
# Shortest Service Time First

- Select the disk I/O request that requires the least movement of the disk arm from its current position
- Always choose the minimum seek time
- 55, 58, 39, 18, 90, 160, 150, 38, 184.



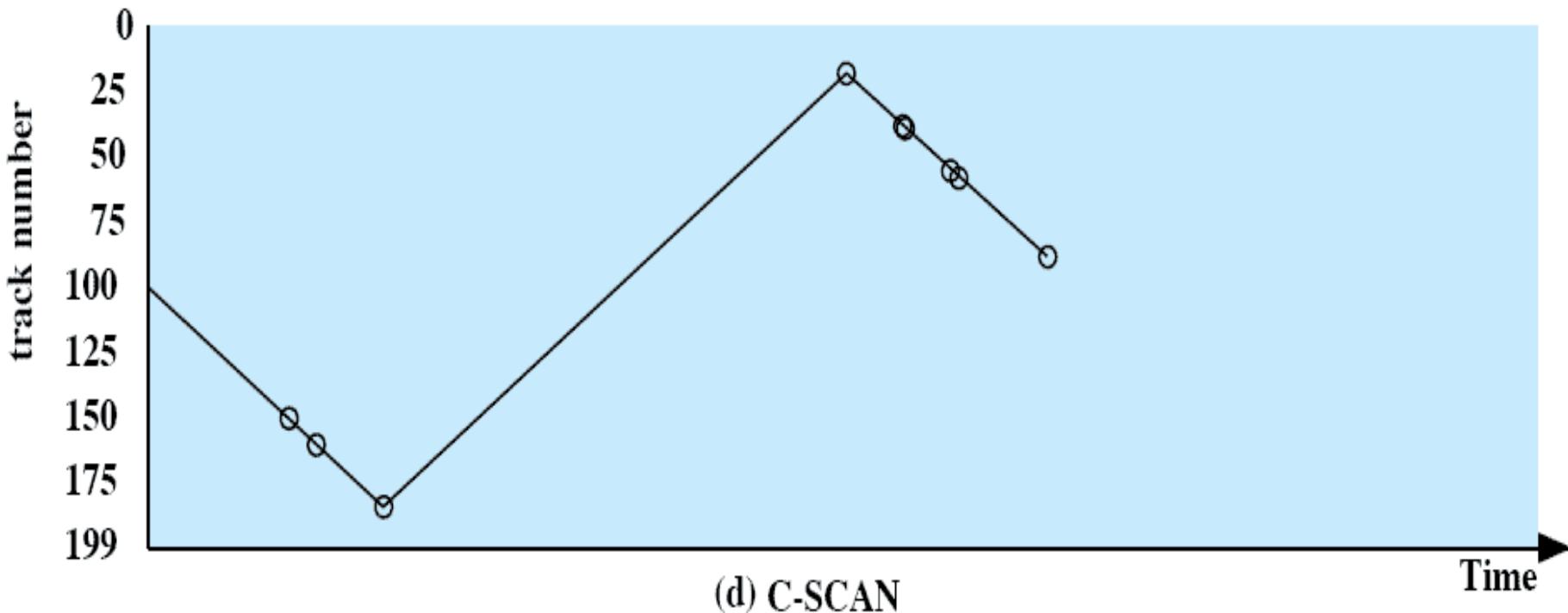
# SCAN

- Arm moves in one direction only, satisfying all outstanding requests until it reaches the last track in that direction then the direction is reversed
- 55, 58, 39, 18, 90, 160, 150, 38, 184.



# C-SCAN

- Restricts scanning to one direction only
- When the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again
- 55, 58, 39, 18, 90, 160, 150, 38, 184.



# N-step-SCAN

- Segments the disk request queue into subqueues of length N
- Subqueues are processed one at a time, using SCAN
- New requests added to other queue when queue is processed

# FSCAN

- Two subqueues
- When a scan begins, all of the requests are in one of the queues, with the other empty.
- All new requests are put into the other queue.
  - Service of new requests is deferred until all of the old requests have been processed.

# Performance Compared

## Comparison of Disk Scheduling Algorithms

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length		55.3		27.5		27.8	
Average seek length		27.5		length		length	

# Disk Scheduling Algorithms

**Table 11.3** Disk Scheduling Algorithms

Name	Description	Remarks
<b>Selection according to requestor</b>		
RSS	Random scheduling	For analysis and simulation
FIFO	First in first out	Fairest of them all
PRI	Priority by process	Control outside of disk queue management
LIFO	Last in first out	Maximize locality and resource utilization
<b>Selection according to requested item</b>		
SSTF	Shortest service time first	High utilization, small queues
SCAN	Back and forth over disk	Better service distribution
C-SCAN	One way with fast return	Lower service variability
N-step-SCAN	SCAN of $N$ records at a time	Service guarantee
FSCAN	N-step-SCAN with $N =$ queue size at beginning of SCAN cycle	Load sensitive



# Unit – 6, Chapter 12

## File Management



# Roadmap

1. Overview
2. File organisation and Access
3. File Directories
4. File Sharing

# Overview: Files

- Files are the central element to most applications
- The File System is one of the most important part of the OS to a user
- Desirable properties of files:
  - Long-term existence
  - Sharable between processes
  - Structure

# File Management

- File management system consists of system utility programs that run as privileged applications
- Concerned with secondary storage

# Typical Operations

- File systems also provide functions which can be performed on files, typically:
  - Create
  - Delete
  - Open
  - Close
  - Read
  - Write

# Terms

- Four terms are in common use when discussing files:
  - Field
  - Record
  - File
  - Database

# Fields and Records

- Fields
  - Basic element of data
  - Contains a single value
  - Characterized by its length and data type
- Records
  - Collection of related fields
  - Treated as a unit

# File and Database

- File
  - Have file names
  - Is a collection of similar records
  - Treated as a single entity
  - May implement access control mechanisms
- Database
  - Collection of related data
  - Relationships exist among elements
  - Consists of one or more files

# File Management Systems

- Provides services to users and applications in the use of files
  - The way a user or application accesses files
- Programmer does not need to develop file management software

# Objectives for a File Management System

- Meet the data management needs of the user
- Guarantee that the data in the file are valid
- Optimize performance
- Provide I/O support for a variety of storage device types
- Minimize lost or destroyed data
- Provide a standardized set of I/O interface routines to user processes
- Provide I/O support for multiple users (if needed)

# Requirements for a general purpose system

1. Each user should be able to create, delete, read, write and modify files
2. Each user may have controlled access to other users' files
3. Each user may control what type of accesses are allowed to the users' files
4. Each user should be able to restructure the user's files in a form appropriate to the problem
5. Each user should be able to move data between files
6. Each user should be able to back up and recover the user's files in case of damage

# Typical software organization

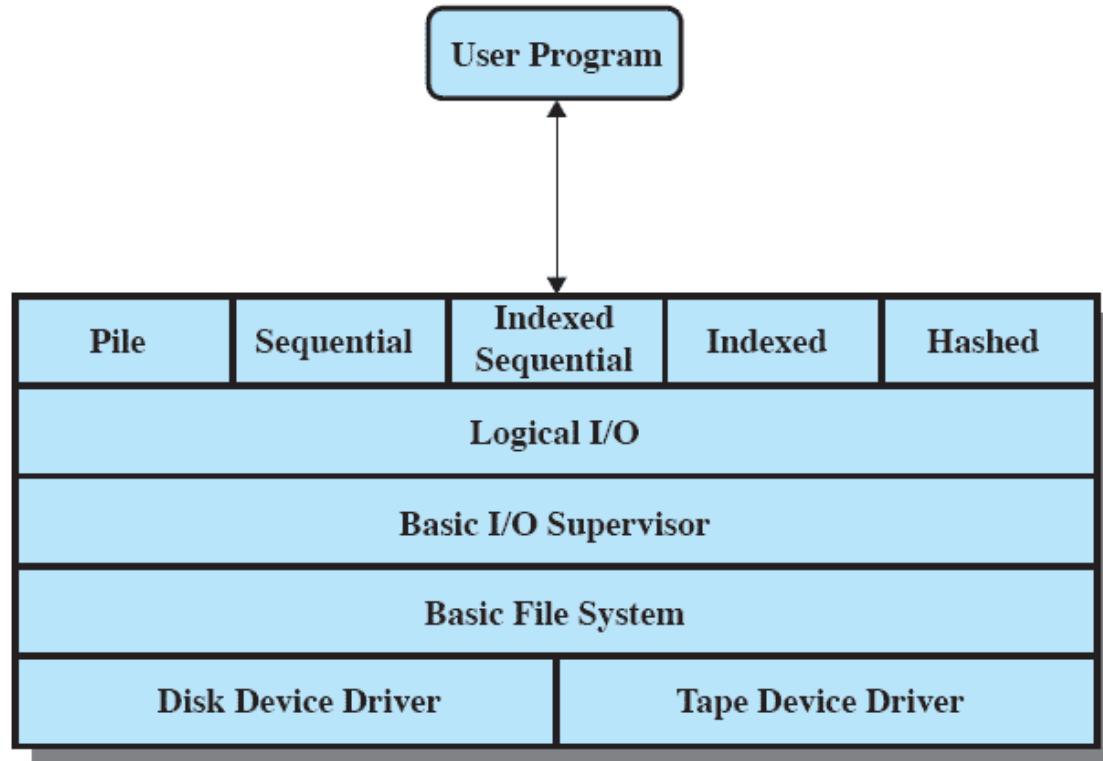


Figure 12.1 File System Software Architecture

# Device Drivers

- Lowest level
- Communicates directly with peripheral devices
- Responsible for starting I/O operations on a device
- Processes the completion of an I/O request

# Basic File System

- Physical I/O
- Primary interface with the environment outside the computer system
- Deals with exchanging blocks of data
- Concerned with the placement of blocks
- Concerned with buffering blocks in main memory

# Basic I/O Supervisor

- Responsible for all file I/O initiation and termination.
- Control structures deal with
  - Device I/O,
  - Scheduling,
  - File status.
- Selects and schedules I/O with the device

# Logical I/O

- Enables users and applications to access records
- Provides general-purpose record I/O capability
- Maintains basic data about file

# Access Method

- Closest to the user
- Reflect different file structures
- Provides a standard interface between applications and the file systems and devices that hold the data
- Access method varies depending on the ways to access and process data for the device.

# Elements of File Management

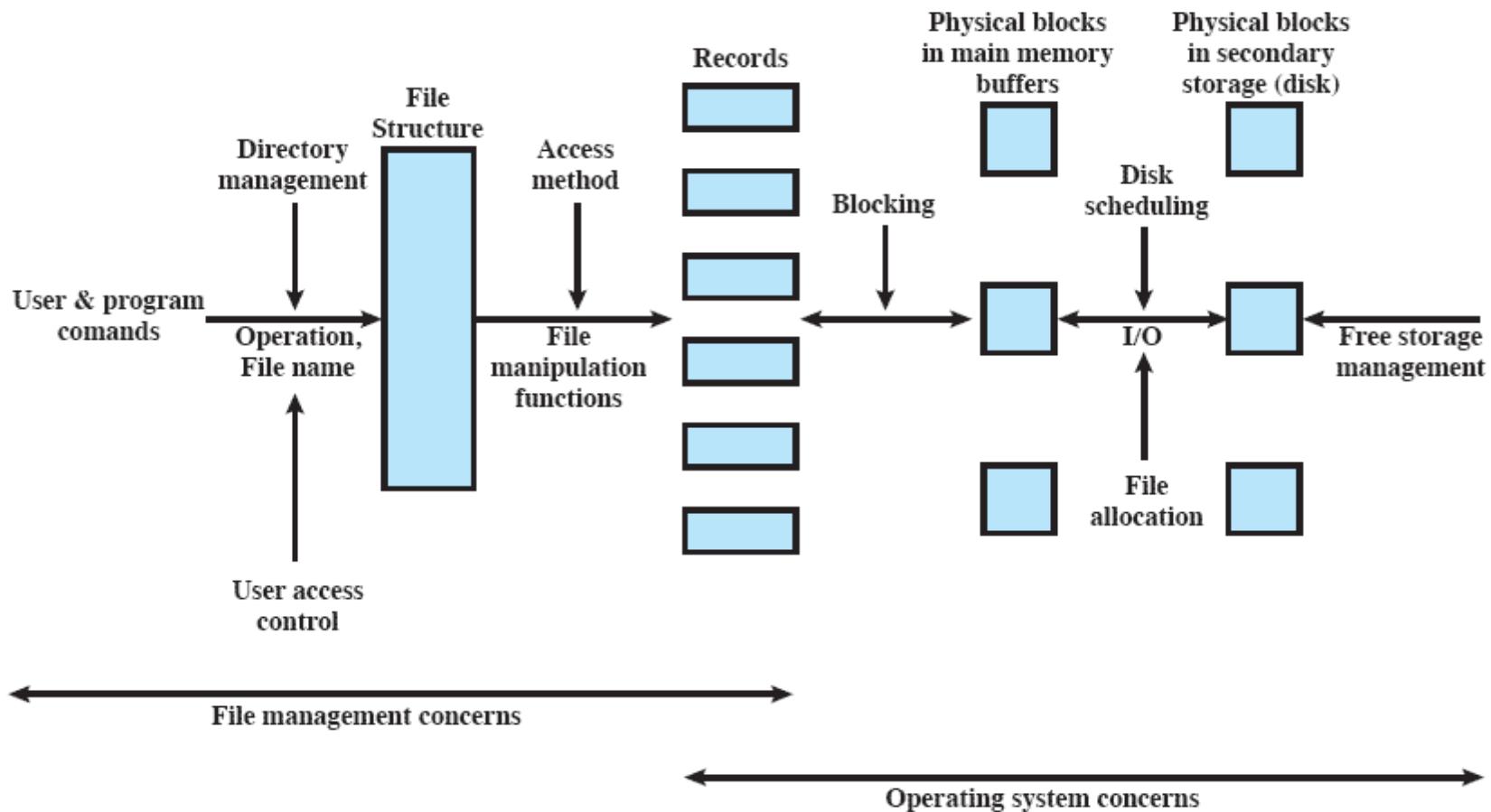


Figure 12.2 Elements of File Management



# Roadmap

1. Overview
2. File organisation and Access
3. File Directories
4. File Sharing

# File Organization

- File Management Referring to the logical structure of records
  - Physical organization discussed later
- Determined by the **way** in which files are accessed

# Criteria for File Organization

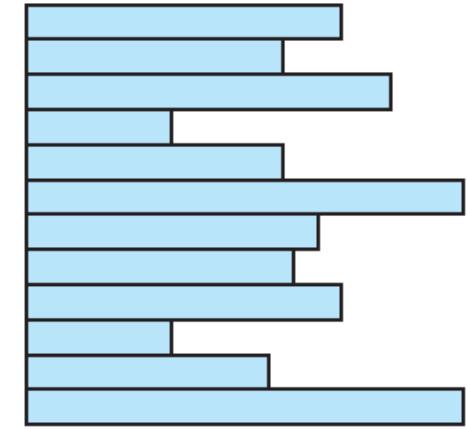
- Important criteria include:
  - Short access time
  - Ease of update
  - Economy of storage
  - Simple maintenance
  - Reliability

# File Organisation Types

- Many exist, but usually variations of:
  - Pile
  - Sequential file
  - Indexed sequential file
  - Indexed file
  - Direct, or hashed, file

# The Pile

- Data are collected in the order they arrive
  - No structure
- Purpose is to accumulate a mass of data and save it
- Records may have different fields
- Record access is by exhaustive search



Variable-length records  
Variable set of fields  
Chronological order

(a) Pile File

# The Sequential File

- Fixed format used for records
  - Records are the same length
  - All fields the same (order and length)
  - Field names and lengths are attributes of the file
  - Key field
    - Uniquely identifies the record
    - Records are stored in key sequence

Fixed-length records  
Fixed set of fields in fixed order  
Sequential order based on key field

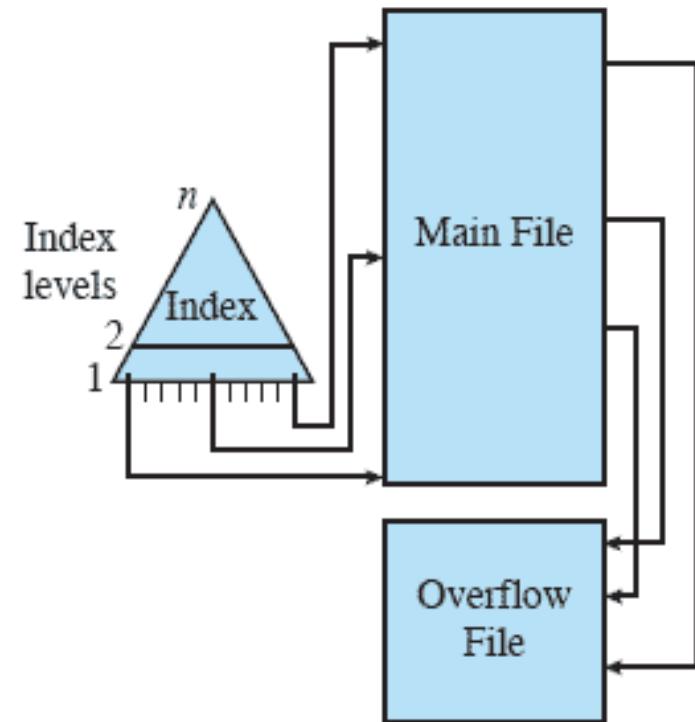
### (b) Sequential File

# Indexed Sequential File

- Maintains the key characteristic of the sequential file:
  - records are organized in sequence based on a key field.

Two features are added:

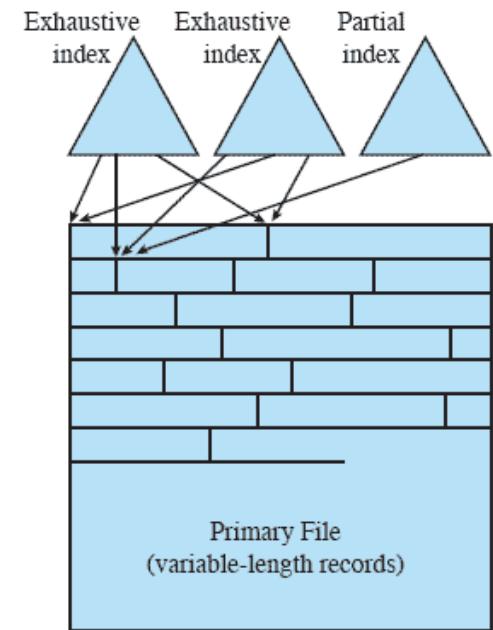
- an index to the file to support random access,
- and an overflow file.



(c) Indexed Sequential File

# Indexed File

- Uses multiple indexes for different key fields
  - May contain an exhaustive index that contains one entry for every record in the main file
  - May contain a partial index
- When a new record is added to the main file, all of the index files must be updated.



(d) Indexed File

# File Organization

- Access directly any block of a known address.
- The Direct or Hashed File
  - Directly access a block at a known address
  - Key field required for each record



# Roadmap

1. Overview
2. File organisation and Access
3. **File Directories**
4. File Sharing

# File Directories : Contents

- Contains information about files
  - Attributes
  - Location
  - Ownership
- Directory itself is a file owned by the operating system
- Provides mapping between file names and the files themselves

# Directory Elements: Basic Information

- File Name
  - Name as chosen by creator (user or program).
  - Must be unique within a specific directory.
- File type
- File Organisation
  - For systems that support different organizations

# Directory Elements: Address Information

- Volume
  - Indicates device on which file is stored
- Starting Address
- Size Used
  - Current size of the file in bytes, words, or blocks
- Size Allocated
  - The maximum size of the file

# Directory Elements: Access Control Information

- Owner
  - The owner may be able to grant/deny access to other users and to change these privileges.
- Access Information
  - May include the user's name and password for each authorized user.
- Permitted Actions
  - Controls reading, writing, executing, transmitting over a network

# Directory Elements: Usage Information

- Date Created
- Identity of Creator
- Date Last Read Access
- Identity of Last Reader
- Date Last Modified
- Identity of Last Modifier
- Date of Last Backup
- Current Usage
  - Current activity, locks, etc

# Simple Structure for a Directory

- The method for storing the previous information varies widely between systems
- Simplest is a list of entries, one for each file
  - Sequential file with the name of the file serving as the key
  - Provides no help in organizing the files
  - Forces user to be careful not to use the same name for two different files

# Operations Performed on a Directory

- A directory system should support a number of operations including:
  - Search
  - Create files
  - Deleting files
  - Listing directory
  - Updating directory

# Two-Level Scheme for a Directory

- One directory for each user and a master directory
  - Master directory contains entry for each user
  - Provides address and access control information
- Each user directory is a simple list of files for that user
  - Does not provide structure for collections of files

# Hierarchical, or Tree-Structured Directory

- Master directory with user directories underneath it
- Each user directory may have subdirectories and files as entries

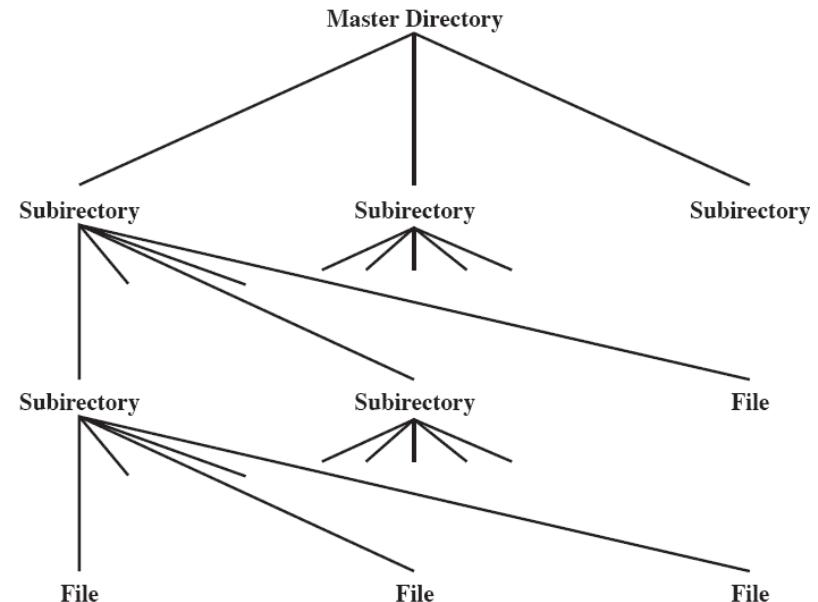


Figure 12.4 Tree-Structured Directory

# Naming

- Users need to be able to refer to a file by name
  - Files need to be named uniquely, but users may not be aware of all filenames on a system
- The tree structure allows users to find a file by following the directory path
  - Duplicate filenames are possible if they have different pathnames

# Example of Tree-Structured Directory

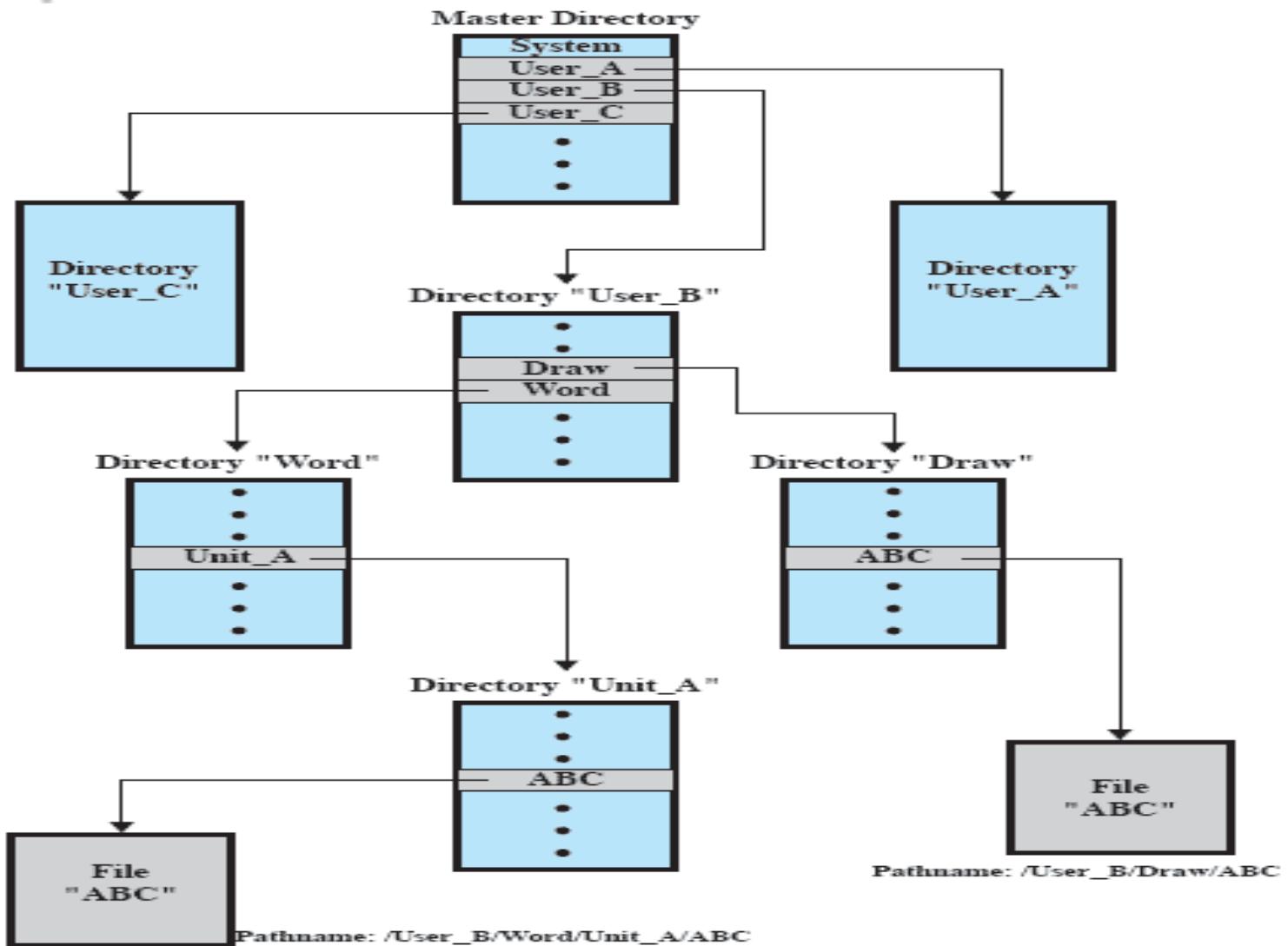


Figure 12.5 Example of Tree-Structured Directory

# Working Directory

- Stating the full pathname and filename is awkward and tedious
- Usually an interactive user or process is associated with a **current or working directory**
  - All file names are referenced as being relative to the working directory unless an explicit full pathname is used



# Roadmap

- 
1. Overview
  2. File organisation and Access
  3. File Directories
  4. File Sharing

# File Sharing

- In multiuser system, allow files to be shared among users
- Two issues
  - Access rights
  - Management of simultaneous access

# Access Rights

- A wide variety of access rights have been used by various systems
  - often as a hierarchy where one right implies previous
- None
  - User may not even know of the files existence
- Knowledge
  - User can only determine that the file exists and who its owner is

# Access Rights cont...

- Execution
  - The user can load and execute a program but cannot copy it
- Reading
  - The user can read the file for any purpose, including copying and execution
- Appending
  - The user can add data to the file but cannot modify or delete any of the file's contents

# Access Rights cont...

- **Updating**
  - The user can modify, delete, and add to the file's data.
- **Changing protection**
  - User can change access rights granted to other users
- **Deletion**
  - User can delete the file

# User Classes

- Owner
  - Usually the files creator, usually has full rights
- Specific Users
  - Rights may be explicitly granted to specific users
- User Groups
  - A set of users identified as a group
- All
  - everyone

# Simultaneous Access

- User may lock entire file when it is to be updated
- User may lock the individual records during the update
- Mutual exclusion and deadlock are issues for shared access