

# OPERATING SYSTEMS [20MCA15C]

## UNIT – II “Processes, CPU Scheduling, Deadlocks”

FACULTY:

**Dr. R. A. Roseline, M.Sc., M.Phil., Ph.D.,**

Associate Professor and Head,  
Post Graduate and Research Department of Computer Applications,  
Government Arts College (Autonomous), Coimbatore – 641 018.



# Processes

- **Process Concept**
- **Process Scheduling**
- **Operations on Processes**
- **Cooperating Processes**
- **Interprocess Communication**
- **Communication in Client-Server Systems**



# Process Concept



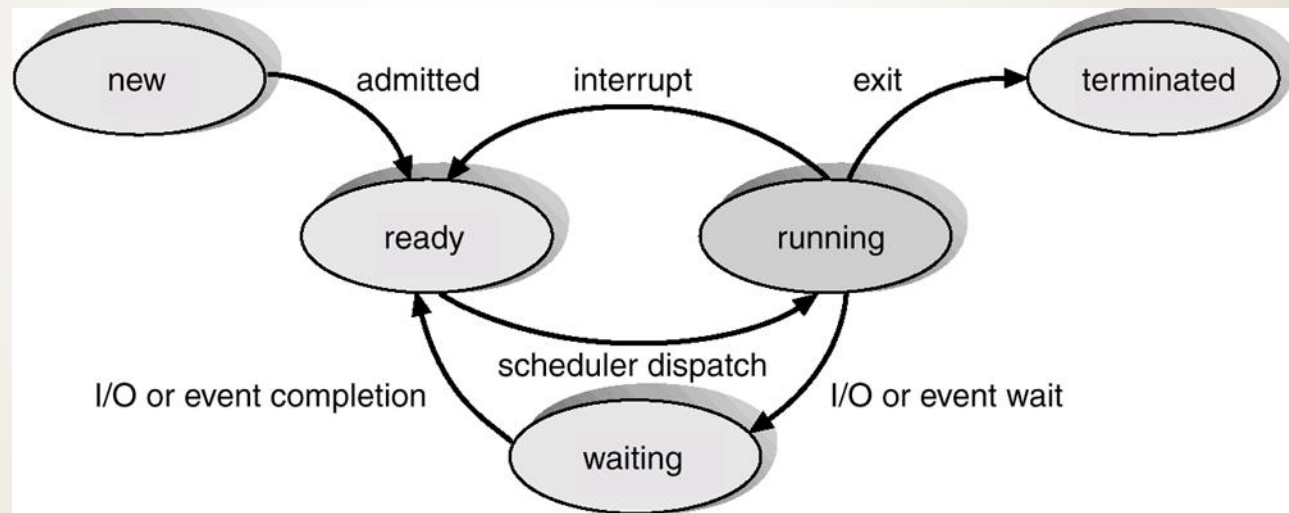
- **An operating system executes a variety of programs:**
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- **Textbook uses the terms job and process almost interchangeably.**
- **Process – a program in execution; process execution must progress in sequential fashion.**
- **A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



# Process State

- **As a process executes, it changes state**
  - **new:** The process is being created.
  - **running:** Instructions are being executed.
  - **waiting:** The process is waiting for some event to occur.
  - **ready:** The process is waiting to be assigned to a process.
  - **terminated:** The process has finished execution.

# Diagram of Process State





# Process Control Block (PCB)

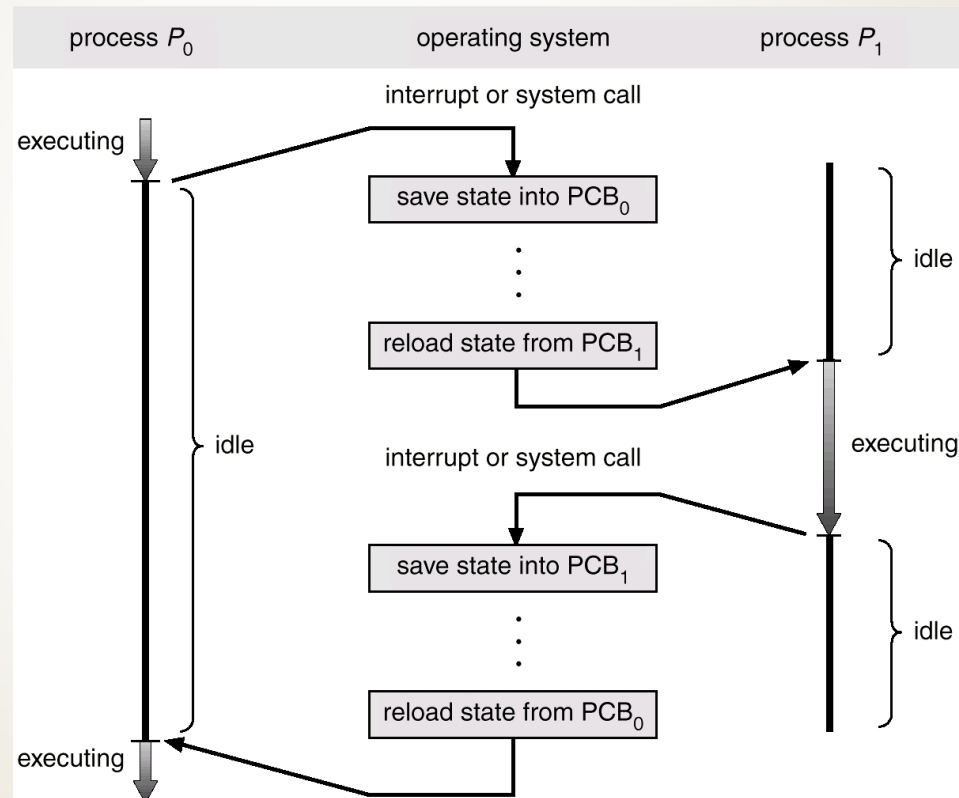
- Information associated with each process.
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



# Process Control Block (PCB)

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	

# CPU Switch From Process to Process



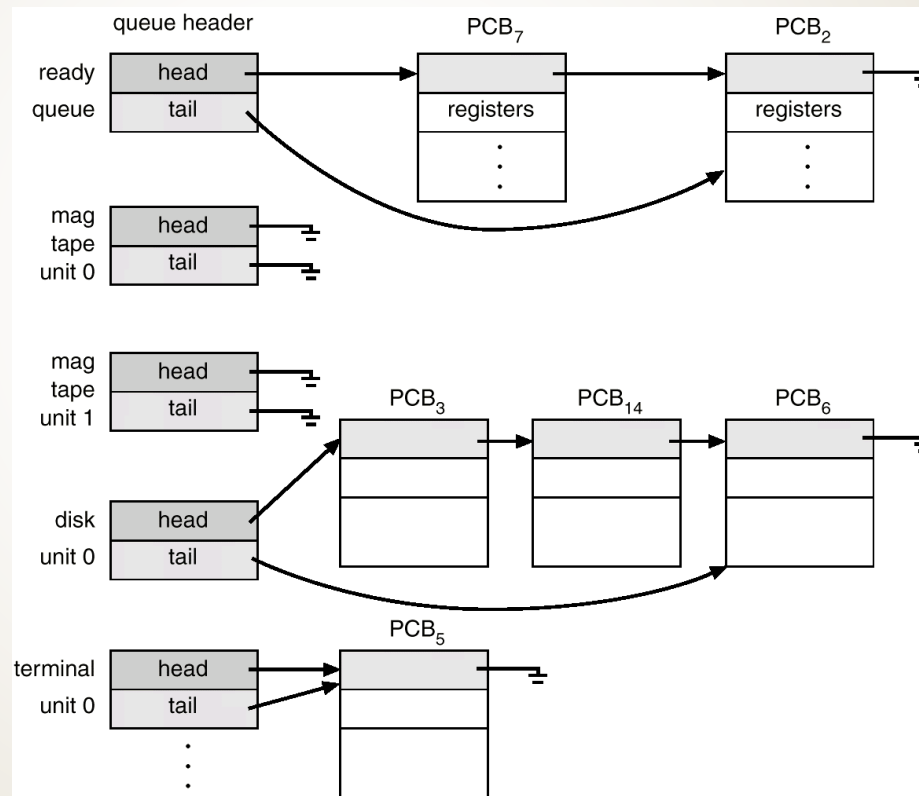




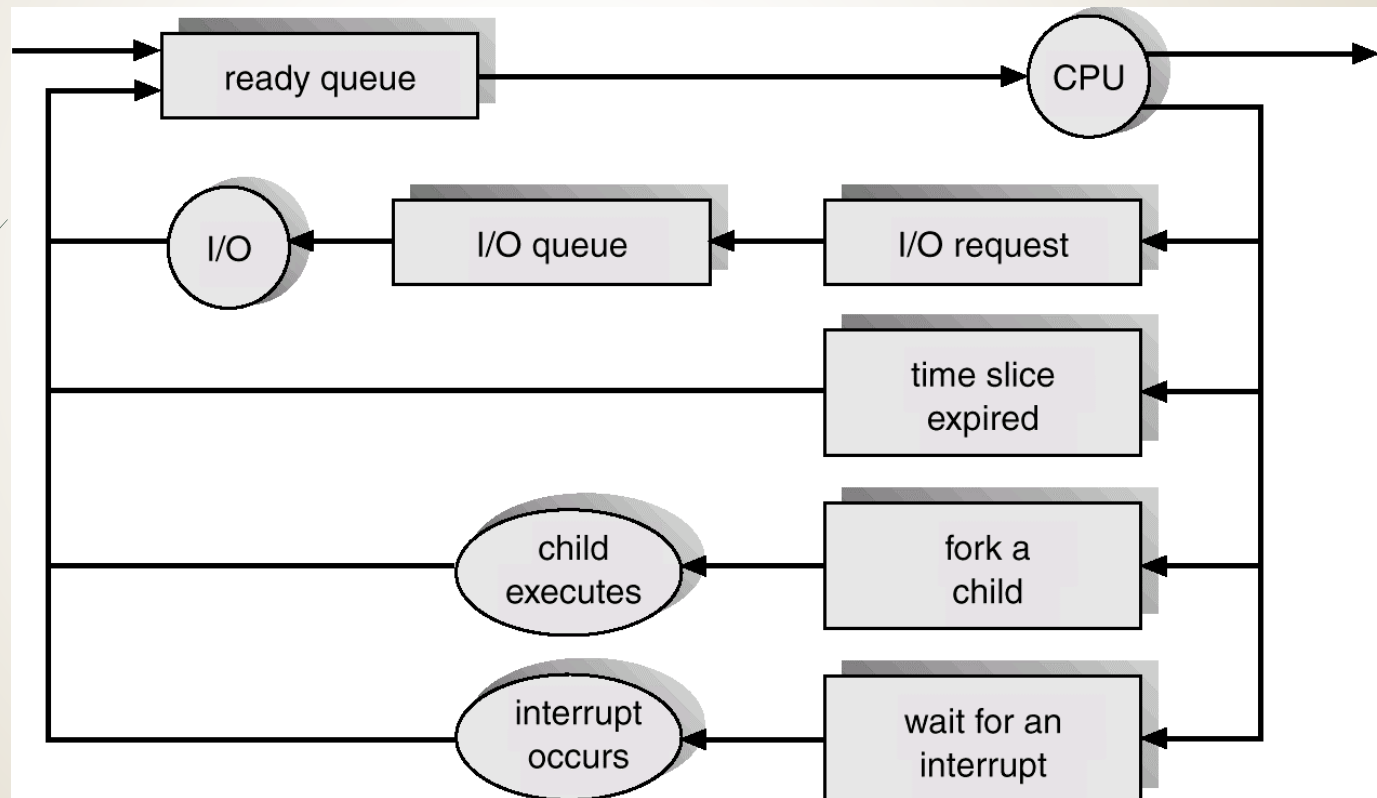
# Process Scheduling Queues

- **Job queue** – set of all processes in the system.
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute.
- **Device queues** – set of processes waiting for an I/O device.
- **Process migration** between the various queues.

# Ready Queue And Various I/O Device Queues



# Representation of Process Scheduling

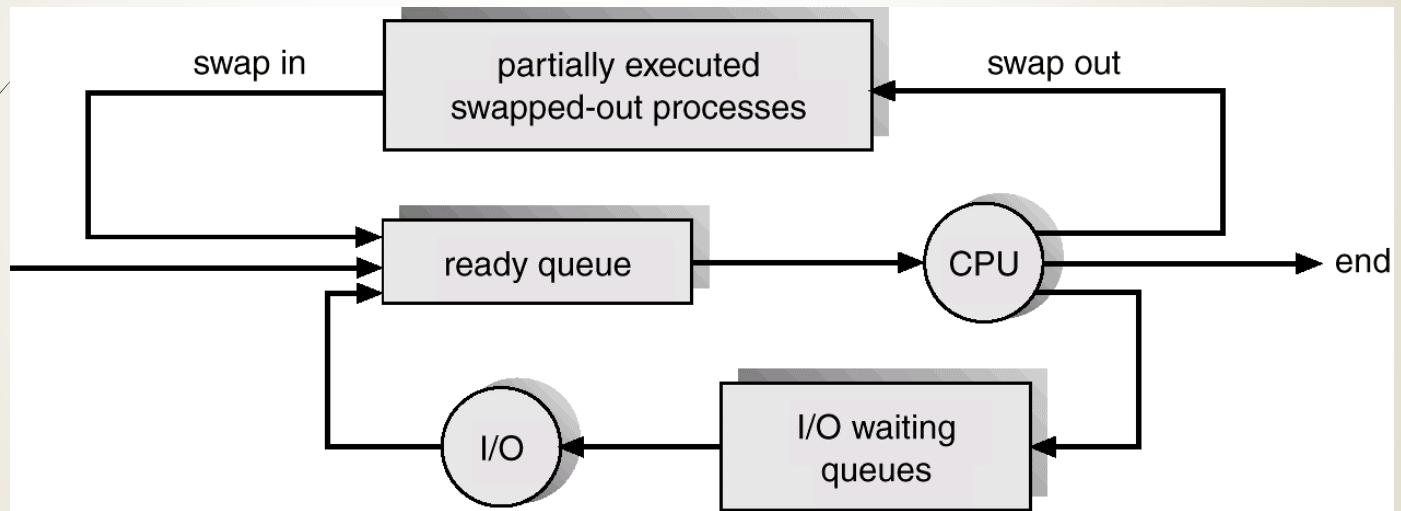




# Schedulers

- **Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue.**
- **Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.**

# Addition of Medium Term Scheduling





# Schedulers (Cont.)

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



# Context Switch

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



# Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes.
- Resource sharing
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

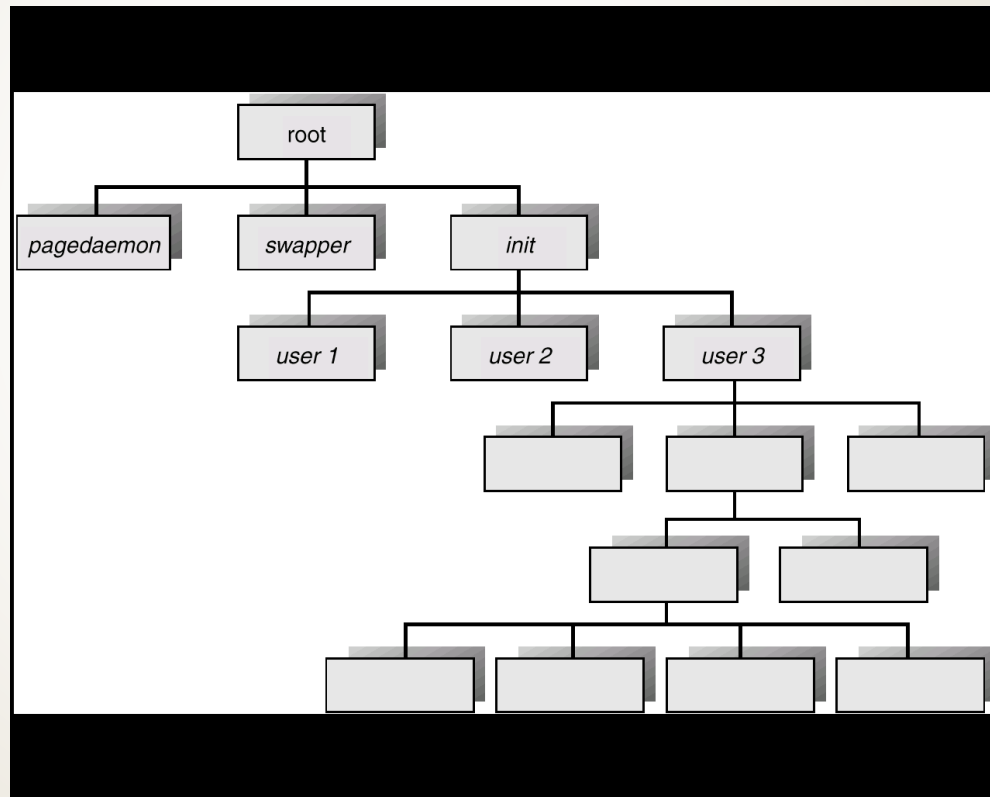




# Process Creation (Cont.)

- **Address space**
  - Child duplicate of parent.
  - Child has a program loaded into it.
- **UNIX examples**
  - fork system call creates new process
  - exec system call used after a fork to replace the process' memory space with a new program.

# Processes Tree on a UNIX System





# Process Termination

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



# Cooperating Processes

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



# Producer-Consumer Problem

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

# Bounded-Buffer – Shared-Memory Solution

## ➤ Shared data

```
➤ #define BUFFER_SIZE 10
➤ Typedef struct {
➤     ...
➤ } item;
➤ item buffer[BUFFER_SIZE];
➤ int in = 0;
➤ int out = 0;
```

## ➤ Solution is correct, but can only use BUFFER\_SIZE-1 elements



# Bounded-Buffer – Producer Process

```
➤ item nextProduced;

➤ while (1) {
➤     while (((in + 1) % BUFFER_SIZE) == out)
➤         ; /* do nothing */
➤     buffer[in] = nextProduced;
➤     in = (in + 1) % BUFFER_SIZE;
➤ }
```



# Bounded-Buffer – Consumer Process

```
➤ item nextConsumed;  
➤ while (1) {  
➤     while (in == out)  
➤         ; /* do nothing */  
➤     nextConsumed = buffer[out];  
➤     out = (out + 1) % BUFFER_SIZE;  
➤ }
```





# Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- IPC facility provides two operations:
  - send(message) – message size fixed or variable
  - receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)



# Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?



# Direct Communication

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



# Indirect Communication

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



# Indirect Communication

- **Operations**

- create a new mailbox
- send and receive messages through mailbox
- destroy a mailbox

- **Primitives are defined as:**

- **send(A, message) – send a message to mailbox A**
- **receive(A, message) – receive a message from mailbox A**



# Indirect Communication

## ➤ Mailbox sharing

- P1, P2, and P3 share mailbox A.
- P1, sends; P2 and P3 receive.
- Who gets the message?

## ➤ Solutions

- Allow a link to be associated with at most two processes.
- Allow only one process at a time to execute a receive operation.
- Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.



# Synchronization

- **Message passing may be either blocking or non-blocking.**
- **Blocking is considered synchronous**
- **Non-blocking is considered asynchronous**
- **send and receive primitives may be either blocking or non-blocking.**



# Buffering

- Queue of messages attached to the link; implemented in one of three ways.
  - 1. Zero capacity – 0 messages  
Sender must wait for receiver (rendezvous).
  - 2. Bounded capacity – finite length of  $n$  messages  
Sender must wait if link full.
  - 3. Unbounded capacity – infinite length  
Sender never waits.





# Client-Server Communication

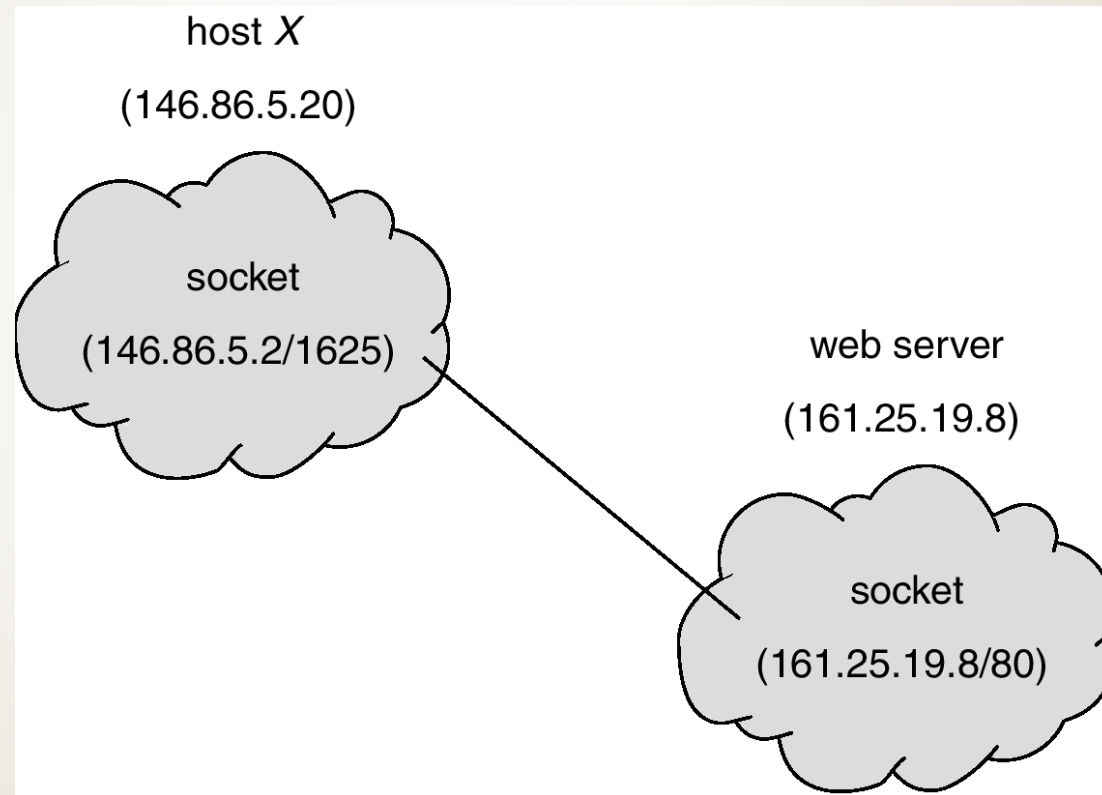
- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)



# Sockets

- A socket is defined as an endpoint for communication.
- Concatenation of IP address and port
- The socket `161.25.19.8:1625` refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets.

# Socket Communication

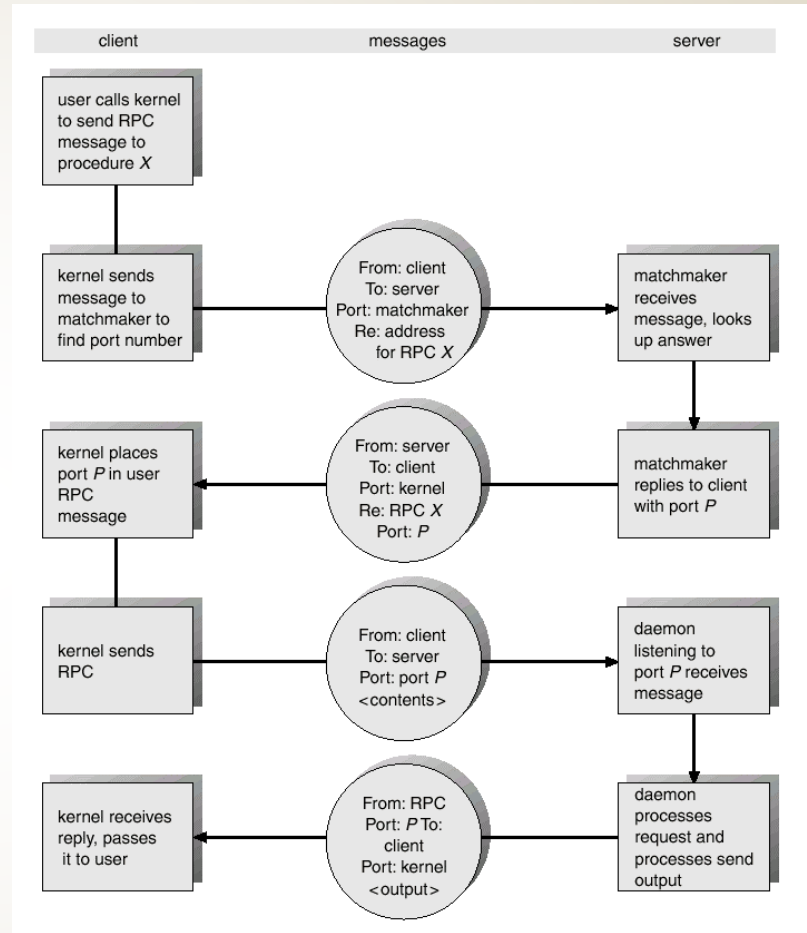




# Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- Stubs – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and marshalls the parameters.
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server.

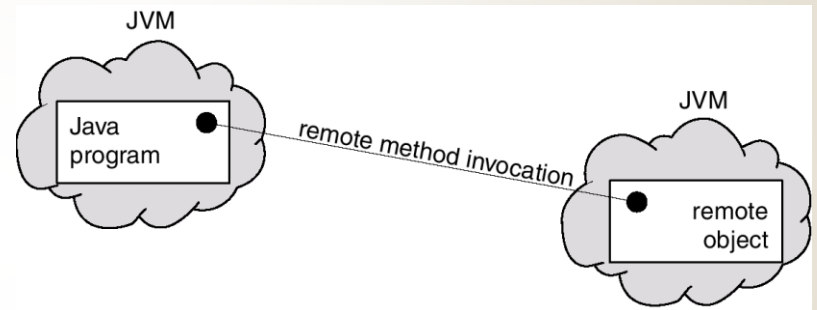
# Execution of RPC



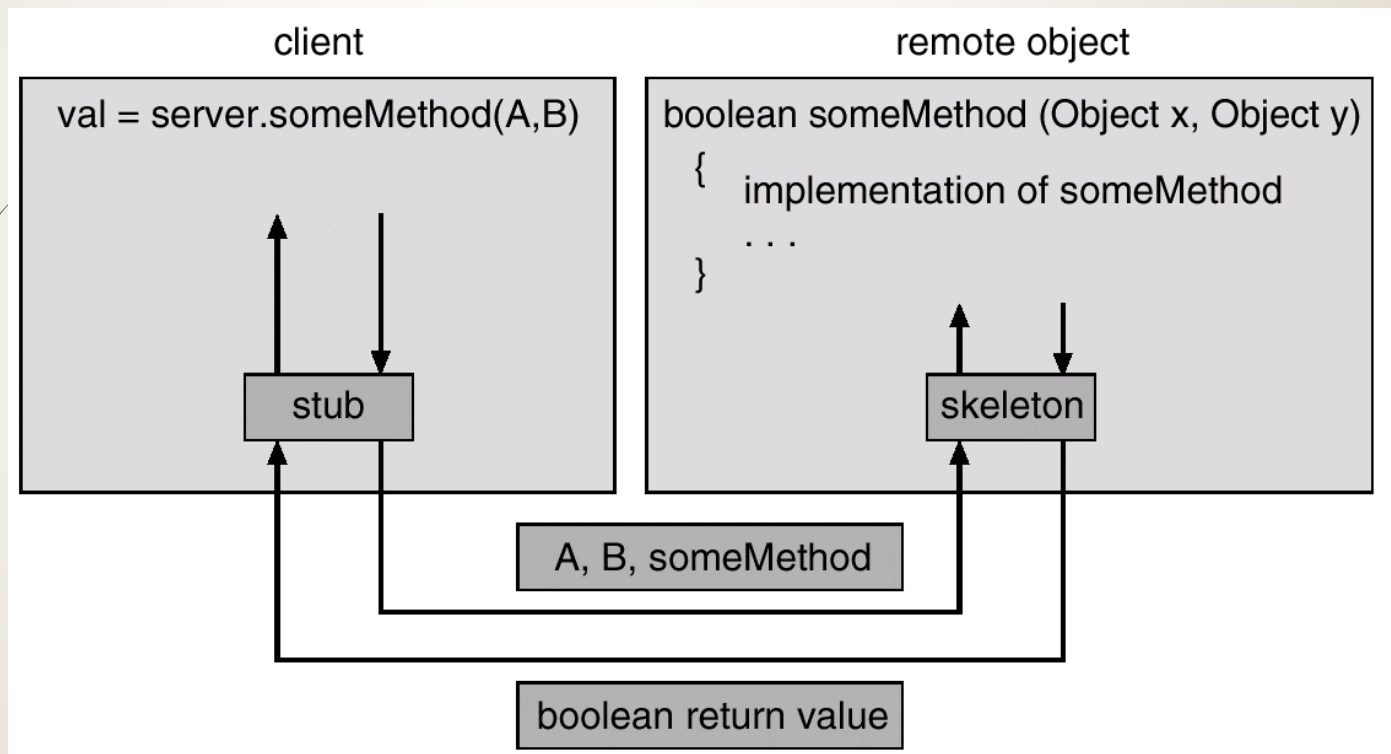
# Remote Method Invocation

Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.

RMI allows a Java program on one machine to invoke a method on a remote object.




# Marshalling Parameters





# CPU Scheduling

- **Basic Concepts**
  - **Scheduling Criteria**
  - **Scheduling Algorithms**
  - **Multiple-Processor Scheduling**
  - **Real-Time Scheduling**
  - **Algorithm Evaluation**
- 

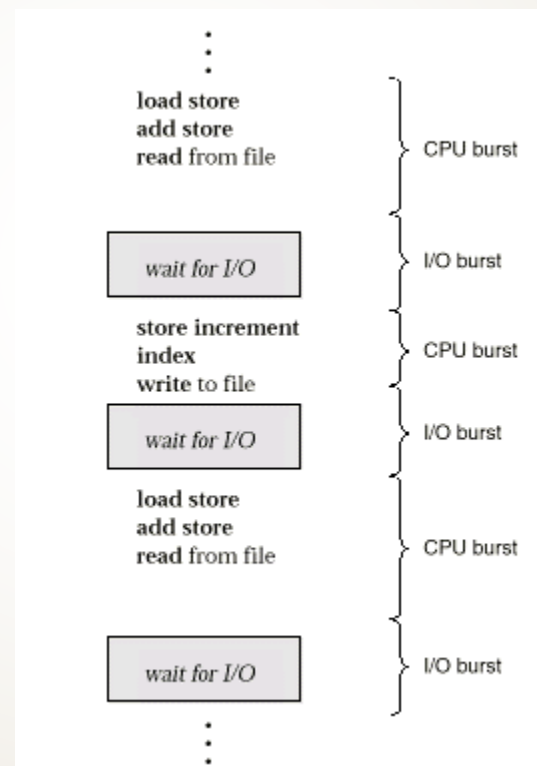




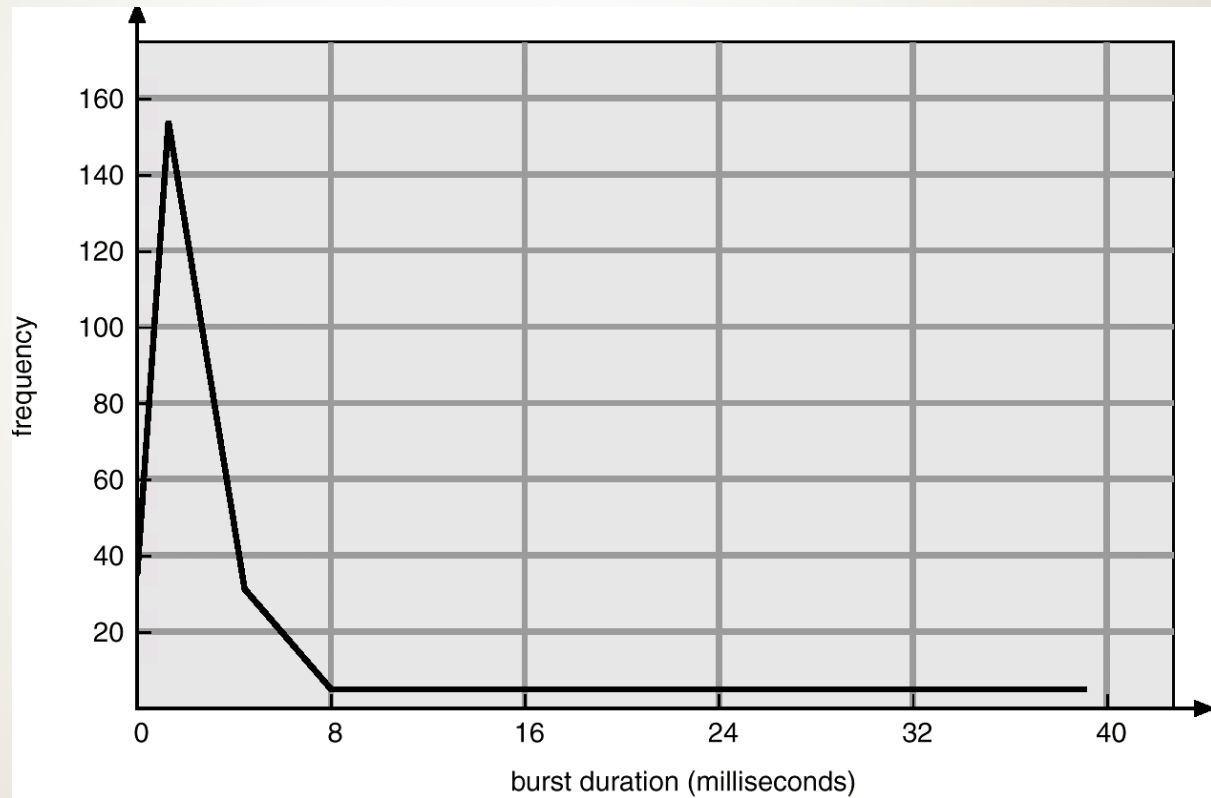
# Basic Concepts

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

# Alternating Sequence of CPU And I/O Bursts



# Histogram of CPU-burst Times





# CPU Scheduler

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



# Dispatcher

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



# 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)



# Optimization Criteria

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

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

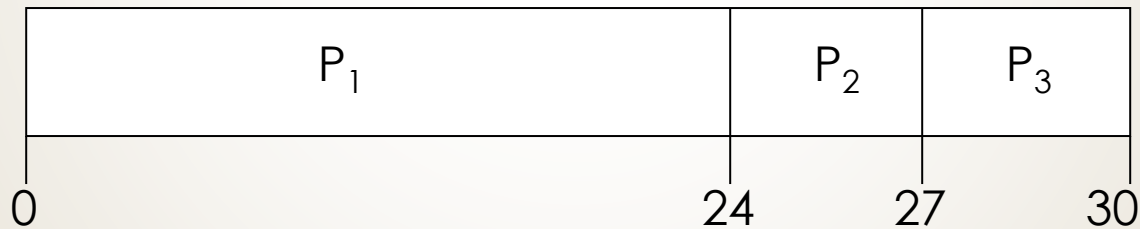
➤ Process Burst Time

➤ P1 24

➤ P2 3

➤ P3 3

➤ Suppose that the processes arrive in the order: P1 , P2 , P3  
The Gantt Chart for the schedule is:



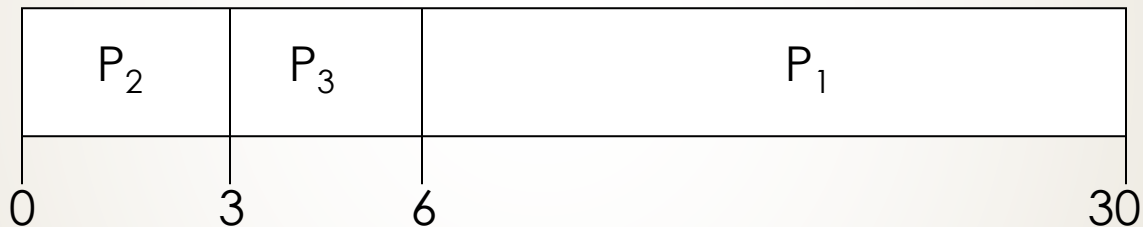
➤ Waiting time for P1 = 0; P2 = 24; P3 = 27

➤ Average waiting time:  $(0 + 24 + 27)/3 = 17$




# FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order
- P<sub>2</sub> , P<sub>3</sub> , P<sub>1</sub> .
- The Gantt chart for the schedule is:



- Waiting time for P<sub>1</sub> = 6; P<sub>2</sub> = 0; P<sub>3</sub> = 3
- Average waiting time:  $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- Convoy effect short process behind long process



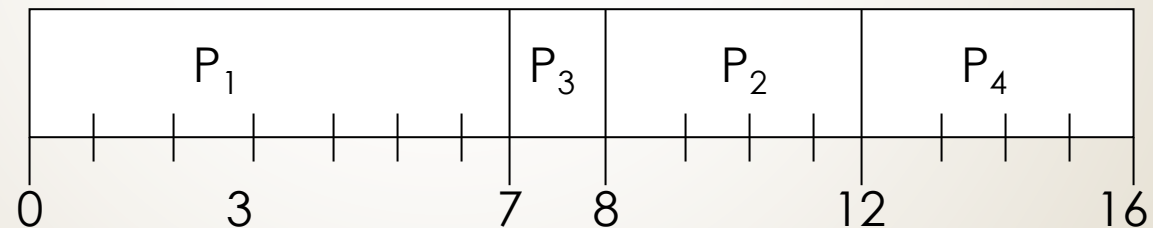
# Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.

# Example of Non-Preemptive SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0.0	7
$P_2$	2.0	4
$P_3$	4.0	1
$P_4$	5.0	4

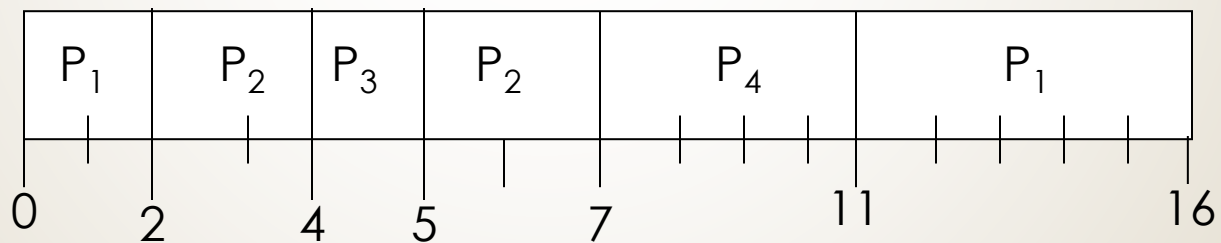
➤ SJF (non-preemptive)



➤ Average waiting time =  $(0 + 6 + 3 + 7)/4 - 4$

# Example of Preemptive SJF

- Process Arrival Time Burst Time
- P1 0.0 7
- P2 2.0 4
- P3 4.0 1
- P4 5.0 4
- SJF (preemptive)



- Average waiting time =  $(9 + 1 + 0 + 2)/4 - 3$

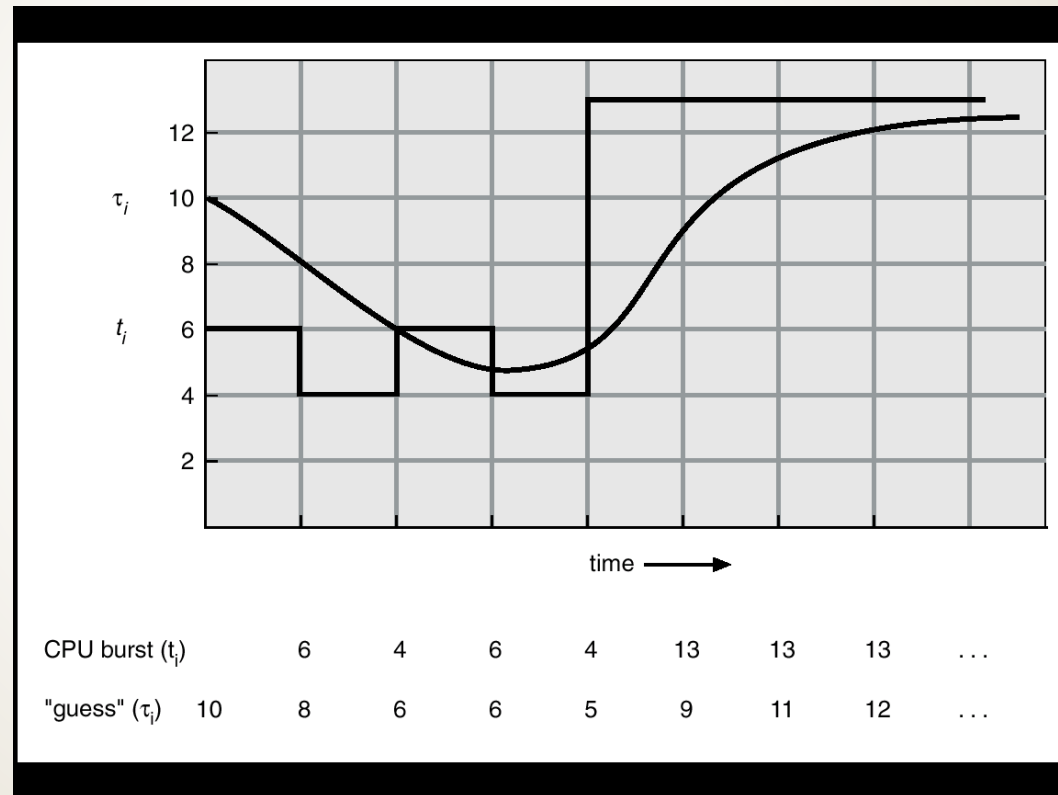
# Determining Length of Next CPU Burst

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.

1.  $t_n$  = actual length of  $n^{\text{th}}$  CPU burst
2.  $\tau_{n+1}$  = predicted value for the next CPU burst
3.  $\alpha, 0 \leq \alpha \leq 1$
4. Define :

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$

# Prediction of the Length of the Next CPU Burst



# Examples of Exponential Averaging

- $\alpha = 0$ 
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count.
- $\alpha = 1$ 
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts.
- If we expand the formula, we get:
  - $\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots$
  - $+ (1 - \alpha)^j \alpha t_{n-j} + \dots$
  - $+ (1 - \alpha)^{n-1} t_n \tau_0$
- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor.



# Priority Scheduling

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





# Round Robin (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 = 20

➤ Process Burst Time

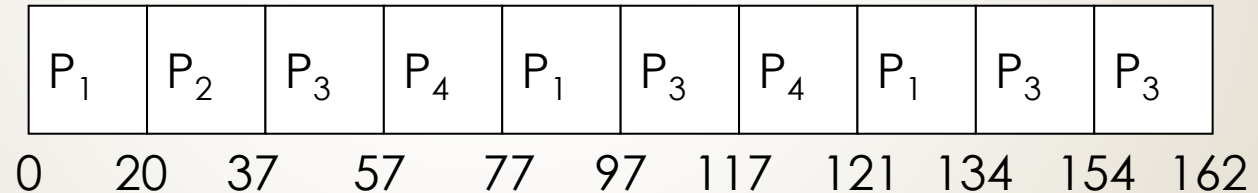
➤ P1 53

➤ P2 17

➤ P3 68

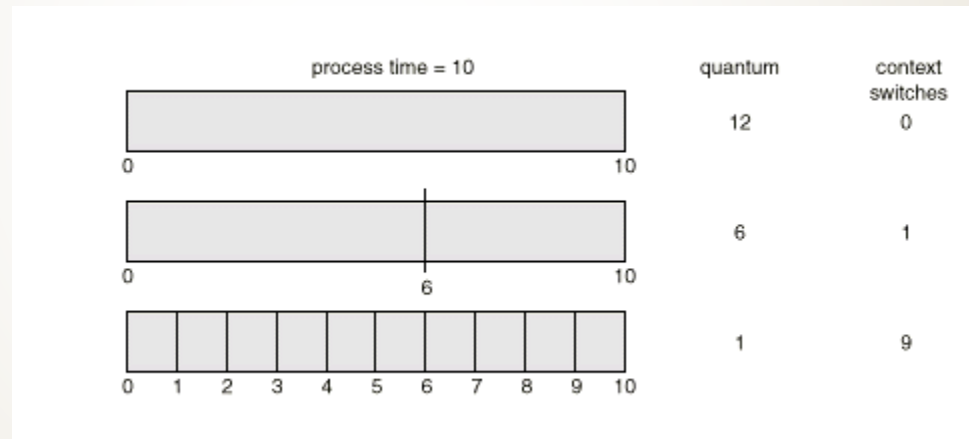
➤ P4 24

➤ The Gantt chart is:

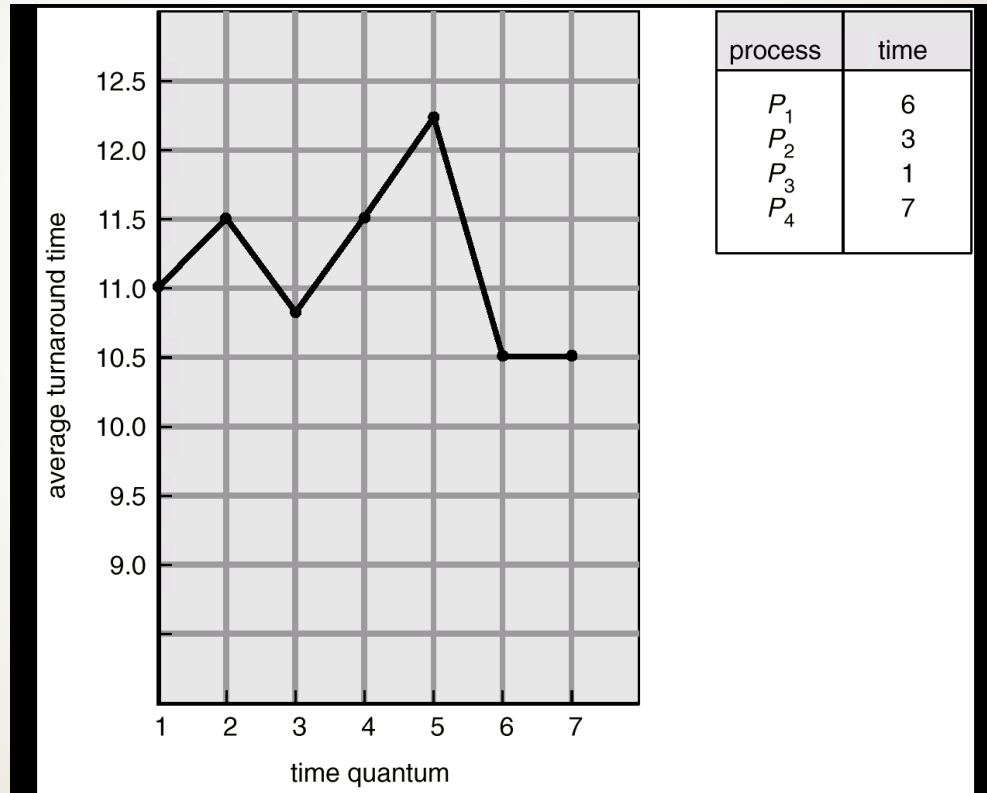


➤ Typically, higher average turnaround than SJF, but better response.

# Time Quantum and Context Switch Time



# Turnaround Time Varies With The Time Quantum

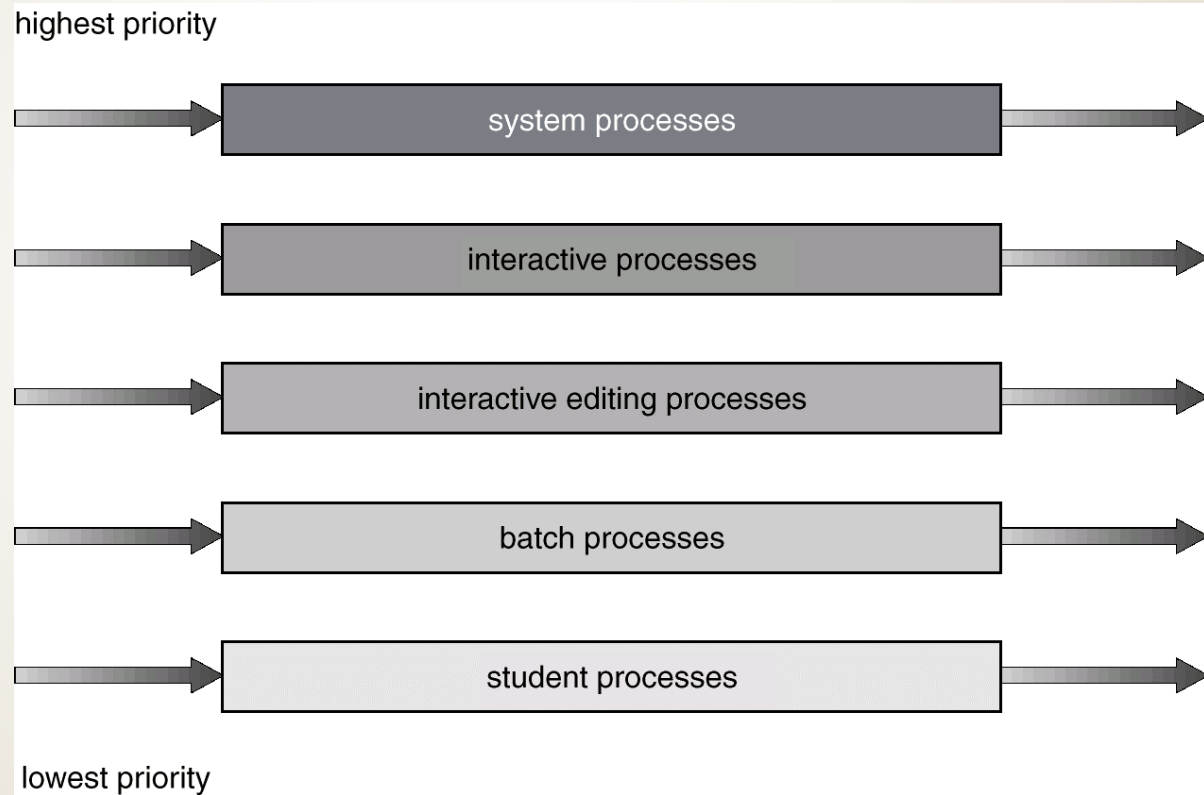




# Multilevel Queue

- 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





# 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 of Multilevel Feedback Queue

- Three queues:

- Q0 – time quantum 8 milliseconds

- Q1 – time quantum 16 milliseconds

- Q2 – FCFS

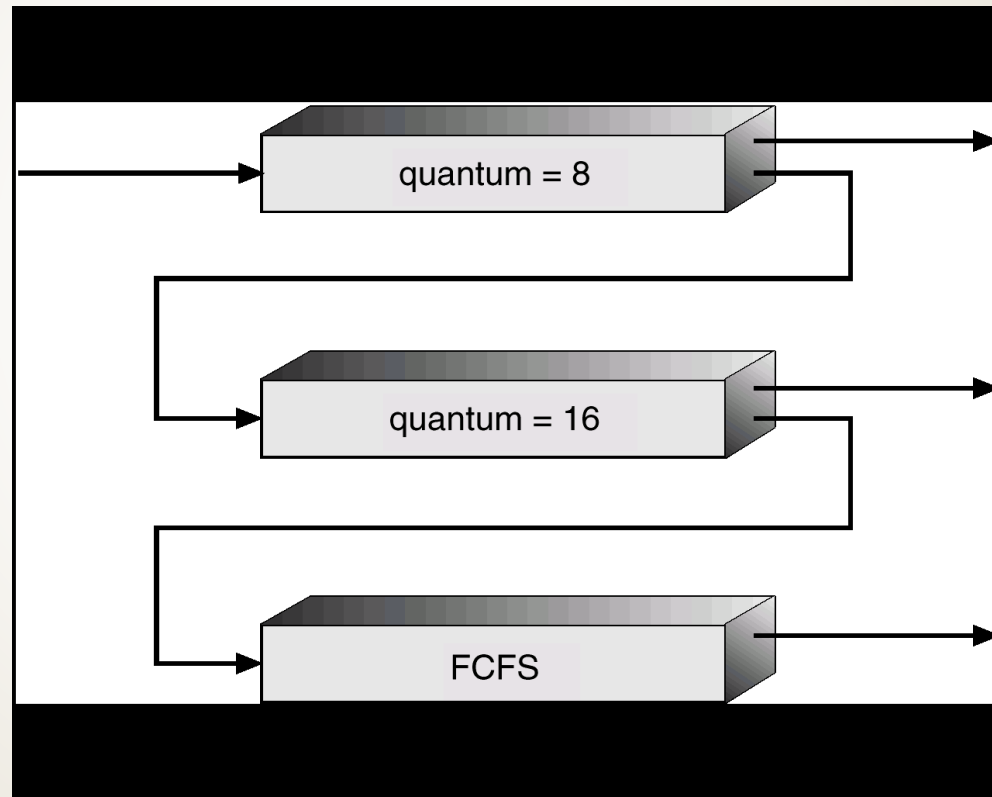
- Scheduling

- A new job enters queue Q0 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 Q1.

- At Q1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q2.



# Multilevel Feedback Queues





# Multiple-Processor Scheduling

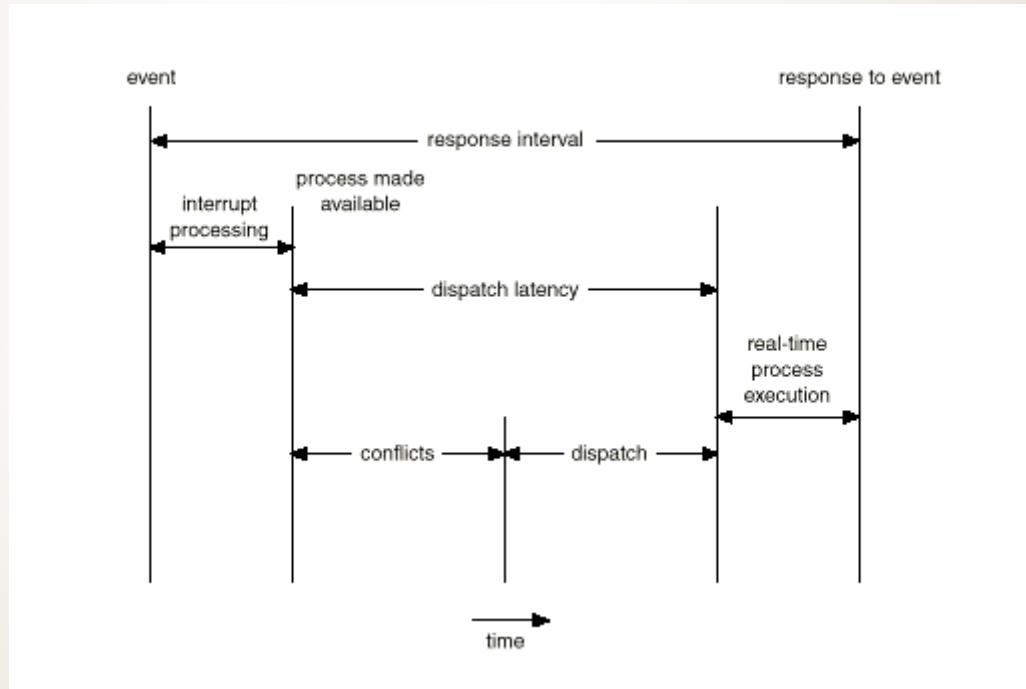
- CPU scheduling more complex when multiple CPUs are available.
- Homogeneous processors within a multiprocessor.
- Load sharing
- Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing.



# Real-Time Scheduling

- **Hard real-time systems** – required to complete a critical task within a guaranteed amount of time.
- **Soft real-time computing** – requires that critical processes receive priority over less fortunate ones.

# Dispatch Latency

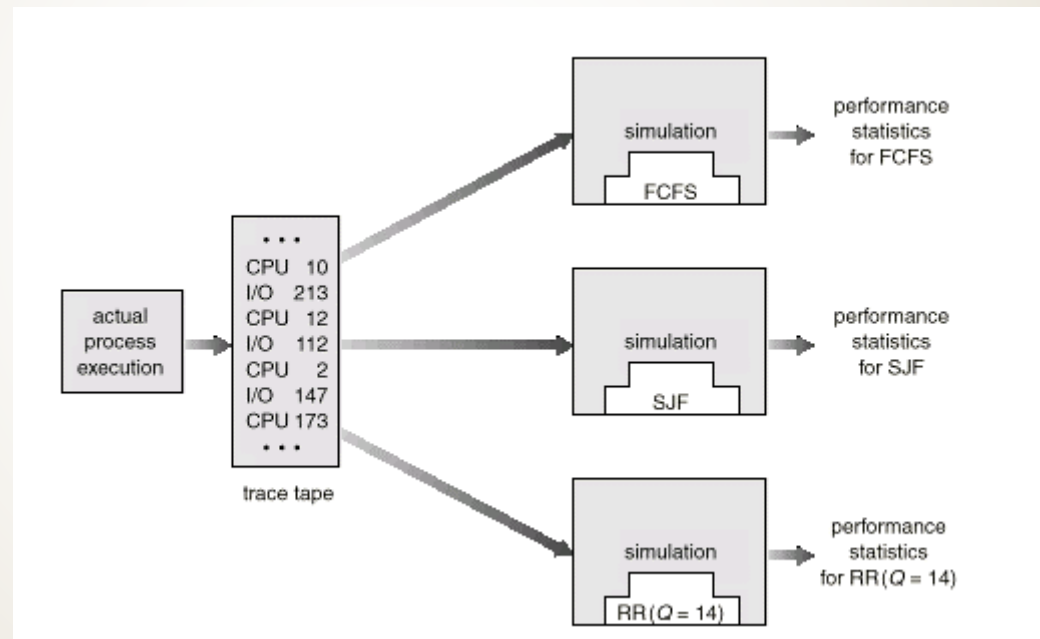




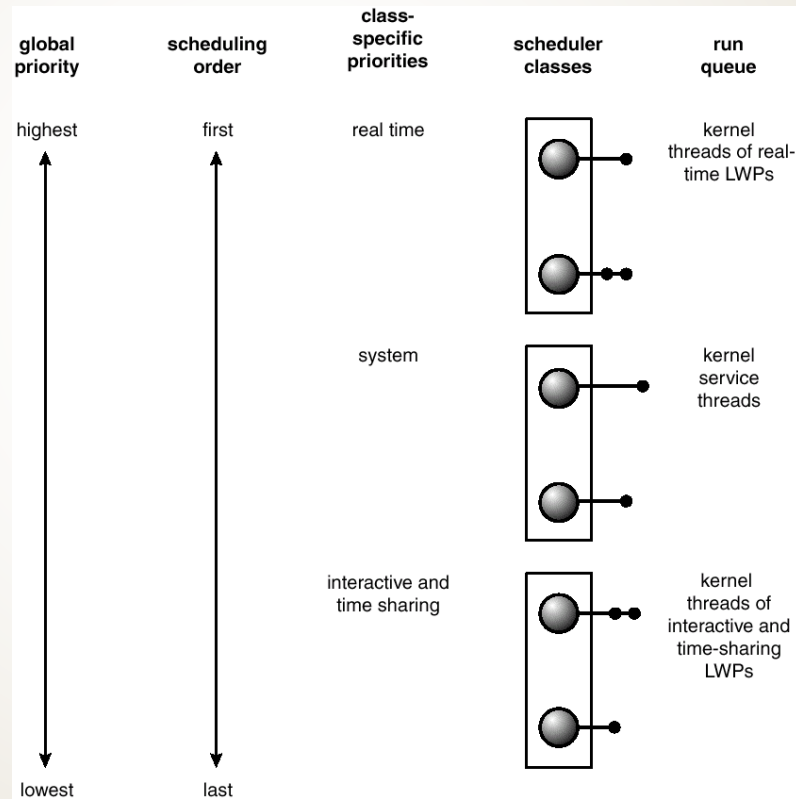
# Algorithm Evaluation

- **Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload.**
- **Queueing models**
- **Implementation**

# Evaluation of CPU Schedulers by Simulation



# Solaris 2 Scheduling



# Windows 2000 Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1





# Deadlocks

- **System Model**
- **Deadlock Characterization**
- **Methods for Handling Deadlocks**
- **Deadlock Prevention**
- **Deadlock Avoidance**
- **Deadlock Detection**
- **Recovery from Deadlock**
- **Combined Approach to Deadlock Handling**

# The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 tape drives.
  - P1 and P2 each hold one tape drive and each needs another one.
- Example
  - semaphores A and B, initialized to 1

➤ P0	P1
➤ wait (A);	wait(B)
➤ wait (B);	wait(A)

## Bridge Crossing Example

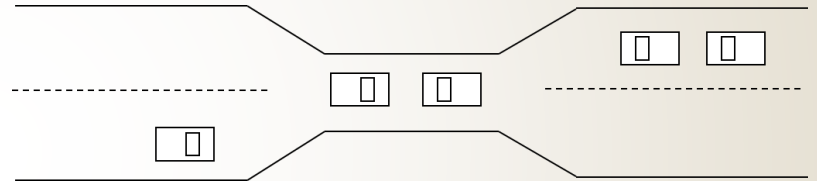
Traffic only in one direction.

Each section of a bridge can be viewed as a resource.

If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).

Several cars may have to be backed up if a deadlock occurs.

Starvation is possible.





# System Model

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



# Deadlock Characterization

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

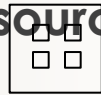
# Resource-Allocation Graph

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

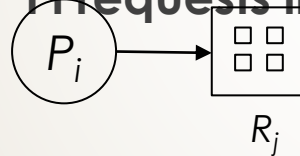
# Resource-Allocation Graph (Cont.)

➤ Process 

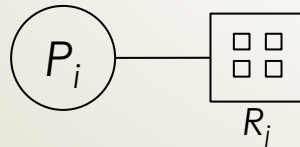
Resource Type with 4 instances



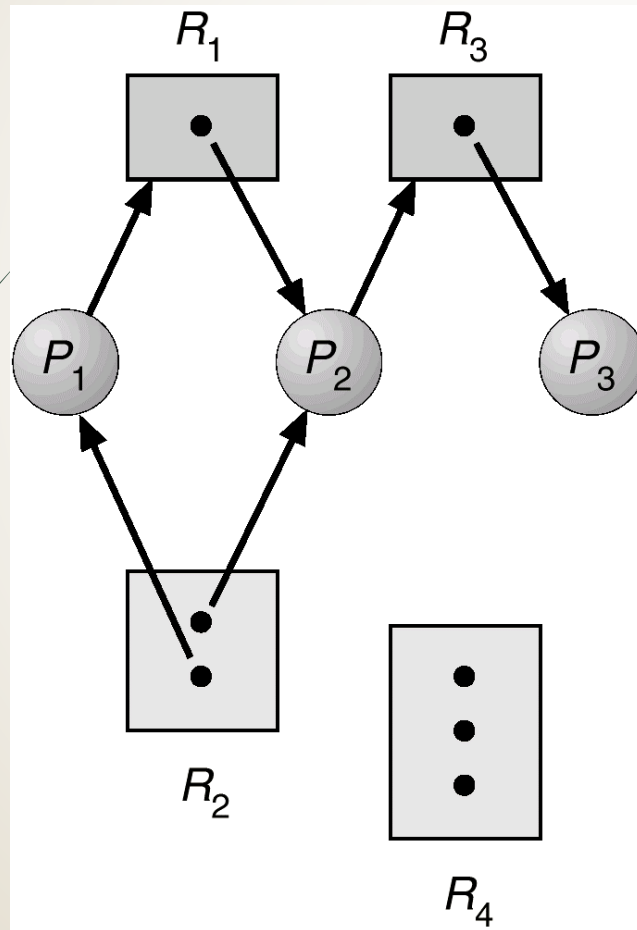
➤  $P_i$  requests instance of  $R_j$



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

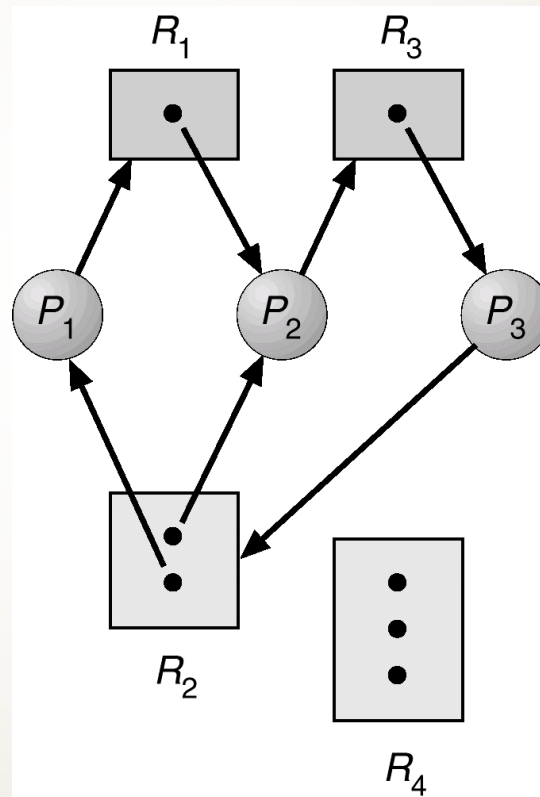


# Example of a Resource Allocation Graph

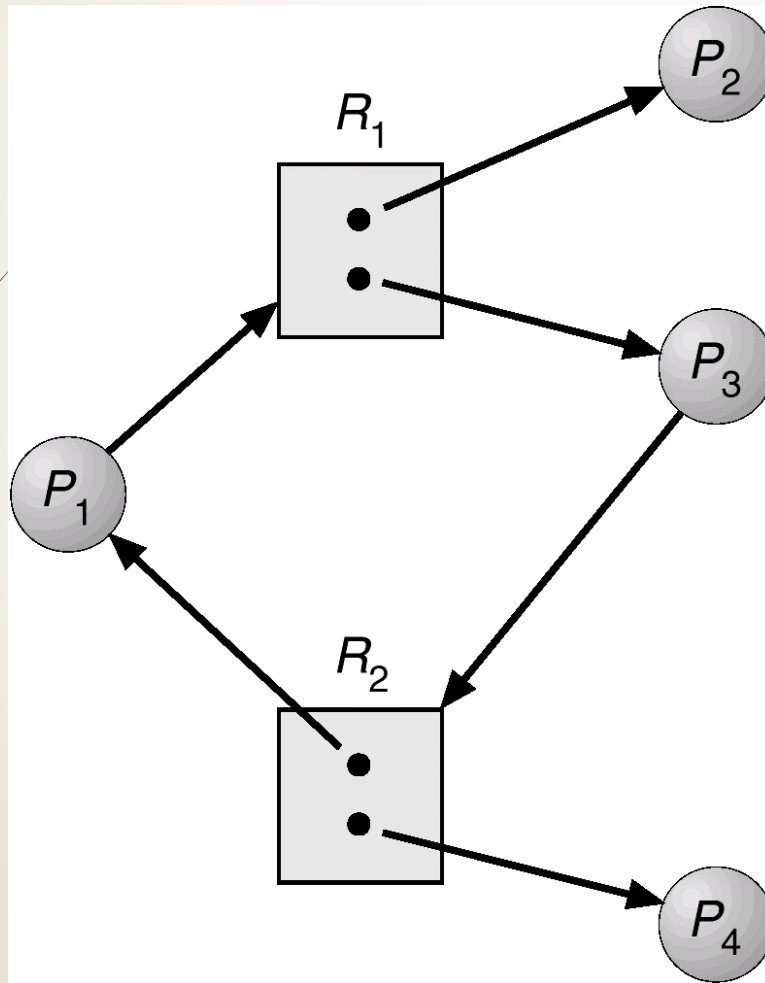




# Resource Allocation Graph With A Deadlock



# Resource Allocation Graph With A Cycle But No Deadlock





# Basic Facts

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



# Methods for Handling Deadlocks

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



# Deadlock Prevention

- **Restrain the ways request can be made.**
- **Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources.**
- **Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.**
  - **Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.**
  - **Low resource utilization; starvation possible.**



# 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

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



# Safe State

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

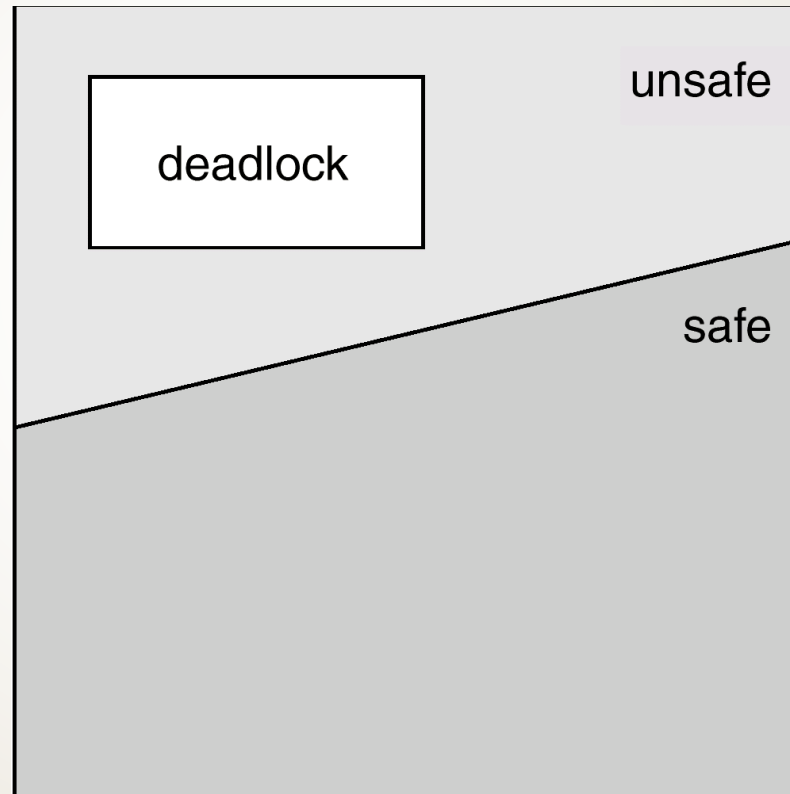




# Basic Facts

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

# Safe, Unsafe , Deadlock State

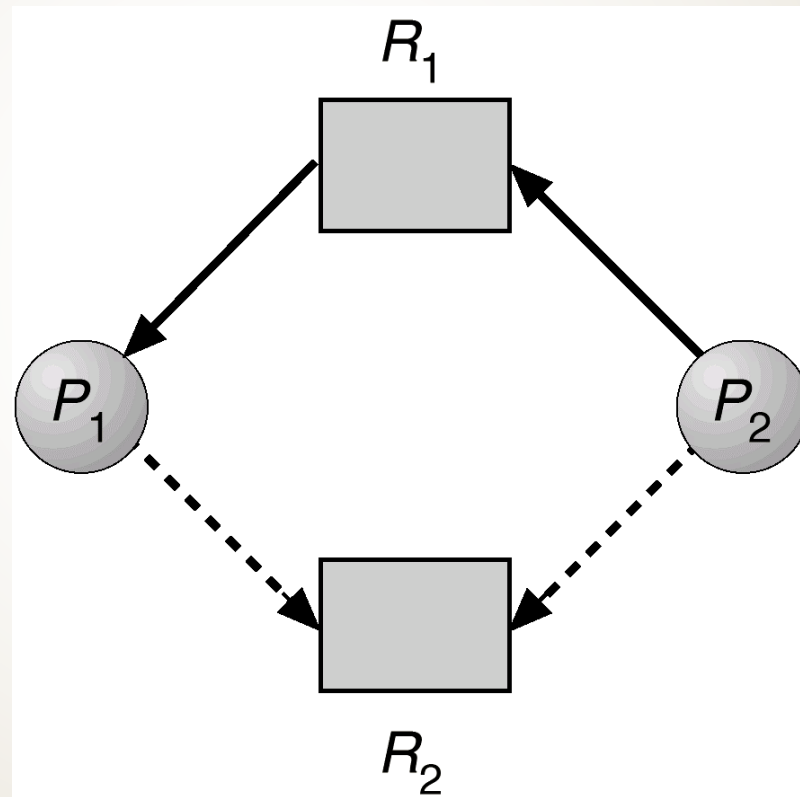




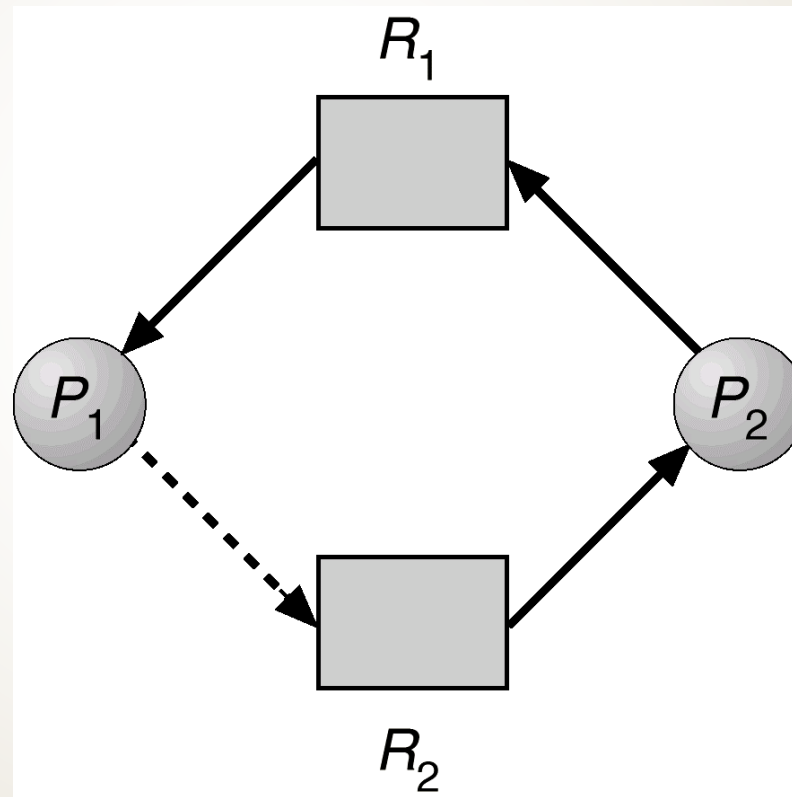
# Resource-Allocation Graph Algorithm

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

# Resource-Allocation Graph For Deadlock Avoidance



# Unsafe State In Resource-Allocation Graph





# Banker's Algorithm

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

# Data Structures for the Banker's Algorithm

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

# Safety Algorithm

- 1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively. Initialize:
  - **Work** = Available
  - **Finish** [i] = false for  $i = 1, 3, \dots, n$ .
- 2. Find and  $i$  such that both:
  - (a) **Finish** [i] = false
  - (b)  $\text{Need}_i \leq \text{Work}$
  - If no such  $i$  exists, go to step 4.
- 3. **Work** = **Work** + **Allocation** <sub>$i$</sub>   
**Finish**[i] = true  
go to step 2.
- 4. If **Finish** [i] == true for all  $i$ , then the system is in a safe state.



# Resource-Request Algorithm for Process $P_i$

- Request = request vector for process  $P_i$ . If  $Request_i[j] = k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$ .
  - 1. If  $Request_i \leq Need_i$  go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
  - 2. If  $Request_i \leq Available$ , go to step 3. Otherwise  $P_i$  must wait, since resources are not available.
  - 3. Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:
    - Available = Available - Request<sub>i</sub>;
    - Allocation<sub>i</sub> = Allocation<sub>i</sub> + Request<sub>i</sub>;
    - Need<sub>i</sub> = Need<sub>i</sub> - Request<sub>i</sub>;;
  - If safe  $\Rightarrow$  the resources are allocated to  $P_i$ .
  - If unsafe  $\Rightarrow$   $P_i$  must wait, and the old resource-allocation state is restored

# Example of Banker's Algorithm

- ▶ 5 processes P0 through P4; 3 resource types A (10 instances), B (5 instances), and C (7 instances).
- ▶ Snapshot at time T0:

	Allocation	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

# Example (Cont.)

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

- Need

- A B C

- P0 7 4 3

- P1 1 2 2

- P2 6 0 0

- P3 0 1 1

- P4 4 3 1

- The system is in a safe state since the sequence  $\langle P1, P3, P4, P2, P0 \rangle$  satisfies safety criteria.

# Example P1 Request (1,0,2) (Cont.)


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


	Allocation	Need	Available
	A B C	A B C	A B C
P0	0 1 0	7 4 3	2 3 0
P1	3 0 2	0 2 0	
P2	3 0 1	6 0 0	
P3	2 1 1	0 1 1	
P4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence  $\langle P1, P3, P4, P0, P2 \rangle$  satisfies safety requirement.
- Can request for  $(3,3,0)$  by P4 be granted?
- Can request for  $(0,2,0)$  by P0 be granted?



# Deadlock Detection

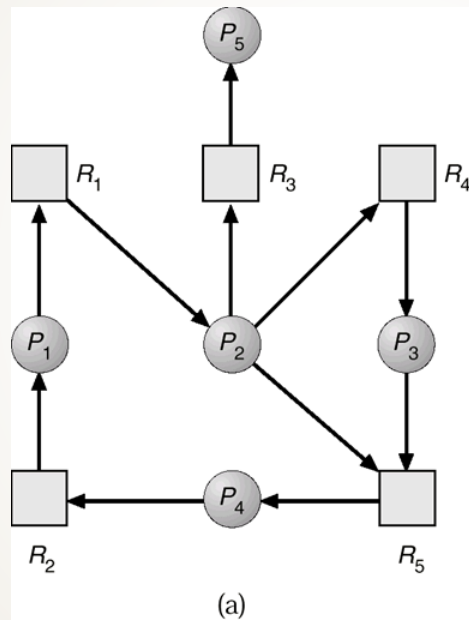
- Allow system to enter deadlock state
  - Detection algorithm
  - Recovery scheme
- 



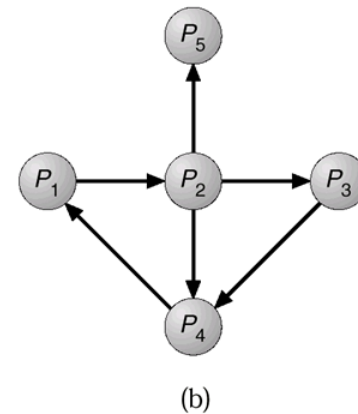
# Single Instance of Each Resource Type

- 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.
- 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  $\text{Request}[ij] = k$ , 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  $\text{Allocation}_i \neq 0$ , then  $\text{Finish}[i] = \text{false}$ ; otherwise,  $\text{Finish}[i] = \text{true}$ .
- 2. Find an index  $i$  such that both:
  - (a)  $\text{Finish}[i] == \text{false}$
  - (b)  $\text{Request}_i \leq \text{Work}$
- 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  $Finish[i] == false$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $Finish[i] == false$ , then  $P_i$  is deadlocked.
- Algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state.

# Example of Detection Algorithm

- Five processes P0 through P4; three resource types A (7 instances), B (2 instances), and C (6 instances).

- Snapshot at time T0:

	Allocation	Request	Available
	A B C	A B C	A B C
➤ P0	0 1 0	0 0 0	0 0 0
➤ P1	2 0 0	2 0 2	
➤ P2	3 0 3	0 0 0	
➤ P3	2 1 1	1 0 0	
➤ P4	0 0 2	0 0 2	

- Sequence <P0, P2, P3, P1, P4> will result in Finish[i] = true for all i.

# Example (Cont.)

- P2 requests an additional instance of type C.

Request

A B C

P0 0 0 0

P1 2 0 1

P2 0 0 1

P3 1 0 0

P4 0 0 2

- State of system?

- Can reclaim resources held by process P0, but insufficient resources to fulfill other processes; requests.
- Deadlock exists, consisting of processes P1, P2, P3, and P4.



# 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.**

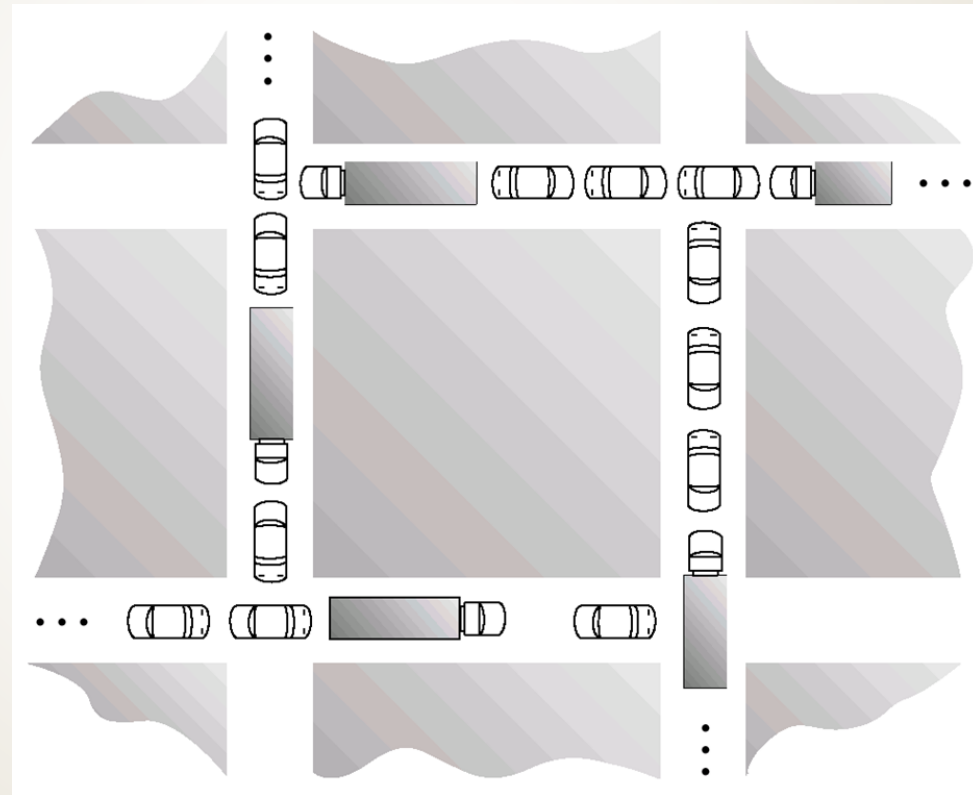


# Combined Approach to Deadlock Handling

- **Combine the three basic approaches**
  - prevention
  - avoidance
  - detection
- **allowing the use of the optimal approach for each of resources in the system.**
- **Partition resources into hierarchically ordered classes.**
- **Use most appropriate technique for handling deadlocks within each class.**



# Traffic Deadlock for Exercise 8.4





# Thank you

The content in this Material are from the Textbooks  
and Reference books given in the Syllabus