BITS, PILANI – K. K. BIRLA GOA CAMPUS

Operating Systems

by

Mrs. Shubhangi Gawali

Dept. of CS and IS

Lecture No. 20:

Handling Deadlocks

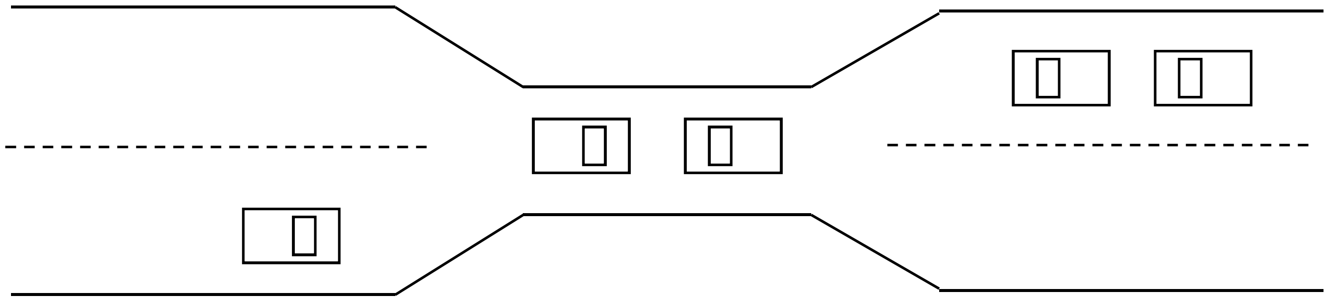
The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- 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

Examples

- Example 1
  - System has 2 disk drives
  - $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ each hold one disk drive and each needs another one

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
- Note – Most OSes do not prevent or deal with deadlocks

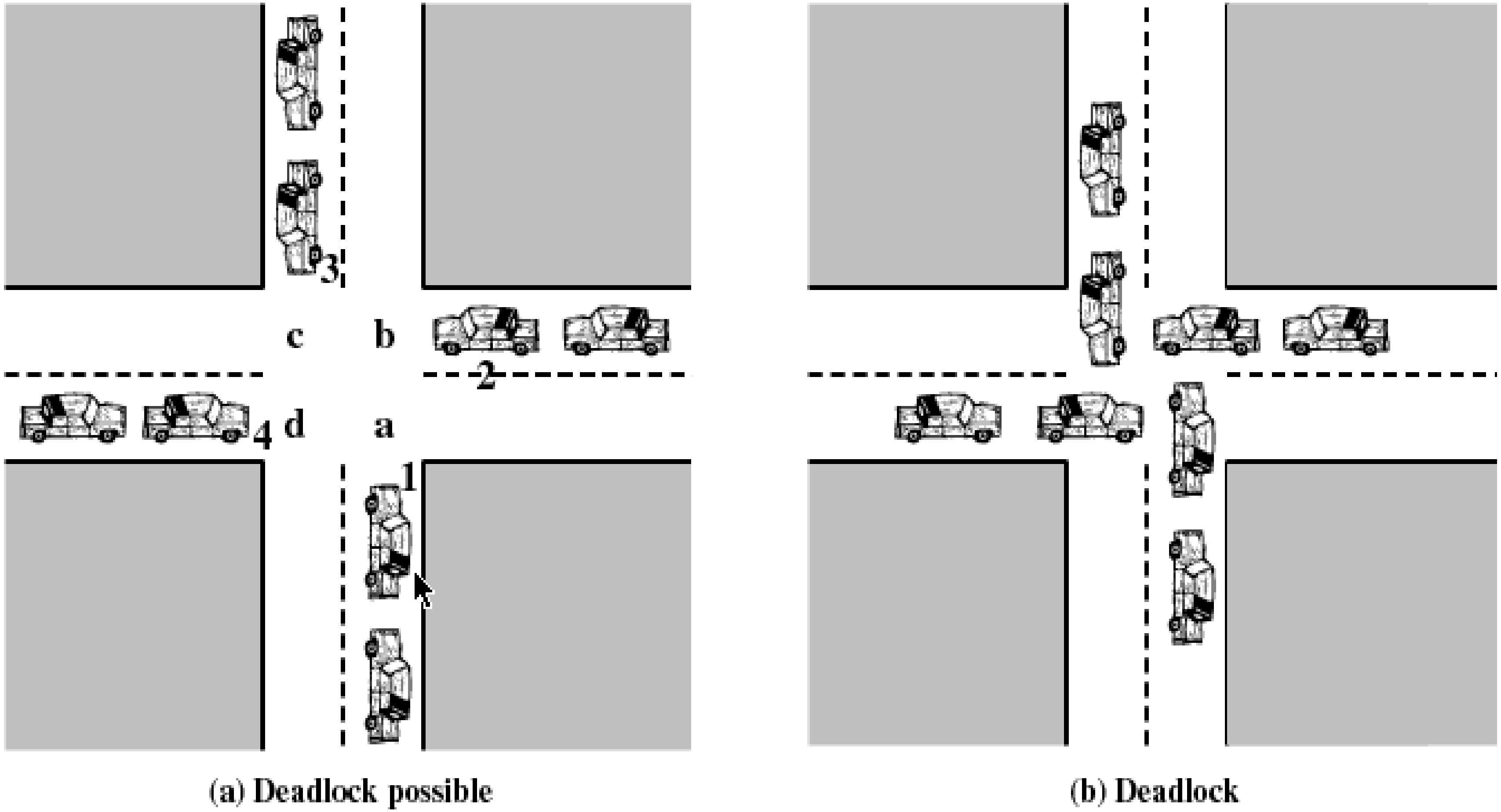


Figure 6.1 Illustration of Deadlock

System Model

- Processes P1, P2, …, Pn
- Resource types $\boldsymbol{R}_1, \boldsymbol{R}_2, \ldots, \boldsymbol{R}_m$

  CPU cycles, memory space, I/O devices
- Each resource type $\boldsymbol{R}_i$ has $\boldsymbol{W}_i$ instances.

NOTICE for IS students

CG T1 papers will be distributed today
at 1:00 p.m.
in A-604

Resource Allocation

- Request
  - If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
- Use
  - Process operates on resource.
- Release
  - Process release the resource.
- Resources
  - Physical resource -- Printer,Memory space…..
  - Logical resource --- Semaphore,Files…..

Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- Examples: 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

Example of Deadlock

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

P1	P2
. . .	. . .
Request 80 Kbytes;	**Request 70 Kbytes;**
. . .	. . .
Request 60 Kbytes;	**Request 80 Kbytes;**

- 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
- There is no limit on the number of consumable resources of a particular type.

Example of Deadlock

- Deadlock occurs if receive is blocking

P1	P2
. . .	. . .
Receive(P2);	Receive(P1);
. . .	. . .
Send(P2, M1);	Send(P1, M2);

Deadlock Characterization

- **Mutual exclusion:**
  - only one process at a time can use a resource.
  - At least one resource must be in a non sharable mode; that is only one process at a time can use a resource.
  - If any other process request the resource, the requesting process must be delayed until the resource has been released.
- **Hold and wait:**
  - a process **holding** at least one resource is **waiting** to acquire additional resources held by other processes

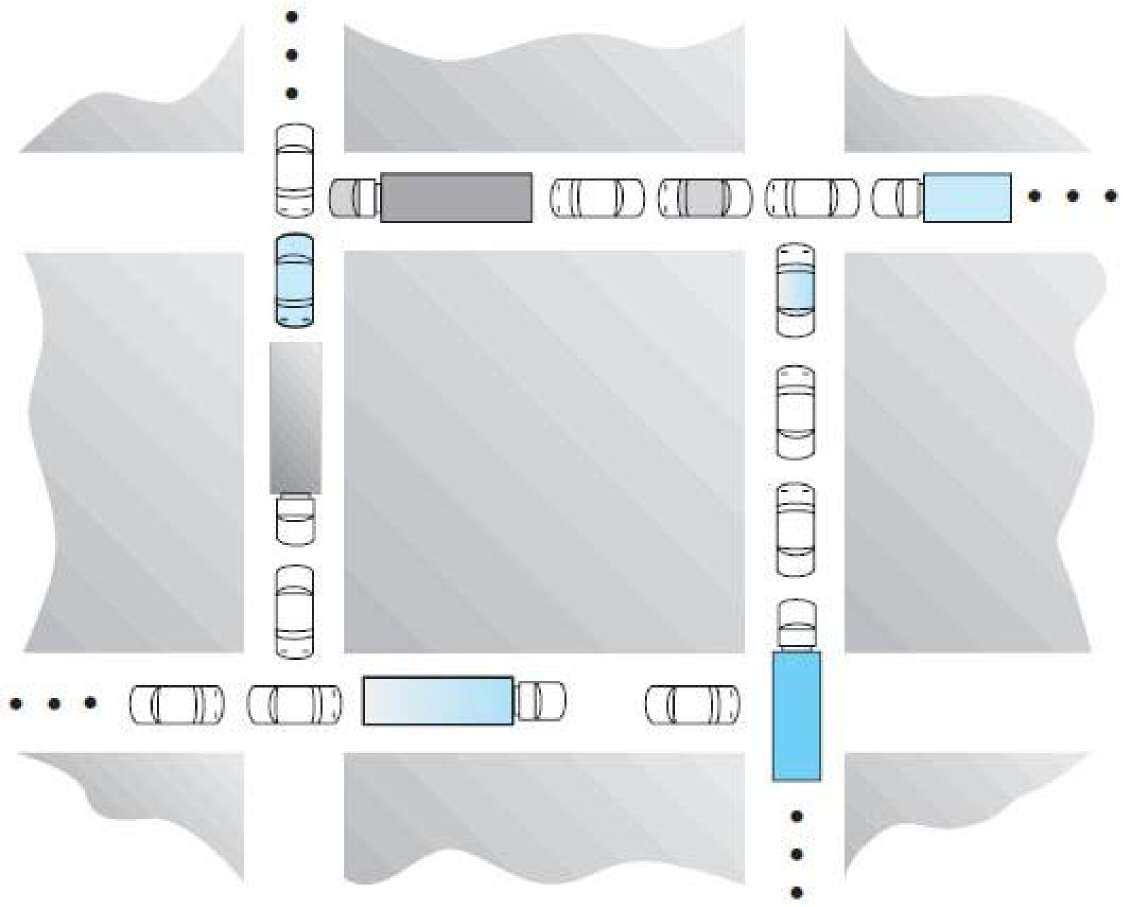
Deadlock Characterization

Deadlock can arise if all four conditions hold simultaneously.

- **No preemption:**
  - a resource can be released only voluntarily by the process holding it, after that process has completed its task .(The resource cannot be preempted).
- **Circular wait:**

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

Traffic Deadlock

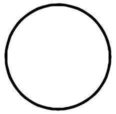


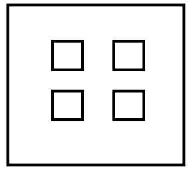
Resource-Allocation Graph

A set of vertices ***V*** and a set of edges ***E***.

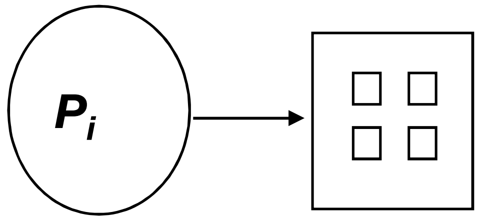
- V is partitioned into two types:
  - $\boldsymbol{P} = \{\boldsymbol{P}_1, \boldsymbol{P}_2, \ldots, \boldsymbol{P}_n\}$, the set consisting of all the processes in the system
  - $\boldsymbol{R} = \{\boldsymbol{R}_1, \boldsymbol{R}_2, \ldots, \boldsymbol{R}_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $\boldsymbol{P}_1 \rightarrow \boldsymbol{R}_j$
- assignment edge – directed edge $\boldsymbol{R}_j \rightarrow \boldsymbol{P}_i$

Resource-Allocation Graph (Cont.)

- Process 

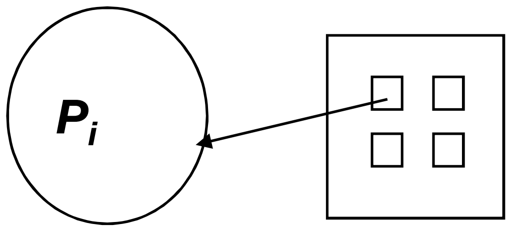
- Resource Type with 4 instances 

- P_i requests instance of R_j



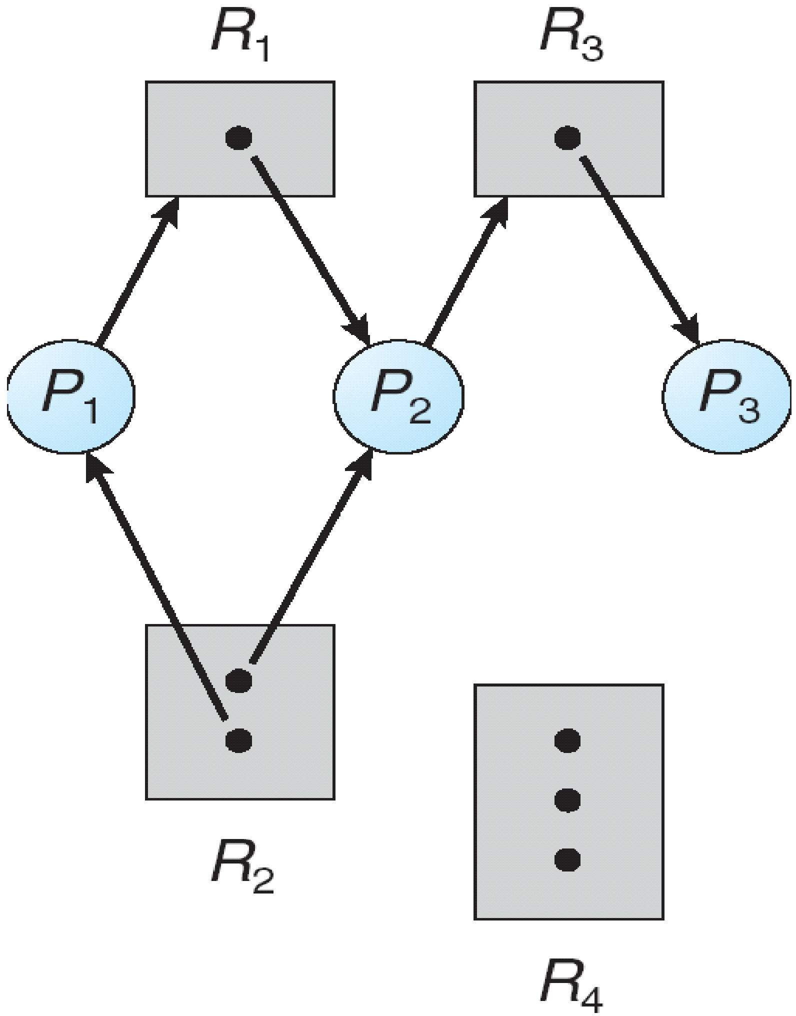
R_j

- P_i is holding an instance of R_j

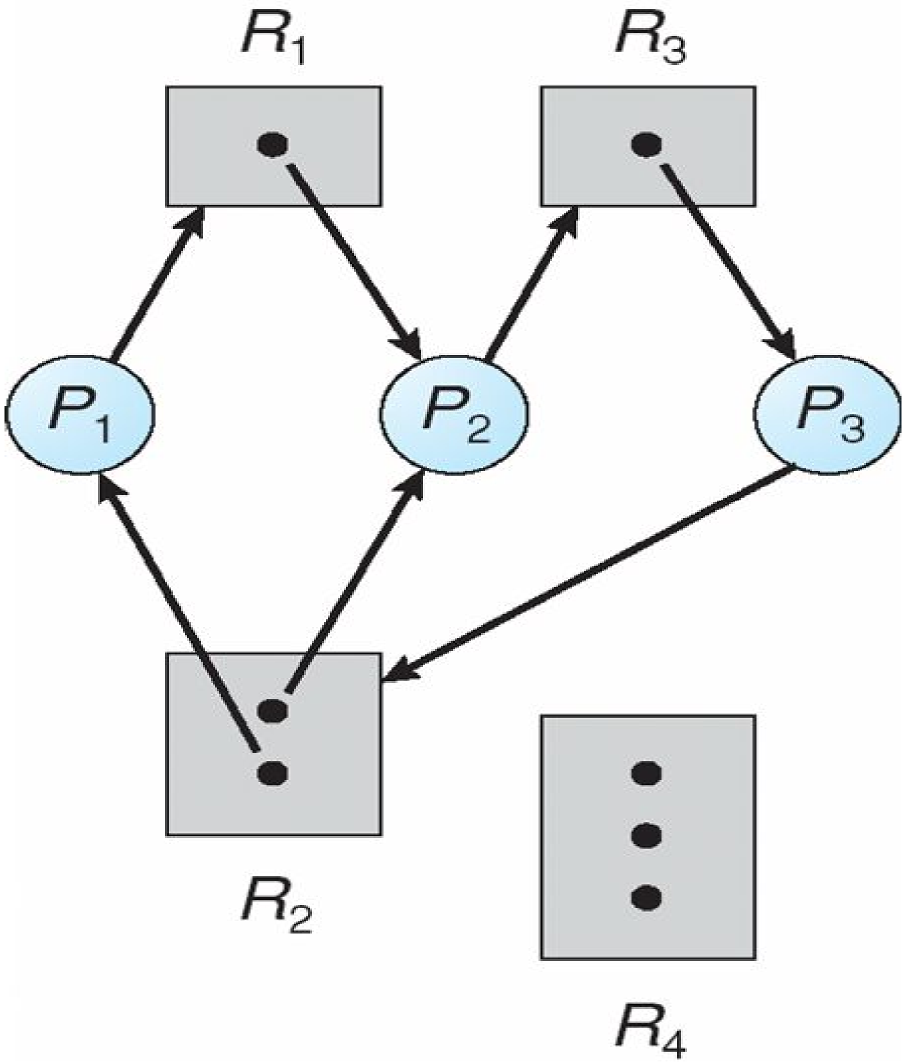


R_j

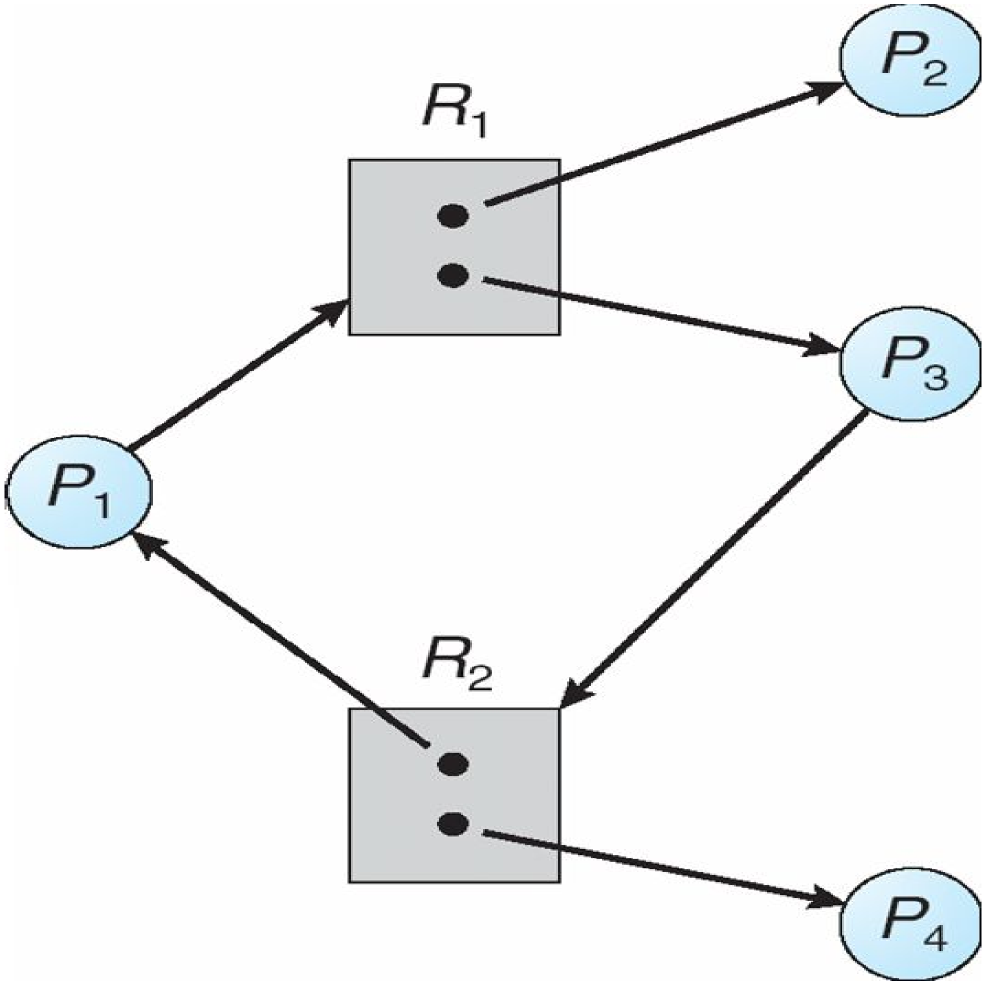
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



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
- Most of the Operating Systems are not handling Deadlock problems
  - It is too costly to prevent / avoid / detect – recover 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

Lecture No. 21:

Handling Deadlocks

Deadlock Prevention Schemes

Deadlock Prevention

- Ensure that at least one of the 4 necessary conditions be prevented.

- **Mutual Exclusion**
  - not required for sharable resources; must hold for non-sharable resources
  - Only one process may use a resource at a time

Deadlock Prevention (Cont.)

- **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.
  - Low resource utilization; **Starvation possible**

Deadlock Prevention (Cont.)

- **Alternative method:**
  - Allow a process to request resources only when the process has no resources.
  - Release all resources before requesting for a new one

- This method is inefficient
  - A process may be held up for a long time waiting for all of its resource request be filled, when it could have proceeded with only some of the resources.
  - A process may not know in advance all of the resources that it will require.

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.
  - Alternate Approach:
    - If a process request some resources,
      - Allot to the process if the resource is free
      - Else, check whether they are allocated to some other process that is waiting for additional resource. If so, preempt the resource from the waiting process and allot them to the requesting process.

Deadlock Prevention (Cont.)

- **Circular Wait**
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

- **Example**
  - Assign a unique integer number to individual resource which allows us to compare two resources and determine one proceed other.
  - Define a one – to – one function F: R→N
  - F(tape drive)=1
  - F(disk drive)=5
  - F(printer) =12
  - Process can request resources only in an increasing order of enumeration.
  - So R(j) is acceptable to a process if and only if F(R(i))< F(R(j)). If F(R(i))> F(R(j)) then release (R(i)) before R(j) entered in.
  - Example: Process needs tape drive and printer must request the tape driver first then printer.

Disadvantage of Deadlock Prevention

- Low device Utilization
- Reduced system throughput.

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 request

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

Avoidance algorithms

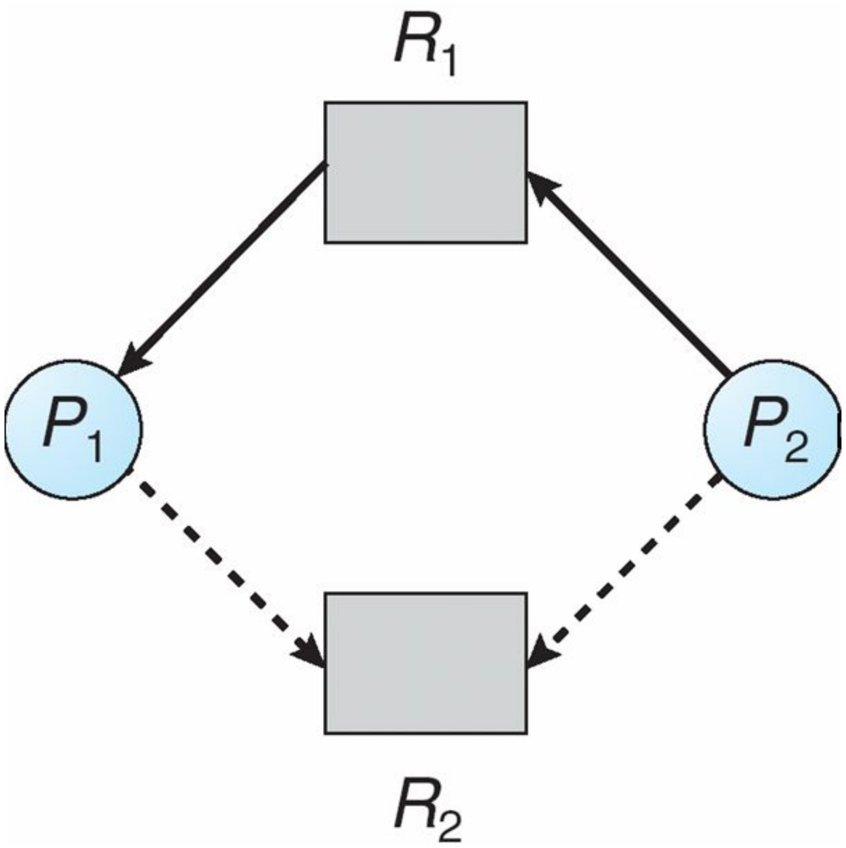
- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the banker's algorithm

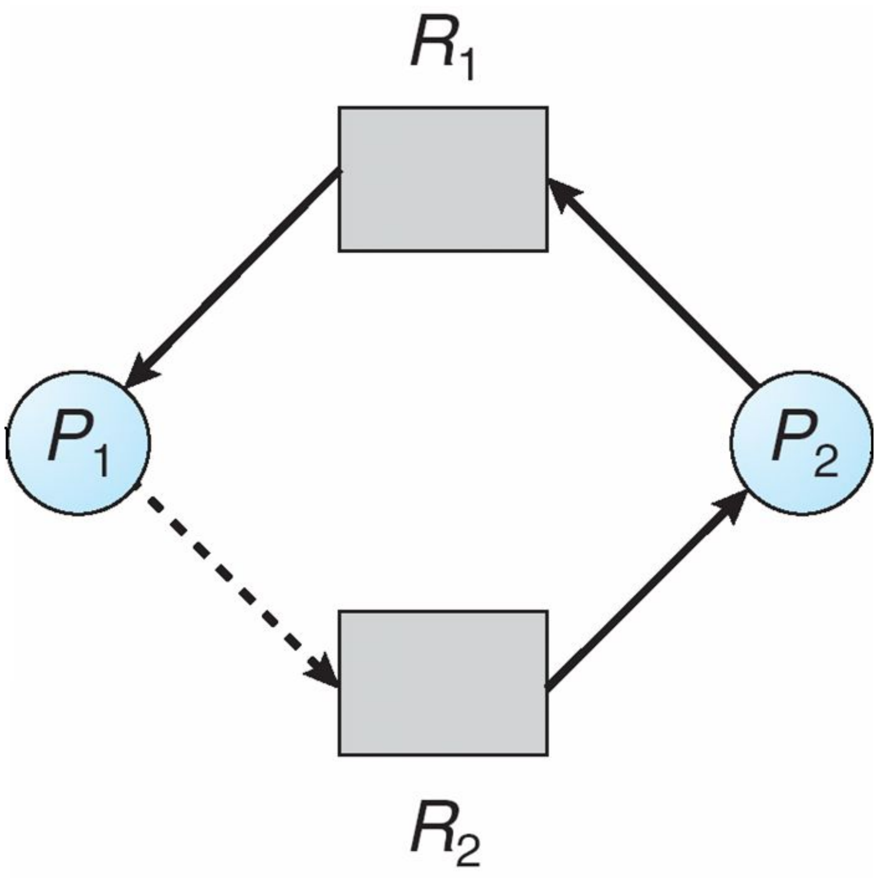
Resource-Allocation Graph Algorithm

- ***3 types of edges***
  - ***Request Edge, Assignment Edge, Claim Edge***
- ***Claim edge*** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed ***a priori*** in the system.
- Request $P_i \rightarrow R_j$ can be granted only if converting the request edge $P_i \rightarrow R_j$ to assignment edge $R_j \rightarrow P_i$ does not result in the formation of cycle in the resource allocation graph.

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

Suppose that process $\boldsymbol{P_i}$ requests a resource $\boldsymbol{R_j}$

- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

- Applicable to multiple instance of each resource type.
- This algorithm is less efficient.
- Each process must a priori claim maximum use.
- When a process requests a resource the system must check whether allocation of these resources will leave the system in safe state.
- 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 $\boldsymbol{n}$ = number of processes, and $\boldsymbol{m}$ = number of resources types.

Available:

- Vector of length $\boldsymbol{m}$. If available $[j] = \boldsymbol{k}$, there are $\boldsymbol{k}$ instances of resource type R_j available

Max:

- $\boldsymbol{n} \times \boldsymbol{m}$ matrix. If $\boldsymbol{Max}\ [i,j] = \boldsymbol{k}$, then process P_i may request at most $\boldsymbol{k}$ instances of resource type R_j

Allocation:

- $\boldsymbol{n}$ x $\boldsymbol{m}$ matrix. If Allocation$[i,j] = \boldsymbol{k}$ then P_i is currently allocated $\boldsymbol{k}$ instances of R_j

Need:

- $\boldsymbol{n}$ x $\boldsymbol{m}$ matrix. If $\boldsymbol{Need}[i,j] = \boldsymbol{k}$, then P_i may need $\boldsymbol{k}$ more instances of R_j to complete its task

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

Example of Banker's Algorithm

- 5 processes P_0 through P_4;
- 3 resource types: ***A*** (10 instances), ***B*** (5instances) and ***C*** (7 instances)

Snapshot at time T_0:

	Allocation	Max	Available
	A B C	***A B C***	***A B C***
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

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

  Need = Max – Allocation

	<u>Need</u>
	A B C
$\boldsymbol{P}_0$	7 4 3
$\boldsymbol{P}_1$	1 2 2
$\boldsymbol{P}_2$	6 0 0
$\boldsymbol{P}_3$	0 1 1
$\boldsymbol{P}_4$	4 3 1

Example of Banker's Algorithm

	Allocation	***Max***	***Need***	***Available***
	A B C	***A B C***	***A B C***	***A B C***
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

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

Safety Algorithm

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

   Work = ***Available***
   Finish [***i***] = ***false*** for ***i*** = 0, 1, …, ***n***- 1

2. Find an ***i*** such that both:
   (a) ***Finish*** [***i***] = ***false***
   (b) $\boldsymbol{Need_i} \leq \boldsymbol{Work}$
   If no such ***i*** exists, go to step 4

3. ***Work*** = ***Work*** + $\boldsymbol{Allocation_i}$
   Finish[***i***] = ***true***
   go to step 2

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

LECTURE NO. 22:

HANDLING DEADLOCKS

Example: P_1 Request (1,0,2)

- Check that Request ≤ Need (that is, (1,0,2) ≤ (1,2,2) ⇒ true
- Check that Request ≤ Available
  (that is, (1,0,2) ≤ (3,3,2) ⇒ true
- ***Snapshot Before allocation***

	Allocation A B C	Need A B C	Available A B C
P_0	0 1 0	7 4 3	3 3 2
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Example: P_1 Request (1,0,2)

- After allocation

	Allocation	Need	Available
	A B C	***A B C***	***A B C***
P_0	0 1 0	7 4 3	3 3 2
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Example: P_1 Request (1,0,2)

- After allocation

	Allocation	Need	Available
	A B C	***A B C***	***A B C***
P_0	0 1 0	7 4 3	3 3 2
P_1	3 0 2	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Example: P_1 Request (1,0,2)

- After allocation

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	3 3 2
P_1	3 0 2	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Example: P_1 Request (1,0,2)

- After allocation

	Allocation	Need	Available
	A B C	**A B C**	**A B C**
P_0	0 1 0	7 4 3	3 3 2
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Example: P_1 Request (1,0,2)

- After allocation

	Allocation	Need	Available
	A B C	***A B C***	***A B C***
P_0	0 1 0	7 4 3	3 3 2
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Example: P_1 Request (1,0,2)

- After allocation

	Allocation	Need	Available
	A B C	***A B C***	***A B C***
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement

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:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

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

Execute resource request algorithm for following

- Can request for (3,3,0) by $\boldsymbol{P}_4$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P}_0$ be granted?

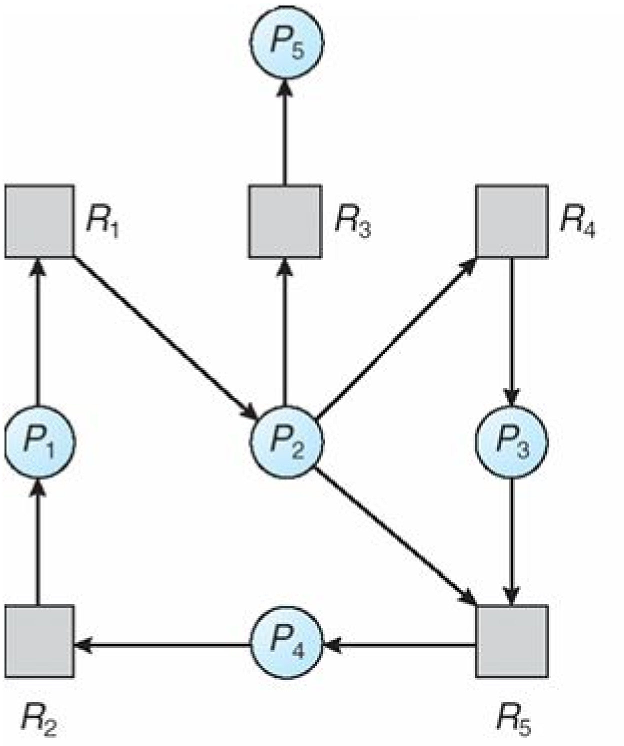
Deadlock Detection

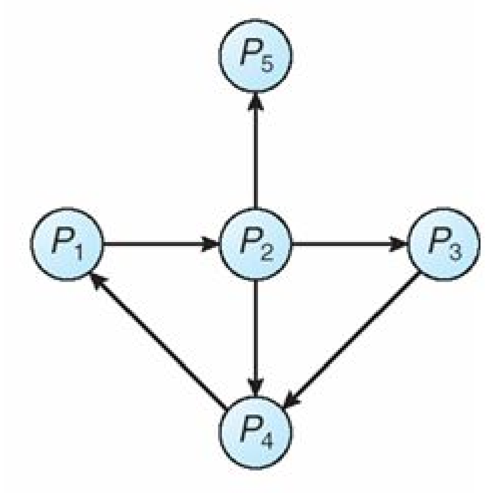
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme
- Includes run time cost of maintaining the necessary information and executing the detection algorithm

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*** x ***m*** matrix defines the number of resources of each type currently allocated to each process

- **Request*:*** An ***n*** x ***m*** matrix indicates the current request of each process. If ***Request*** [i,j] = ***k***, then process P_i is requesting ***k*** more instances of resource type R_j

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:

	Allocation	Request	Available
	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	

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 0
P_3	1 0 0
P_4	0 0 2

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

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:

	Allocation	Request	Available
	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 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	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

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:
   (a) ***Work*** = ***Available***
   (b) For ***i*** = 1,2, …, ***n***, if ***Allocation***$_i \neq 0$, then
   Finish[i] = false;otherwise, ***Finish***[i] = ***true***
2. Find an index ***i*** such that both:
   (a) ***Finish***[***i***] == ***false*** (b) ***Request***$_i \leq$ ***Work***
   If no such ***i*** exists, go to step 4
3. ***Work*** = ***Work*** + ***Allocation***$_i$
   Finish[***i***] = ***true***
   go to step 2
4. If ***Finish***[***i***] == 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* x *n*$^{2)}$ operations to detect whether the system is in deadlocked state

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
- Practical approach about deadlock detection algo.
  - Invoke the algorithm at less frequent intervals
  - Invoke whenever CPU utilization drops below 40%.

Recovery from Deadlock

- 2 methods to recover from deadlock
  - Recover manually (by the operator)
  - Recover automatically

- Automatic recovery
  - Process Termination
  - Resource Preemption

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes (Very expensive)
- Abort one process at a time until the deadlock cycle is eliminated
  - Considerable overhead.
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Types of Resources the process has used
  - No. of 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 –
  - Which resources and from which processes are to be preempted?
  - Determine the order of preemption to minimize cost.
  - Cost factors may include parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.
- Rollback –We must roll back the process to some safe state and restart it from that state.
  - Total rollback
  - Partial rollback
    - this method requires the system to keep more information about the state of all running processes.
    - checkpoint a process periodically

Recovery from Deadlock: Resource Preemption contd…

- Starvation – same process may always be picked as victim
  - ensure that a process can be picked as a victim (preemption) only a (small) finite number of times.
  - The most common solution is to include the number of rollbacks in the 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

Strengths and Weaknesses of the Strategies

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates on-line handling	•Inherent preemption losses