DEADLOCKS

Deadlocks

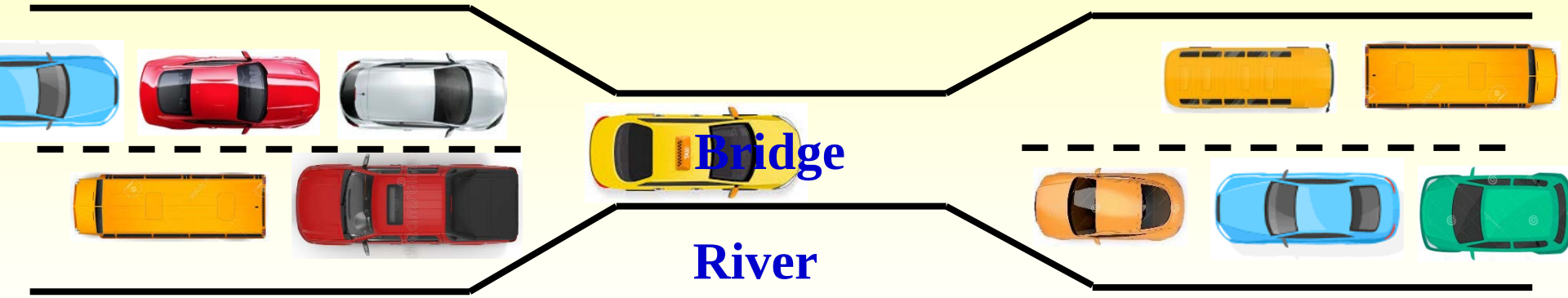
- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

The Deadlock Problem

- **In a computer system deadlocks arise when members of a group of processes which hold resources are blocked indefinitely from access to resources held by other processes within the group.**
- **Example 1:**
  - **A computer system has 2 tape drives.**
  - **P_1 and P_2 each hold one tape drive and each needs another one.**
- **Example 2 : To engage class in any of the 3 available class rooms every Faculty need 3 resources Laptop, MIC and LCD**

Ashok in CR1	**Antony** in CR2	**Akbar** in CR3
has LCD, Laptop	has LCD, MIC	has Laptop, MIC
Waiting for MIC	*Waiting for LT*	*Waiting for LCD*

Example 3: Bridge Crossing

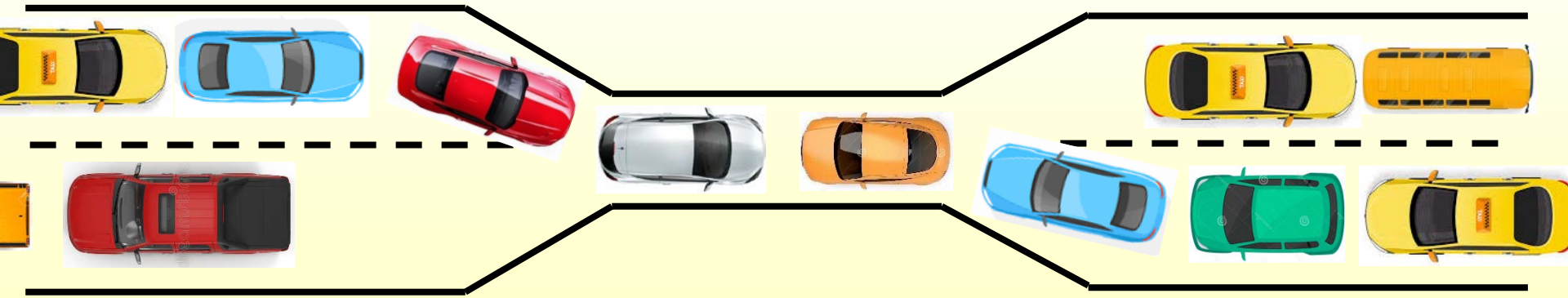


Mulky

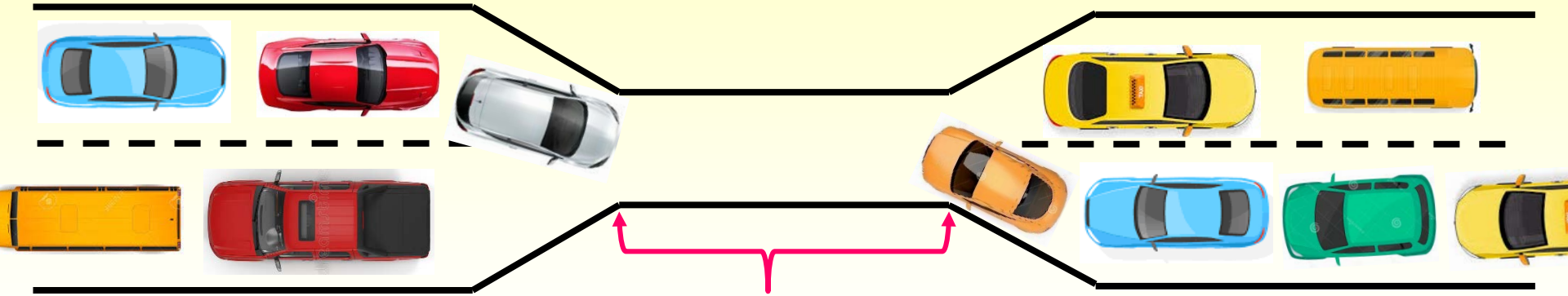
Surathkal

Is everything fine now?

Will it lead to a deadlock situation?



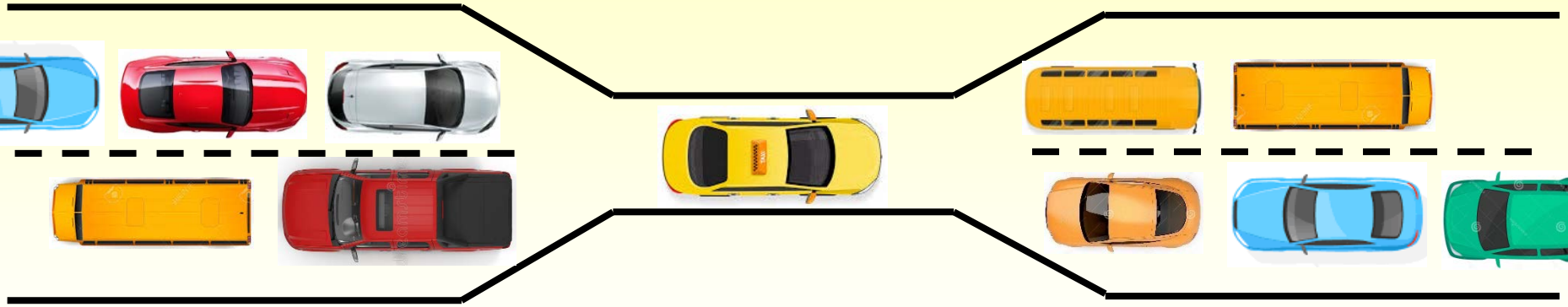
Bridge Crossing Example



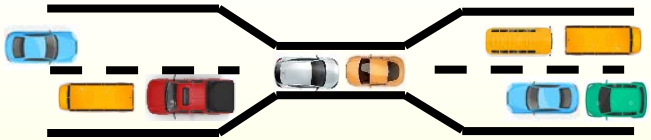
If this mutually exclusive shared resource is accessed by more than one vehicle simultaneously, it will lead to

DEADLOCK

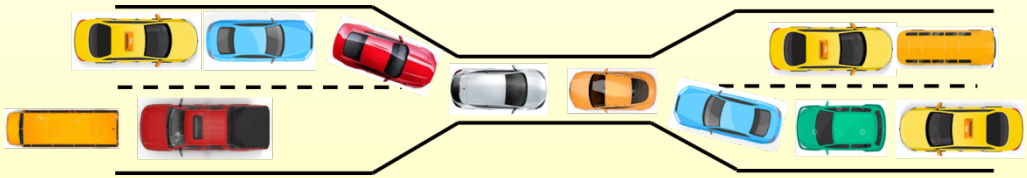
Bridge Crossing Example



- Traffic only in one direction on the bridge.
- 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 $R_1, R_2, \ldots, R_m$

  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

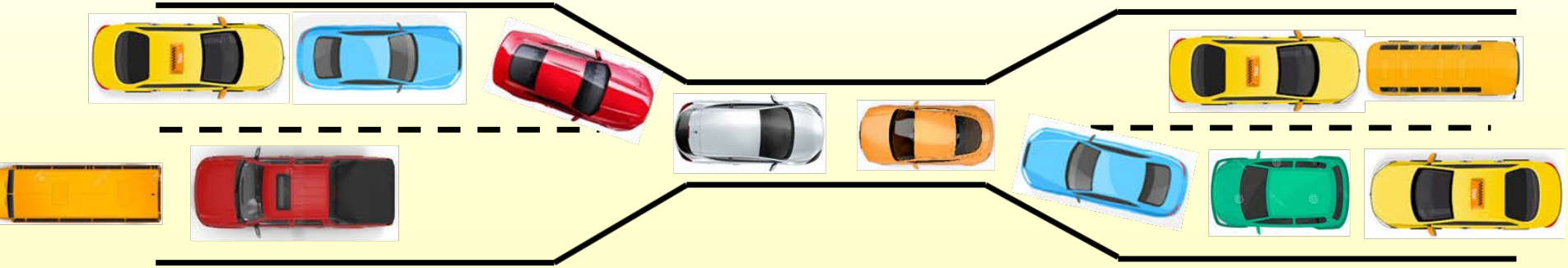
  Release

Deadlock Characterization

Deadlock can arise if following 4 conditions hold simultaneously

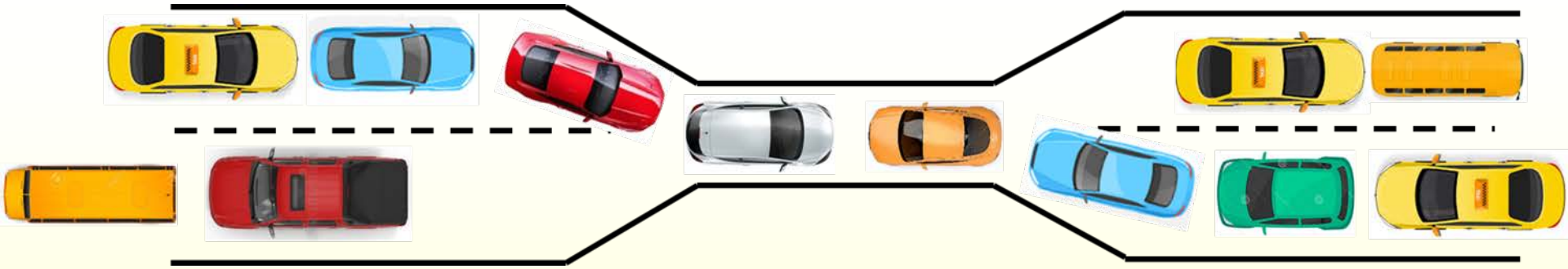
1. **Mutual exclusion: Only one process at a time can use a resource**

2. **Hold and wait: A process holding at least one resource is waiting to acquire additional resources held by other processes**



Deadlock Characterization

3. **No preemption: A resource can be released only voluntarily by the process holding it, after that process has completed its task**

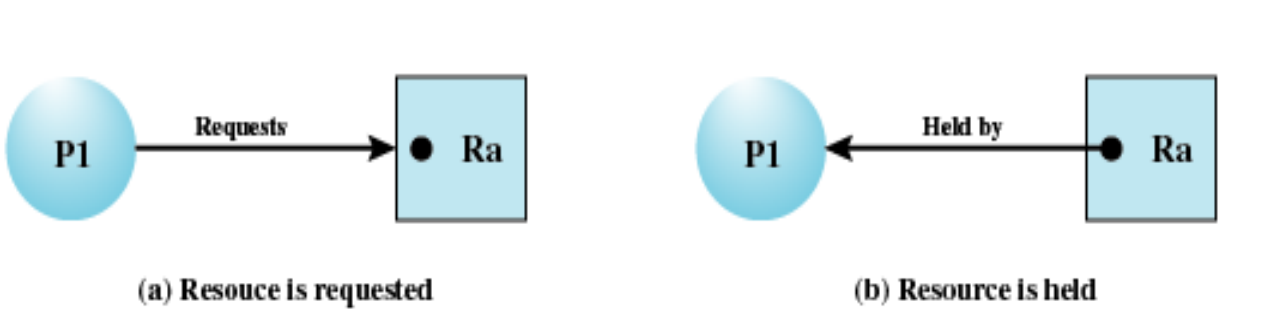


Deadlock Characterization

4. **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0

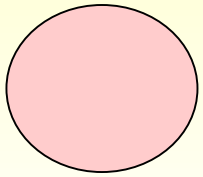
Resource-Allocation Graph

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



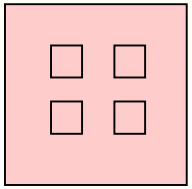
- A set of vertices V and a set of edges E
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- Request edge – directed edge $P_i \rightarrow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

Resource-Allocation Graph (Contd.)

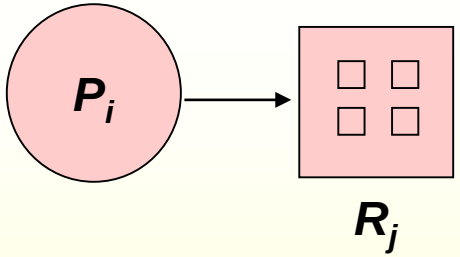


- Process

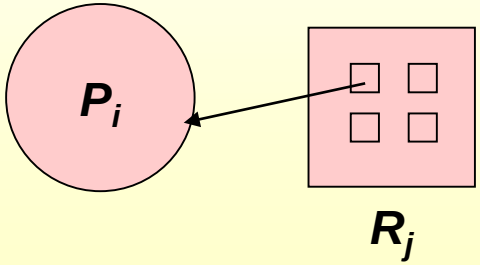
- Resource Type with 4 instances



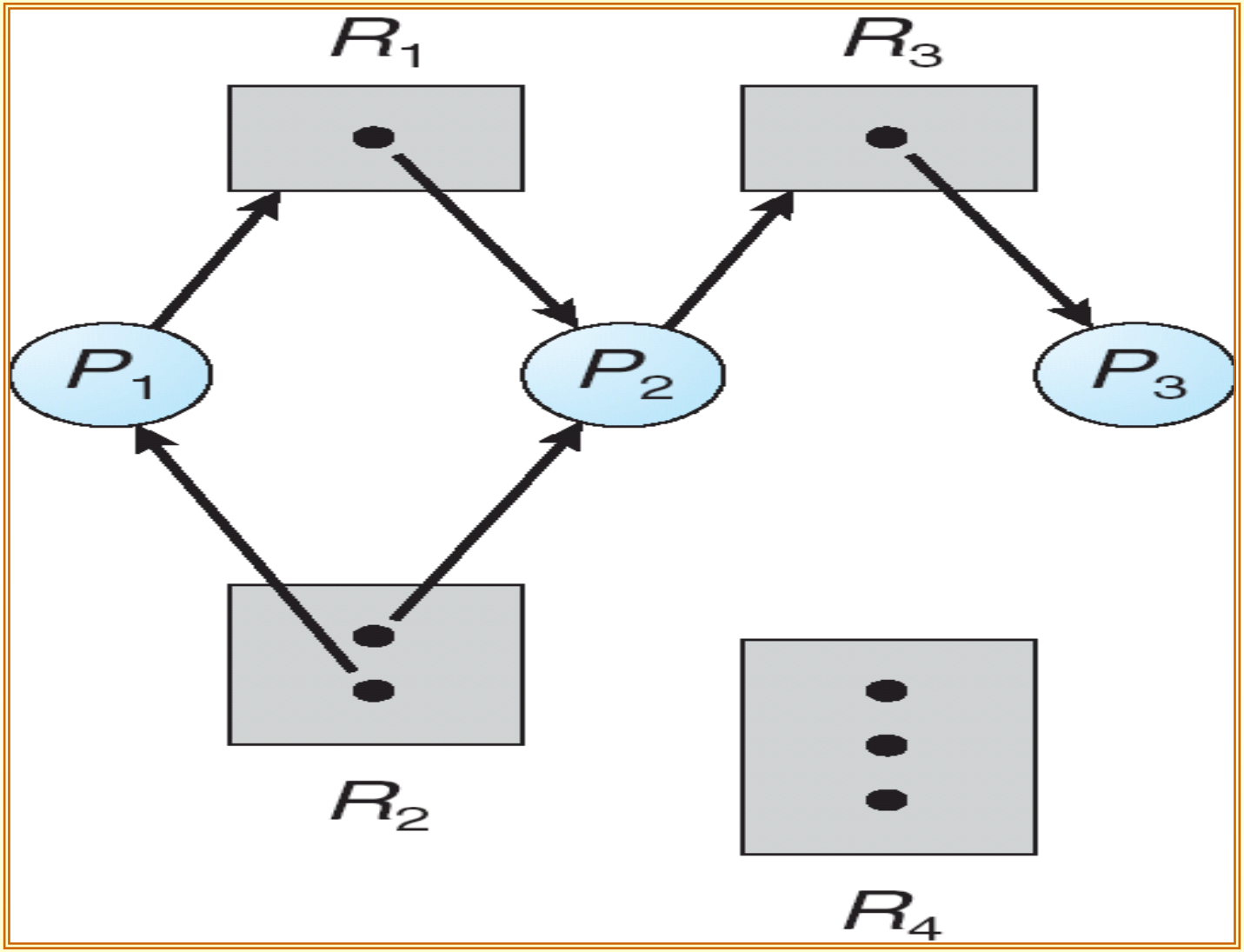
- P_i requests instance of R_j



- P_i is holding an instance of R_j

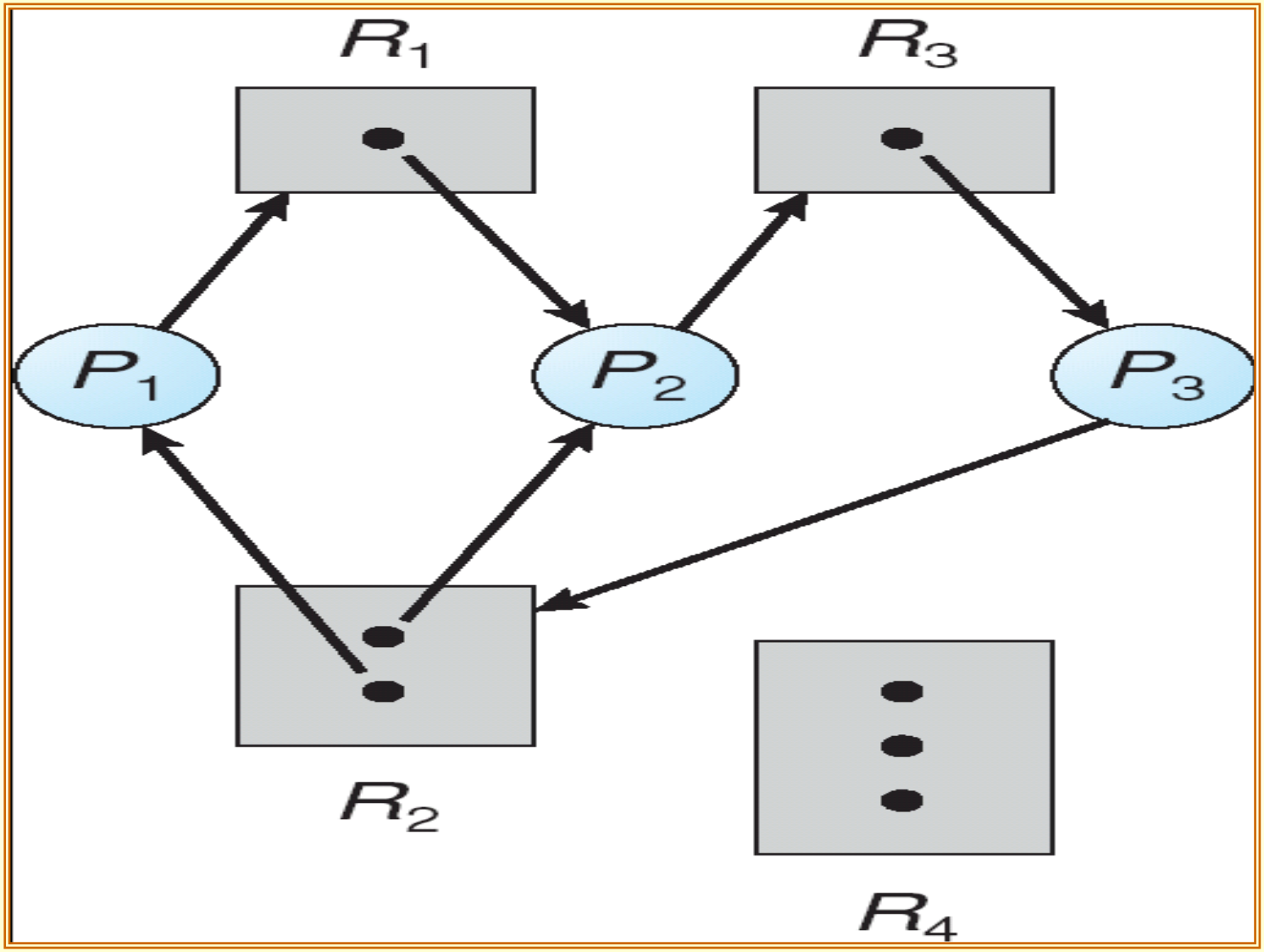


Example of a Resource Allocation Graph



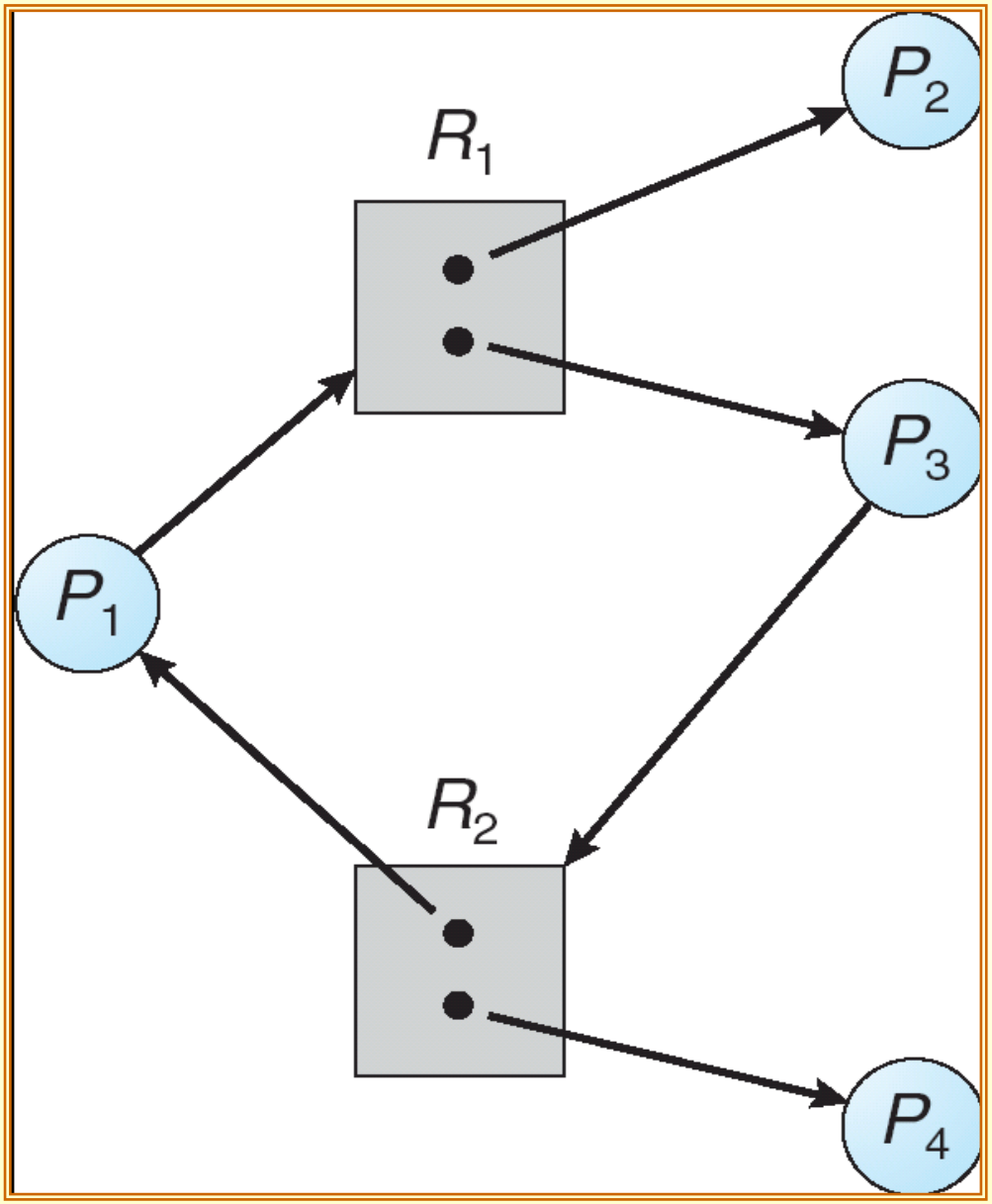
Is it in Deadlock state ? **No**

Example of a Resource Allocation Graph



Is it in Deadlock state ? **Yes**

Resource Allocation Graph with a Cycle



Is it in Deadlock state ?

No

Multiple instances of resources R_1 and R_2. P_2 and P_4 can complete, freeing up resources for P_1 and P_3

Basic Facts

- If graph contains no cycles, then

  no deadlock

- If graph contains a cycle, then
  - if only one instance per resource type, then

    ➡ deadlock

  - if several instances per resource type, then

    ➡ possibility of deadlock

Methods for Handling Deadlocks

- 3 methods for dealing with the deadlock problem:
  1. Ensure that the system will *never* enter a deadlock state
  2. Allow the system to enter a deadlock state and then recover
  3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX and Windows. Application developer need to write programs that handle deadlocks
- To ensure that deadlocks never occur, system can use either
  - **Deadlock prevention**: set of methods for ensuring that at least one of the necessary conditions cannot hold
  - **Deadlock avoidance**: set of methods for giving a prior information to the O.S. regarding resource requirements of a process during its life time

Methods for Handling Deadlocks

- **Deadlock Detection and Recovery**
  - **Grant resource requests when possible, but periodically check for the presence of deadlock and then take action to recover from it**

Deadlock Prevention

- **Mutual Exclusion – not required for sharable resources; must hold for non-sharable resources**
  - **Some resources like printers are intrinsically non sharable**
  - **Read-only files are sharable resource**

In General we can not prevent deadlocks by denying the mutual exclusion condition, because some resources are intrinsically nonsharable

Deadlock Prevention

- **Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources**
  - **Two protocols :**
    - **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**
  - **Disadvantages of these protocols**
    - **Low resource utilization**
    - **Starvation possible**

Deadlock Prevention

- **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)**
  - **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.**
  - **Alternatively**
    - **If a process requests some resources, first check whether they are available, if available then allocate them.**

Deadlock Prevention

- If resources are not available, check whether they are allocated to some other process that is waiting for additional resources.
- If so, preempt the desired resources from the waiting process and allocate then to the requesting process.
- If the resources are not either available or held by a waiting process, the requesting process must wait.
- While it is waiting, some of its resources may be preempted, but only if another process requests them.
- Process can be restarted only when all the requesting resources are allocated to it and recover any resources that were preempted when it was waiting.

Deadlock Prevention

- **Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.**
  - **Let R = { R1, R2, …. Rn } be the set of resource types.**
  - **Assign a unique integer to each resource type, which helps in verifying the ordering.**
  - **A process initially can request any number of instances of a resource type Ri**
  - **After that the process can request instances of resource type Rj if and only if**

    Resource identifier of Rj > Resource identifier of Ri.

Deadlock Avoidance

- **Possible side effects of Deadlock prevention methods are low device utilization and reduced system throughput**
- **Deadlock avoidance requires that the system has some additional *apriori* 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 $<P_1, P_2, \ldots, P_n>$ 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

Safe State - Example

- System has 12 DVD writers: At time t0 the system has the status like

Process	Maximum need	Currently holding
P0	10	5
P1	4	2
P2	9	2

so 3 are free

- Sequence < P1, P0, P2 > is SAFE.
- P1 can be immediately allocated all its drives.
- When P1 returns, available 5 drives can be allotted to P0
- When P0 returns, out of available 10 drives 7 can be allotted to P2

Safe State - Example

- A system can go from Safe state to unsafe state
  - At time t1 if P2 request one more DVD and if it is allotted

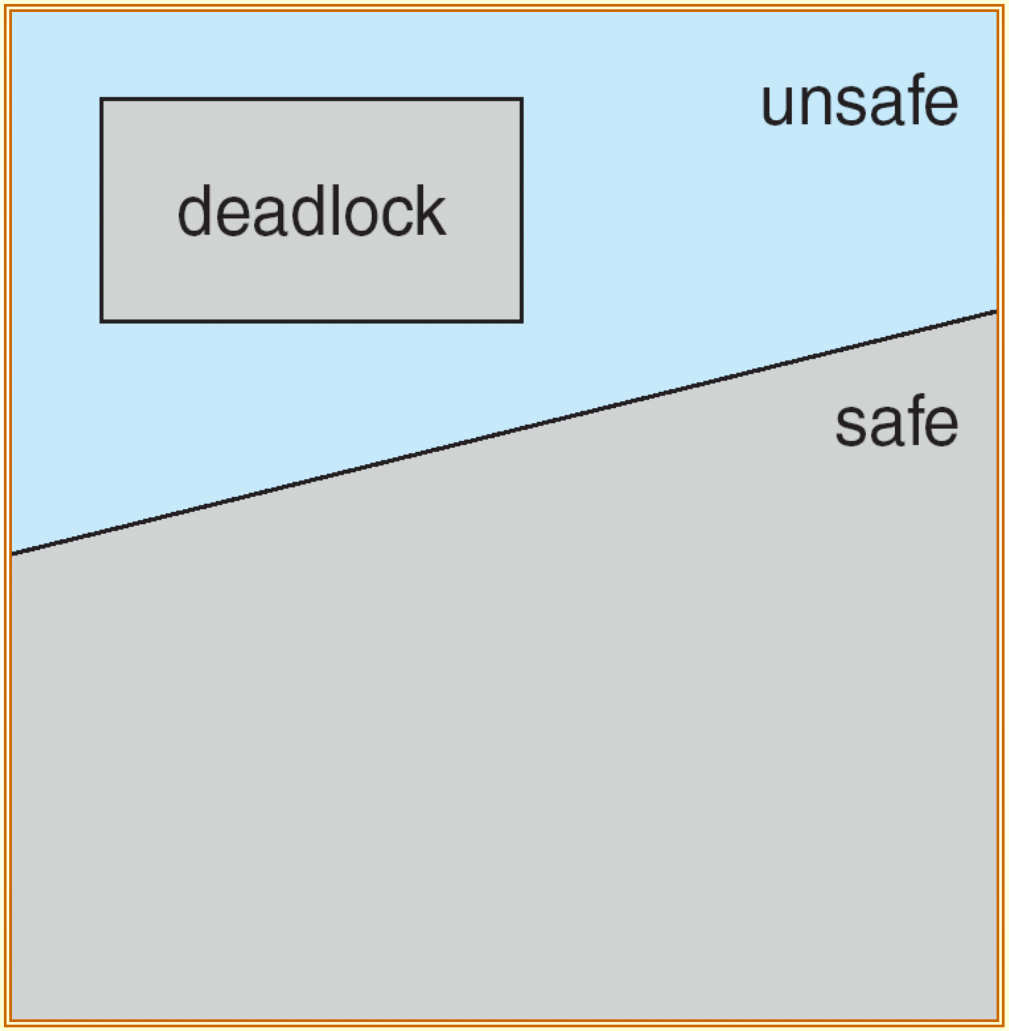
Process	Maximum needs	currently holding	
P0	10	5	
P1	4	2	
P2	9	3	so 2 are free

  - Now the system is not in SAFE state
  - Now only P1 request can be satisfied with remaining 2 drives
  - When P1 returns, system has 4 drives free
  - P0 requires 5 drives it has to wait
  - P2 requires 6 drives it has to wait. This results in Deadlock
- So, whenever a process requests a resource that is currently available, the system must decide whether the resource can be allotted immediately, or it has to wait

Basic Facts

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

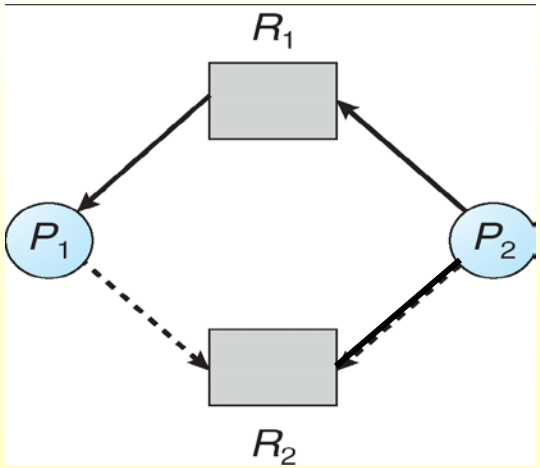
Safe, Unsafe , Deadlock State

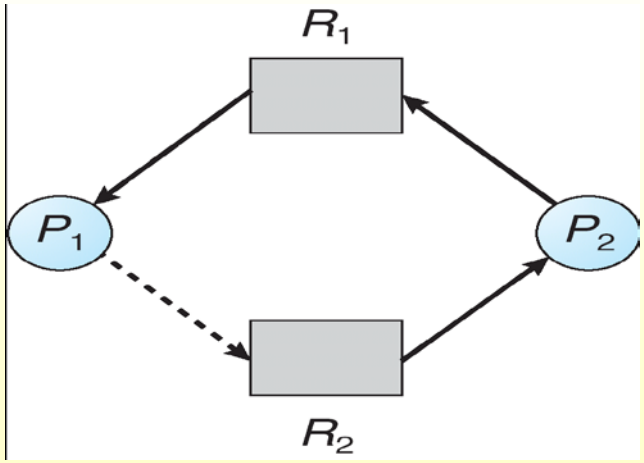


- A safe state is not a deadlock state
- Conversely, a deadlock state is an unsafe state
- Not all unsafe states are deadlocks

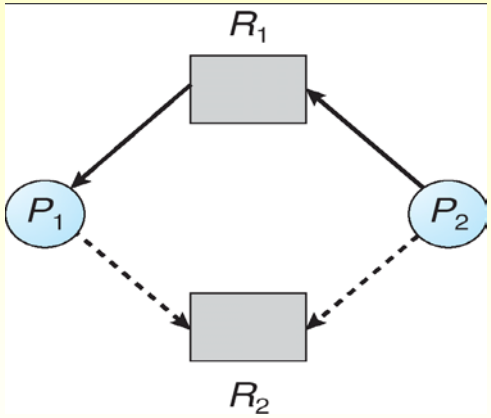
Resource-Allocation Graph Algorithm

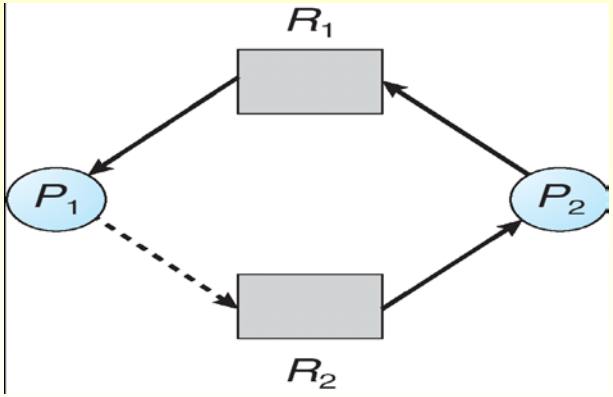
- **Claim edge P_i ---→ R_j indicates that process P_i may request resource R_j , represented by a dashed line**
- **Claim edge converts to request edge when a process requests a resource**
- **When a resource is released by a process, assignment edge reconverts to a claim edge**
- **Resources must be claimed a priori in the system**





Resource-Allocation Graph algorithm for Deadlock Avoidance





- **Unsafe state - will create a cycle**
- **Before starting the process, all Claim edges must appear on graph**
- **Resource is allotted only if converting the Request edge to assignment edge does not result in a Cycle.**
- **RA graph algorithm is not applicable for resources with multiple instances**
- **So, use Banker's algorithm.**

Banker's Algorithm for Deadlock avoidance

- When a new process enters the system it must declare the maximum number of instances of each resource type needed
  - This number may not exceed the total number of resources in the system
- When user requests a set of resources, the system must determine:
  - Whether the allocation of these resources will leave the system in a safe state
  - If it is safe, then the resources are allocated
  - Otherwise, the process must wait until some other process releases enough resources

Data Structures for the Banker's Algorithm

Let n = number of processes, and

m = number of resources types

Data structures needed are:

- **Available: Vector of length m. If Available [j] = k, k instances of resource type R_j are available.**
- **Max: n x m matrix. If Max [i, j] = k, then process P_i may request at most k instances of resource type R_j.**
- **Allocation: n x m matrix. If Allocation[i, j] = k then P_i is currently allocated k instances of R_j.**
- **Need: n x 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] = Max[i, j] – Allocation [i, j]

Safety Algorithm

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

   Initialize: **Work = Available**

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

2. Find an i such that both:

   (a) Finish [i] == false

   (b) $Need_i \leq Work$

   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] == true for all i, then the system is in safe state

Requires an order of m * n^2 operations to decide whether a state is safe

Resource-Request Algorithm for Process P_i

$Request_i$ = 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:**

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

If resulting state is 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 P_0 through P_4; 3 resource types *A* (10 instances), *B* (5 instances) 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 **Need** matrix is defined to be

Max - Allocation

	Allocation	***Max***	***Available***	***Need***	***Work***
			A B C		*A B C*
	A B C	*A B C*	3 3 2	*A B C*	3 3 2
P_0	0 1 0	7 5 3		7 4 3	
P1	2 0 0	3 2 2		1 2 2	5 3 2
P_2	3 0 2	9 0 2		6 0 0	
P_3	2 1 1	2 2 2		0 1 1	7 4 3
P_4	0 0 2	4 3 3		4 3 1	7 4 5

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

Example (Cont.)

P_1 requests (1,0,2). Now $Request_1 \leq Available$ (i.e., (1,0,2) ≤ (3,3,2))

	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 1	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 <P1, P3, P4, P0, P2> (or <P1, P3, P0, P2, P4>) satisfies safety requirement**
- **Can request for (3,3,0) by P4 be granted?** **NO**
- **Can request for (0,2,0) by P0 be granted?** **NO**

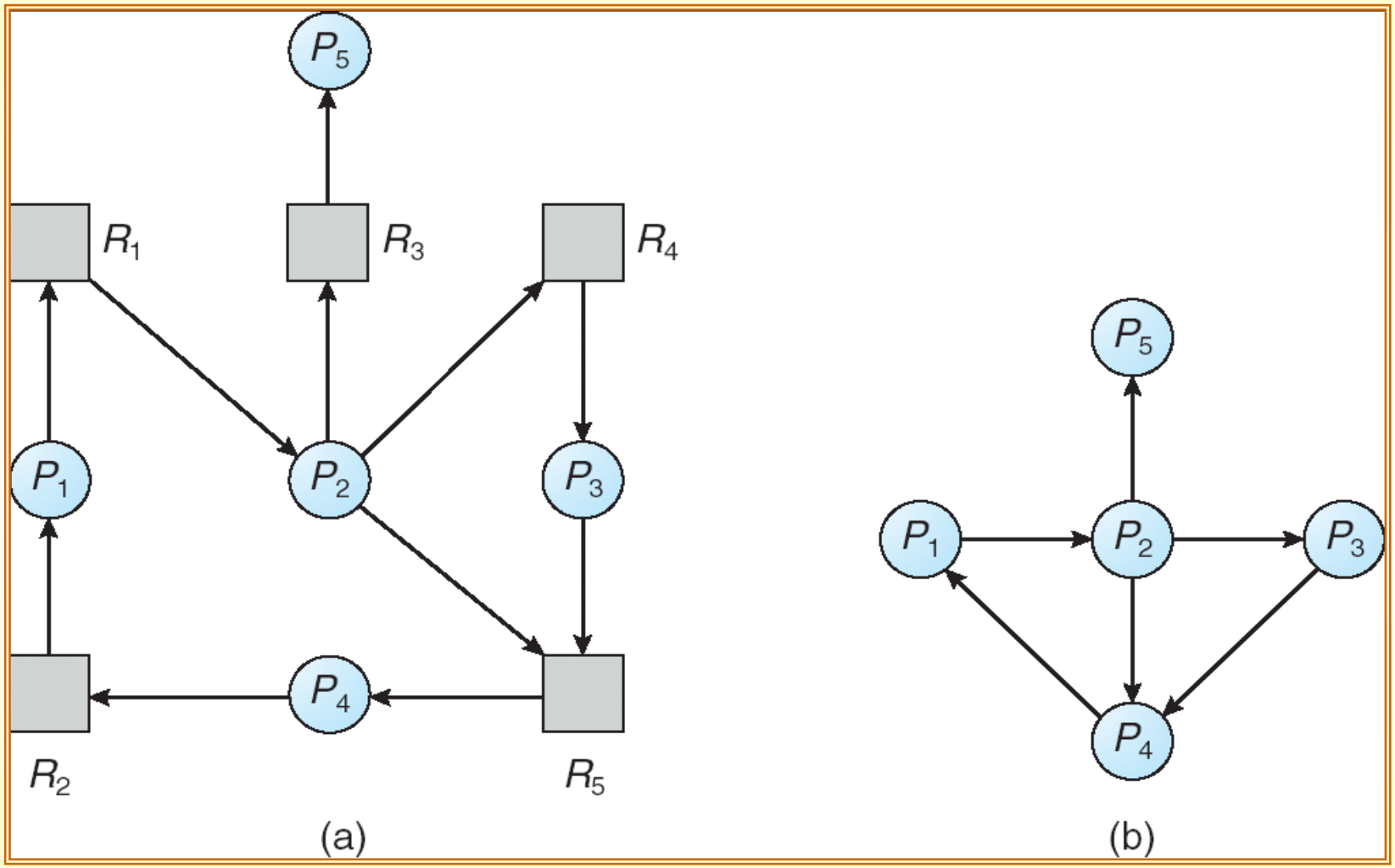
Deadlock Detection

- The system may enter a deadlock state
- The system needs
  - A Detection algorithm that periodically determines whether a deadlock has occurred in the system
  - A procedure to Recover from a deadlock

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 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

Detection Algorithm

This algorithm simply investigates every possible allocation sequence for the processes that remain to be completed

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

Initialize: *Work* = *Available*

For *i* = 0, 1, …, *n* - *1*

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.

Detection Algorithm (Cont.)

3. **Work = Work + Allocation /* Reclaim the resources of process Pi since Request$_i$ < work, so Pi is not in deadlock */**

   Finish*[*i*] = *true

   go to step 2.

4. **If *Finish*[*i*] == false, for some *i*, $0 \le i < n$, then the system is in deadlock state.**

   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.

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	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*. So, System is not in Deadlocked state.

Example (Cont.)

- P_2 requests an additional instance of type *C*.

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
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 the detection algorithm ?

It depends on:

1. How often a deadlock is likely to occur?
2. How many processes will need to be rolled back?
   - one for each disjoint cycle

- If deadlock occur frequently, then the detection algorithm should be invoked frequently
- Invoking deadlock detection algorithm for every resource request incur considerable overhead in computation time
- Invoke the algorithm at less frequent intervals Ex. Once per hour or when the CPU utilization drops below 40%
- 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

- **Two alternatives to do when deadlock detection algorithm determines deadlock**
  1. **To inform the operator, so that he deals manually**
  2. **To let the system recover from deadlock automatically**

- **Two options to break a deadlock:**
  - **Abort one or more processes to break circular wait**
  - **Preempt some resources from one or more of the deadlocked processes**

Recovery from Deadlock : Process termination

- **To eliminate deadlocks by aborting process 2 methods are used:**
  1. **Abort all deadlocked processes**
  2. **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

- Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- Three issues need to be addressed are:
  - Selecting a victim
    - Which resources and which processes are to be preempted?
    - – minimize cost.
  - Rollback
    - If we preempt a resource from a process, what should be done with that process?
    - – return to some safe state, restart process for that state.
  - Starvation
    - How to ensure that starvation will not occur?
    - – same process may always be picked as victim, include number of rollback in cost factor.

End of Chapter 5