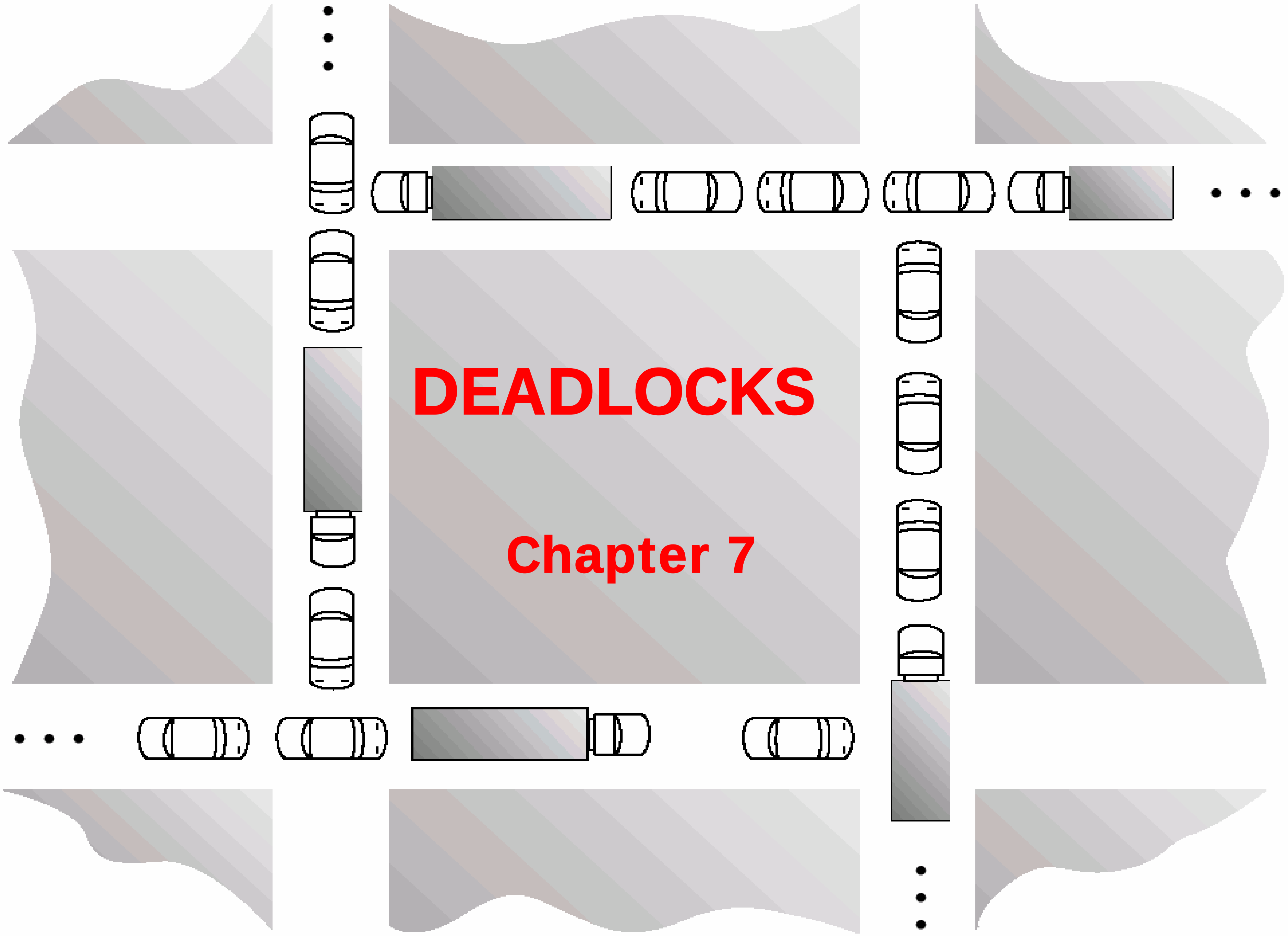


DEADLOCKS

Chapter 7

Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system.

Outline

- The Deadlock Problem +
- System Model +
- Deadlock Characterization +
- Resource Allocation Graph +
- Methods for Handling Deadlocks +
- Deadlock Prevention +
- Deadlock Avoidance +
- Deadlock Detection +
- Recovery from Deadlock +
- Summary +

- The Deadlock Problem ...

- In a multiprogramming environment processes compete for resources.

- A process requests a resource if the resource is not available, the process enters a wait state.

- Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called **deadlock.**

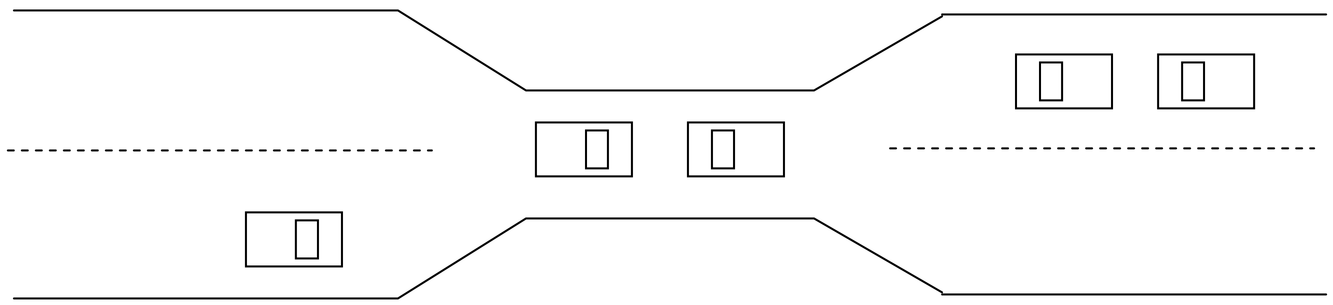
- Example of a deadlock:
    - In the 5 philosophers problem of the previous chapter, if every philosopher holds one chopstick and never puts it back till he gets a second one.

... - 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.
  - P_1 and P_2 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)
```

… - The Deadlock Problem



- 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

- Resources
  - Resources are partitioned into resource types.
  - Each resource type consists of a number of identical instances.
  - Example of resource types are: printers, CPU cycle, file etc.
  - If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request.
  - The number of resources requested may not exceed the total number available in the system.
- Under normal mode of operation, a process may utilize a resource in only the following sequence:
  - **Request**: It may have to wait if requested resource can not be granted immediately.
  - **Use**
  - **Release**

- Deadlock Characterization

- Necessary Conditions for Deadlock +
- Resource Allocation Graph (RAG) +
- Basic Fact +

-- Necessary Conditions for Deadlock

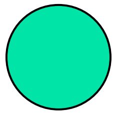
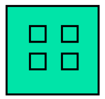
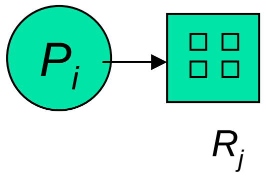
- A deadlock situation can arise if the following ***four conditions hold simultaneously*** in a system:
  - **Mutual exclusion condition**: Each resource is either currently assigned to exactly one process or is available.
  - **Hold and wait condition**: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - **No preemption condition**: Resources can not be preempted; that is, a process can be released only voluntarily by the process holding it, after that process has completed its task.
  - **Circular wait condition**: There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

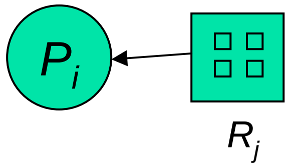
- Resource-Allocation Graph ...

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

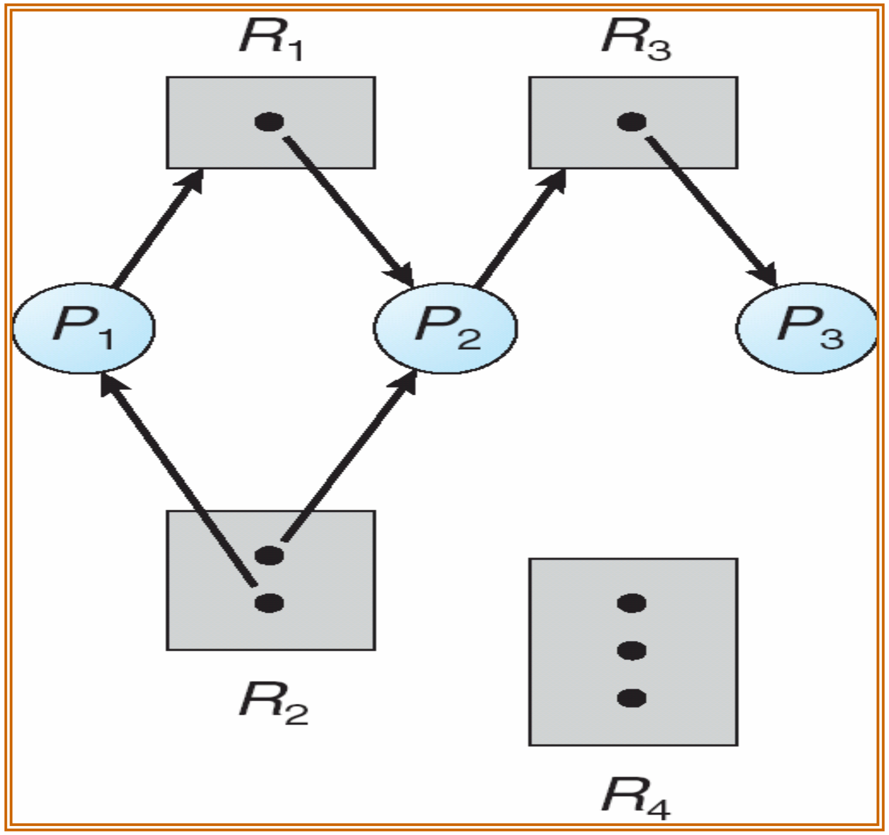
- V is partitioned into two types:
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge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

... - Resource-Allocation Graph (RAG)

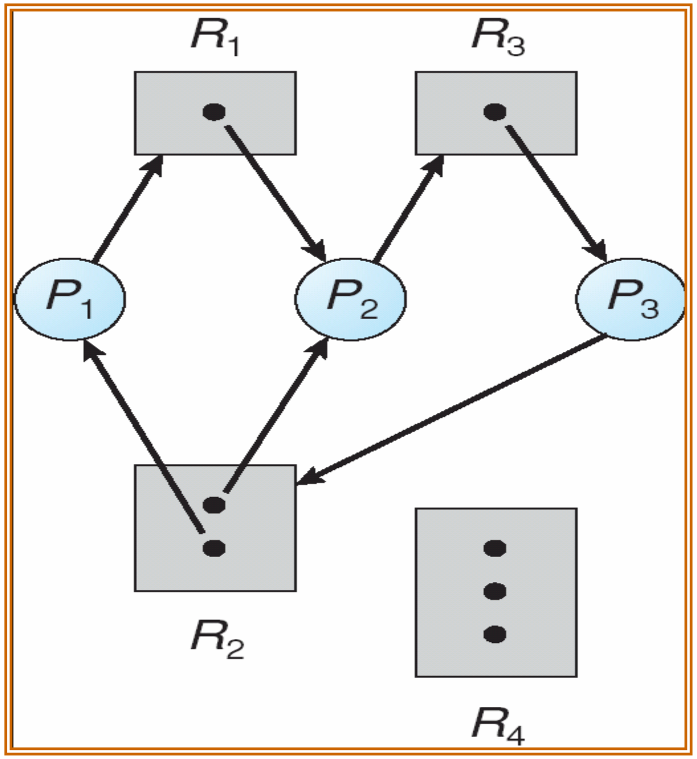
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

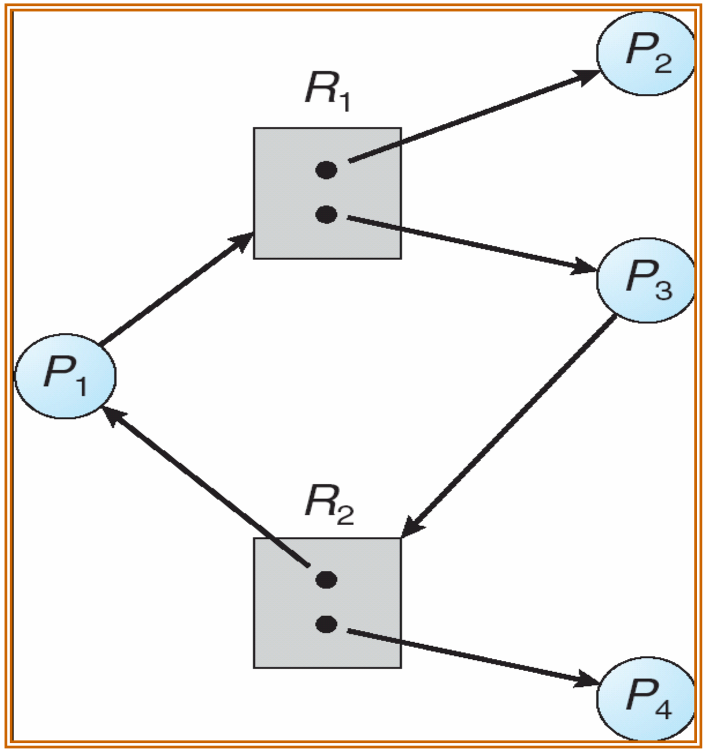
-- Example of a Resource Allocation Graph



-- Example of a RAG



Deadlock



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

- **Prevent** deadlocks, by negating one of the four necessary conditions.
- **Avoid** deadlocks, ensuring that the system will never enter a deadlock state. This requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime.
- **Recover** from deadlocks: 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. (The ostrich algorithm).

- Deadlock Prevention ...

- Is done by restraining the ways request can be made. It ensures that one of the Following four conditions can not occur.
  - **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

- **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.

- **Problems** with prevention is Low device utilization and reduced system throughput.

- Deadlock Avoidance

- Basic Concept +
- Safe State +
- Resource-Allocation Graph Algorithm +
- Bankers Algorithm +

-- Basic Concept

- Deadlock avoidance approach 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 $<P_1, P_2, \ldots, P_n>$ 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.

--- Example: Safe State

- Consider a system with 12 magnetic tape drivers and 3 processes, namely, P0, P1, P2. P0 requires 10 tape drivers, P1 may need 4, and P2 may need up to 9. Suppose at time T0, P0 is holding 5, P1 is holding 2, and P2 is holding 2 tape drives. Thus there are 3 free tape drives.

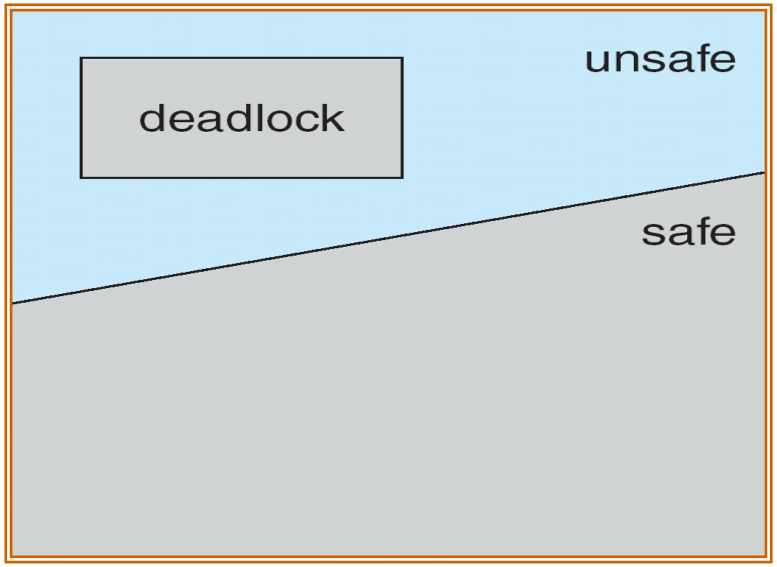
	Maximum Needs	Allocated
P0	10	5
P1	4	2
P2	9	2

- The sequence <P1, P0, P2> satisfies the safety condition.

- If at time T1, P2 requests and is allocated 1 more tape drive the system will be in **unsafe** state.

--- Basic Facts

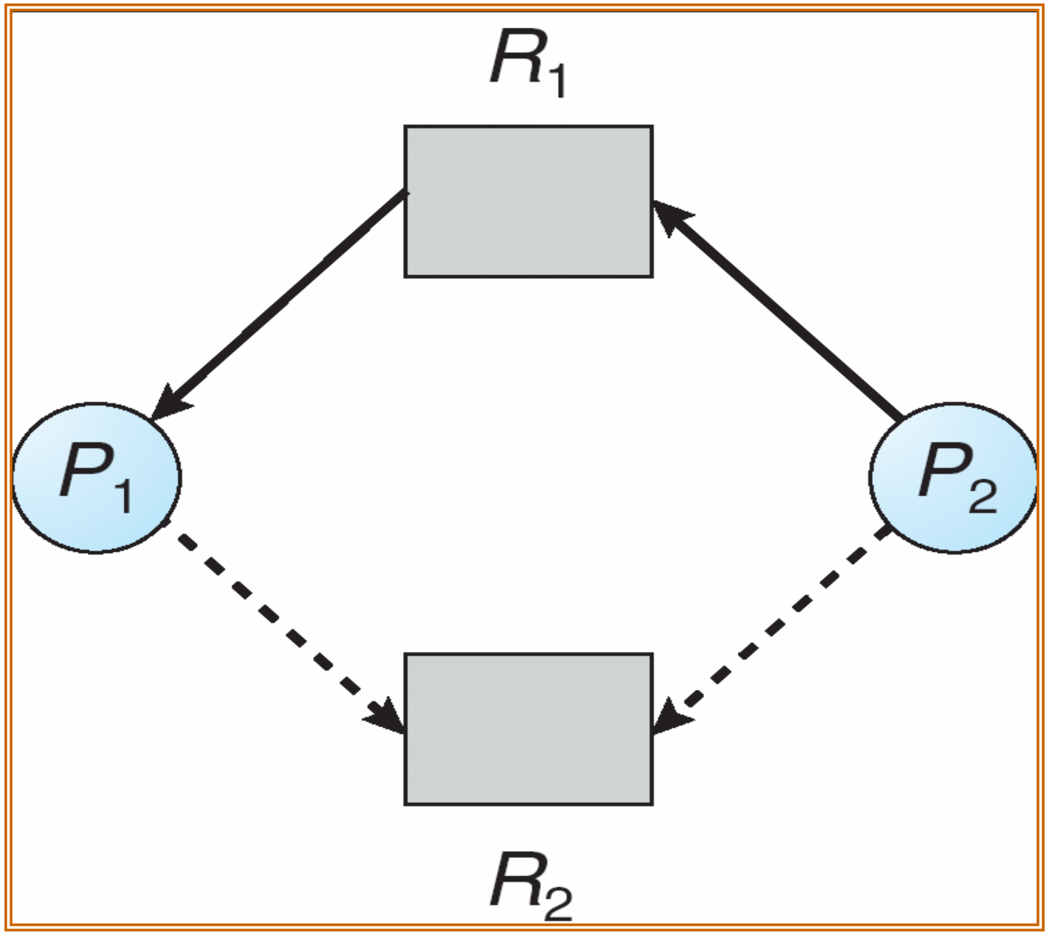
- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.



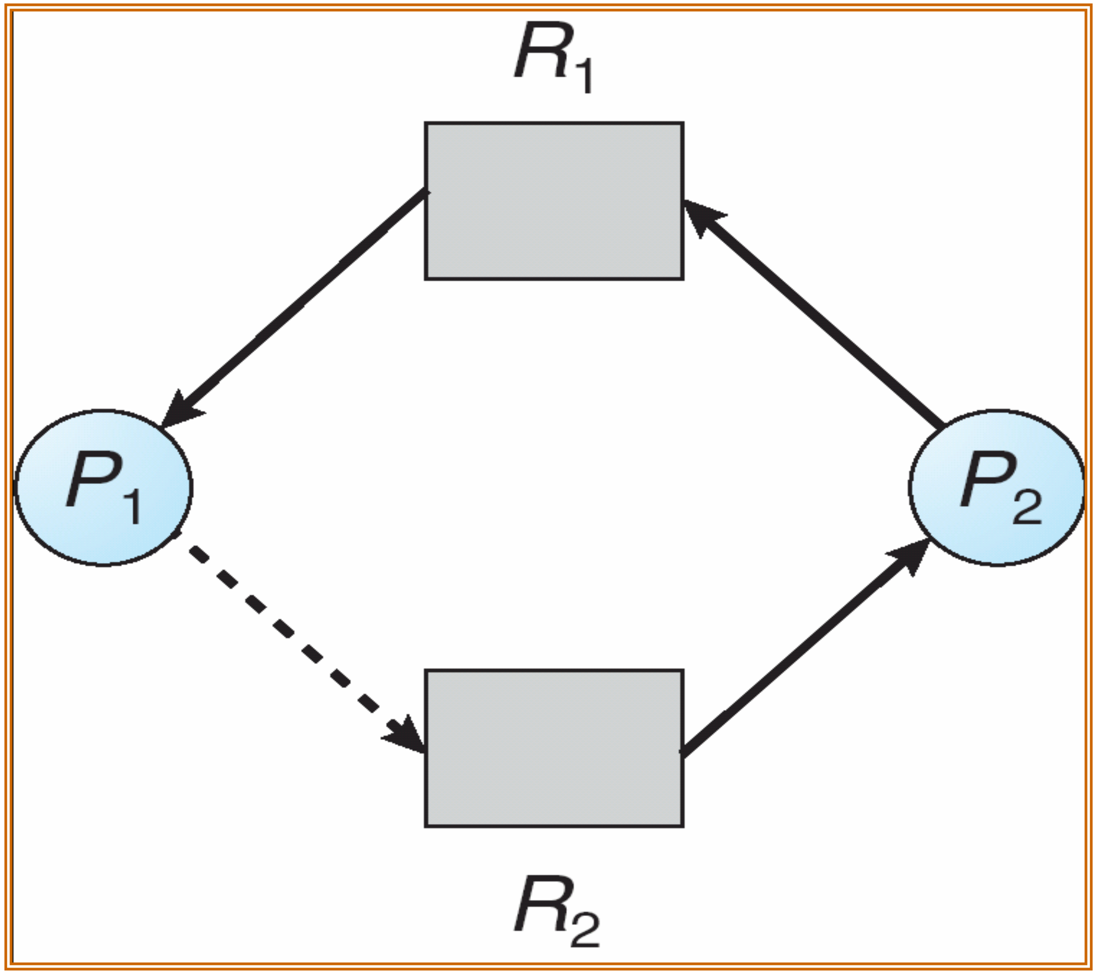
-- Resource-Allocation Graph Algorithm

- Assume one instance per resource type.
- ***Claim edge*** $P_i \dashrightarrow R_j$ 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.

--- RAG For Deadlock Avoidance



--- Unsafe State In RAG



-- 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 $\text{Allocation}[i,j] = k$ 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.

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j].$$

--- Safety Algorithm

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

   Work = *Available*

   Finish [*i*] = *false* for *i* - 1,3, …, *n.*

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 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:

   $Available = Available - Request_i$;

   $Allocation_i = Allocation_i + Request_i$;

   $Need_i = Need_i – Request_i$;;

   - *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 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 of Banker's Algorithm …

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

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

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

… --- Example of Banker's Algorithm

- P_1 Request (1,0,2)
  - Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ *true.*

	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 $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.
  - Can request for (3,3,0) by P_4 be granted?
  - Can request for (0,2,0) by P_0 be granted?

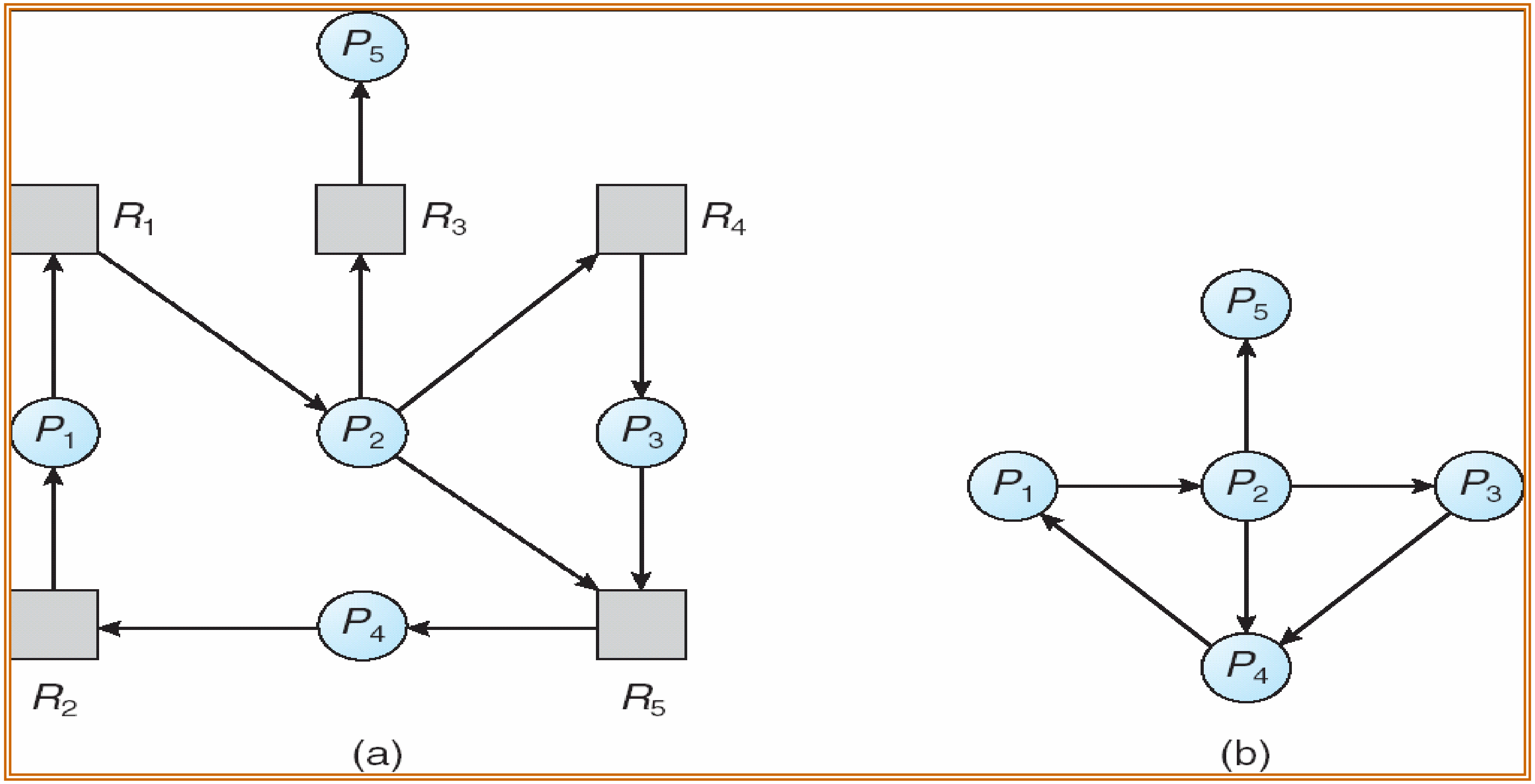
- Deadlock Detection

- Idea
  - Allow system to enter deadlock state
  - Detection algorithm
  - Recovery scheme
- Detection Algorithms
  - Single Instance of Each Resource Type +
  - Several Instances of a Resource Type +

-- 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 ...

1. Let *Work* and *Finish* be vectors of length m and n, respectively Initialize:
   (a) $Work = Available$
   (b) For $i = 1,2, \ldots, n$, 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.

... --- Detection Algorithm

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.

---- 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*.

---- Example of Detection Algorithm

- 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 depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected when it happens
- 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.
  - Starvation – same process may always be picked as victim, include number of rollback in cost factor.
- **Rollback** – return to some safe state, restart process for that state.

- Summary

- The 4 Necessary Conditions for Deadlock
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Cyclic waiting
- Resource Allocation Graph
- Methods for Handling Deadlocks
  - Ostrich algorithm
  - Deadlock Prevention
    - Negate 1 of the 4 necessary conditions for deadlock
  - Deadlock avoidance
    - Safe State
    - Claim edge
    - Banker's algorithm
  - Recovery from deadlock
    - Wait-for-graph
    - Detection algorithm

End of Chapter 7