

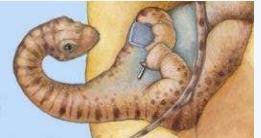
Chapter 7

Deadlocks



Chapter 7: Deadlocks

- 7.1 System Model
- 7.2 Deadlock Characterization
- 7.3 Methods for Handling Deadlocks
- 7.4 Deadlock Prevention
- 7.5 Deadlock Avoidance
- 7.6 Deadlock Detection
- 7.7 Recovery from Deadlock



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, avoiding, or detecting deadlocks in a computer system



7.1 System Model



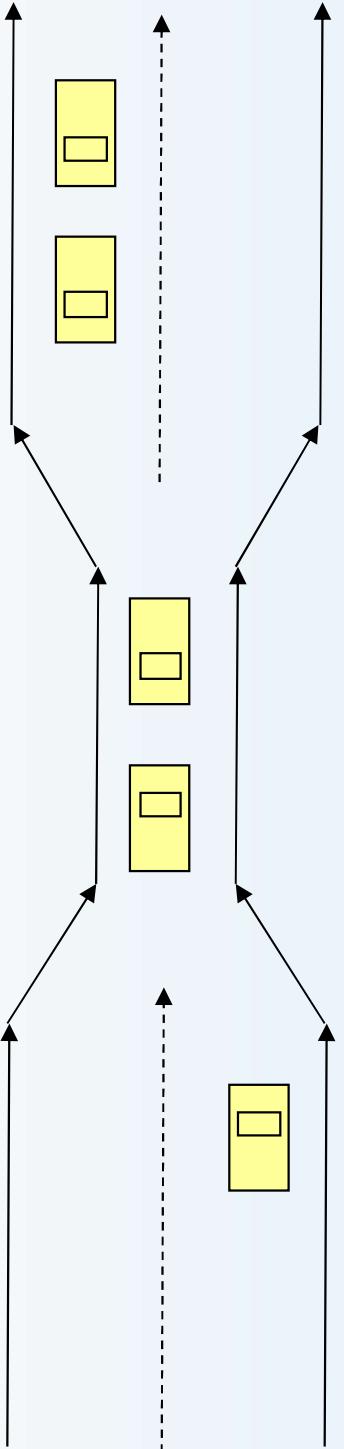
The Deadlock Problem

- A deadlock consists of a set of blocked processes, each holding a resource and waiting to acquire a resource held by another process in the set
- Example #1
 - A system has 2 disk drives
 - P_1 and P_2 each hold one disk drive and each needs the other one
- Example #2
 - Semaphores A and B , initialized to 1

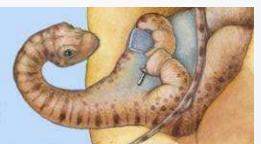
P_0 P_1
`wait (A);` `wait(B)`
`wait (B);` `wait(A)`



Bridge Crossing Example



- Traffic only in one direction
- The resource is a one-lane bridge
- 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

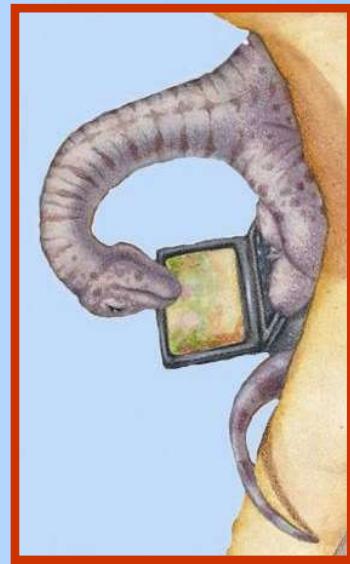


System Model

- Resource types R_1, R_2, \dots, R_m
- CPU cycles, memory space, I/O devices
- Each resource type R_i has 1 or more instances
- Each process utilizes a resource as follows:
 - request
 - use
 - release



7.2 Deadlock Characterization



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 $\{P_0, P_1, \dots, P_n\}$ 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

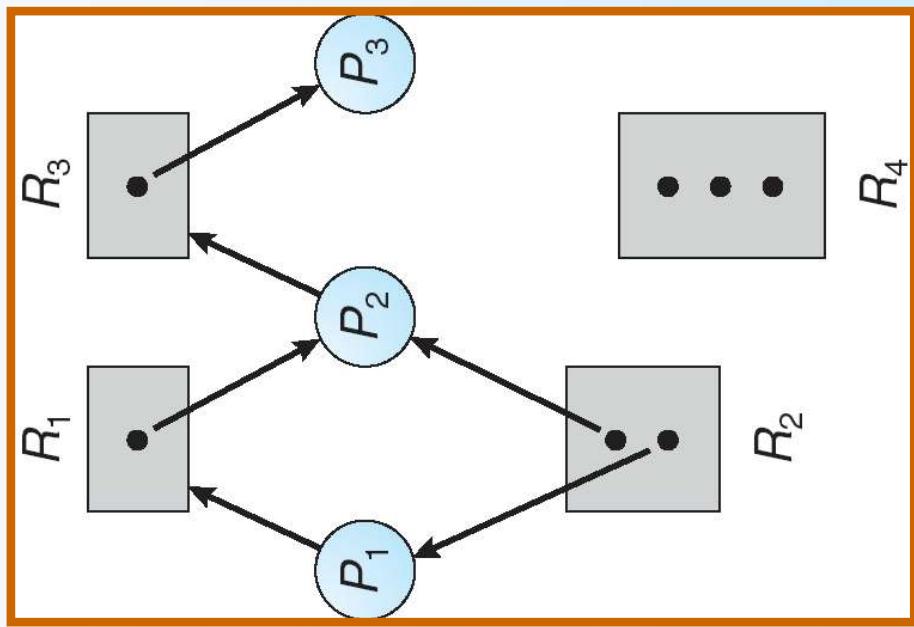


Resource-Allocation Graph

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

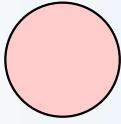
■ V is partitioned into two types:

- $P = \{P_1, P_2, \dots, P_n\}$, 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 $R_j \rightarrow P_i$

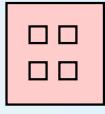


Resource-Allocation Graph (Cont.)

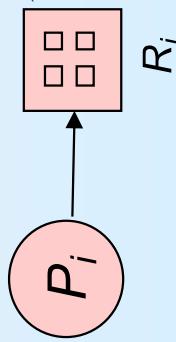
- Process



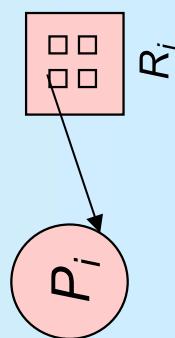
- Resource Type with 4 instances



- P_i requests instance of R_j

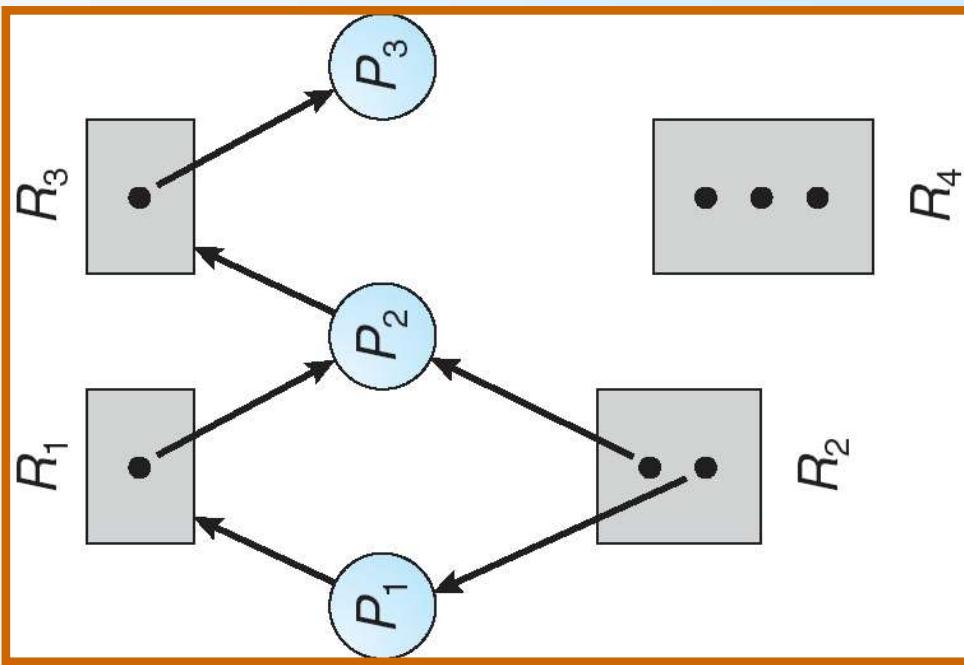


- P_i is holding an instance of R_j

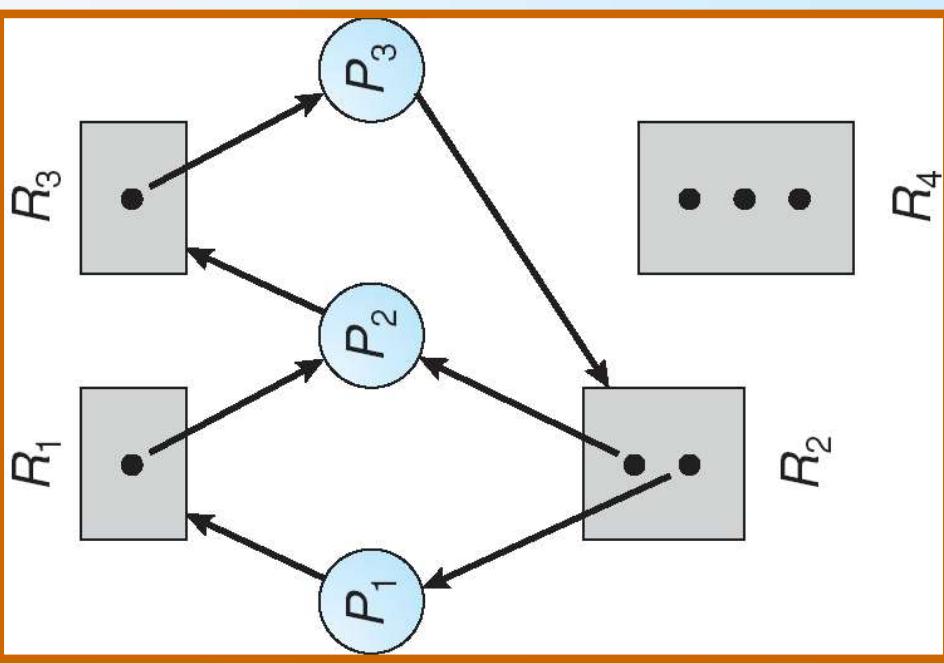


Resource Allocation Graph With A Deadlock

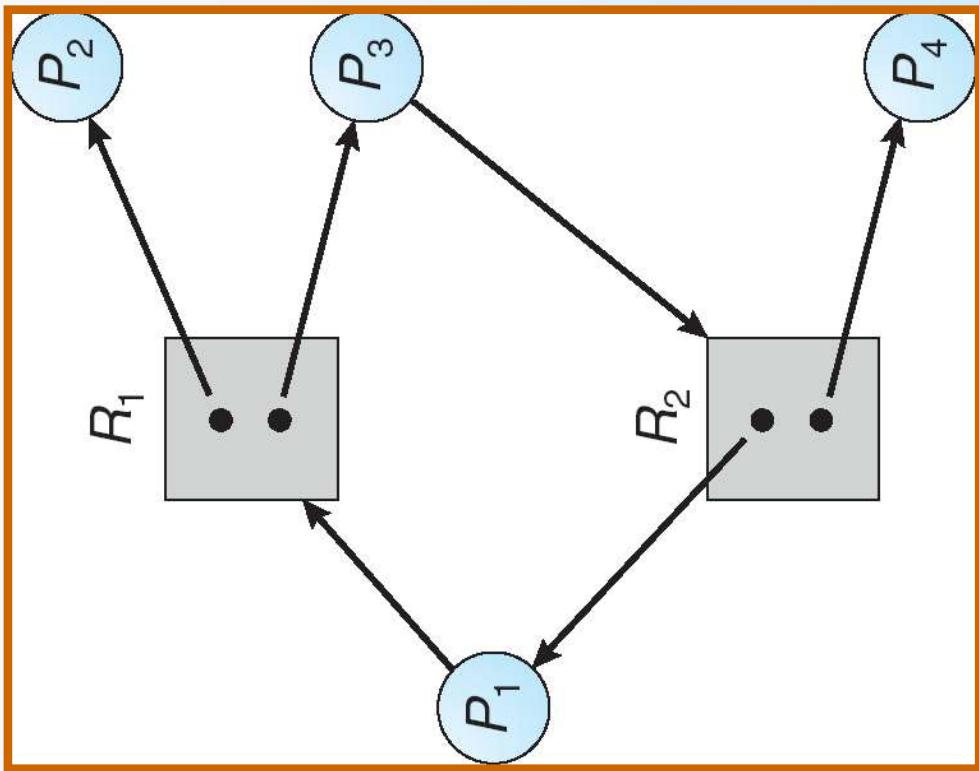
Before P_3 requested an instance of R_2



After P_3 requested an instance of R_2



Graph With A Cycle But No Deadlock

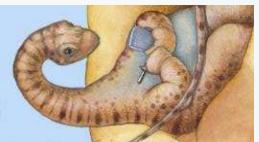


Process P_4 may release its instance of resource type R_2 . That resource can then be allocated to P_3 , thereby breaking the cycle.



Relationship of cycles to deadlocks

- If a resource allocation graph contains no cycles \Rightarrow no deadlock
- If a resource allocation graph contains a cycle and if only one instance exists per resource type \Rightarrow deadlock
- If a resource allocation graph contains a cycle and and if several instances exists per resource type \Rightarrow possibility of deadlock

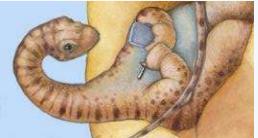


7.3 Methods for Handling Deadlocks



Methods for Handling Deadlocks

- **Prevention**
 - Ensure that the system will *never* enter a deadlock state
- **Avoidance**
 - Ensure that the system will *never* enter an unsafe state
- **Detection**
 - Allow the system to enter a deadlock state and then recover
- **Do Nothing**
 - Ignore the problem and let the user or system administrator respond to the problem; used by most operating systems, including Windows and UNIX



7.4 Deadlock Prevention



Deadlock Prevention

To prevent deadlock, we can restrain the ways that a request can be made

- **Mutual Exclusion** – The mutual-exclusion condition must hold for non-shareable resources
- **Hold and Wait** – we must guarantee that whenever a process requests a resource, it does not hold any other resources
 - Require a process to request and be allocated all its resources before it begins execution, or allow a process to request resources only when the process has none
 - Result: 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
- A 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. For example:

$$\begin{aligned}F(\text{tape drive}) &= 1 \\ F(\text{disk drive}) &= 5 \\ F(\text{printer}) &= 12\end{aligned}$$



7.5 Deadlock Avoidance



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
- A resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

a priori: formed or conceived beforehand



Safe State

- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a safe state
- A system is in a safe state only if there exists a safe sequence
 - A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i , the resource requests that P_i can still make, can be satisfied by currently available resources plus resources held by all P_j , with $j < i$.
- That is:
 - If the 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_j 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

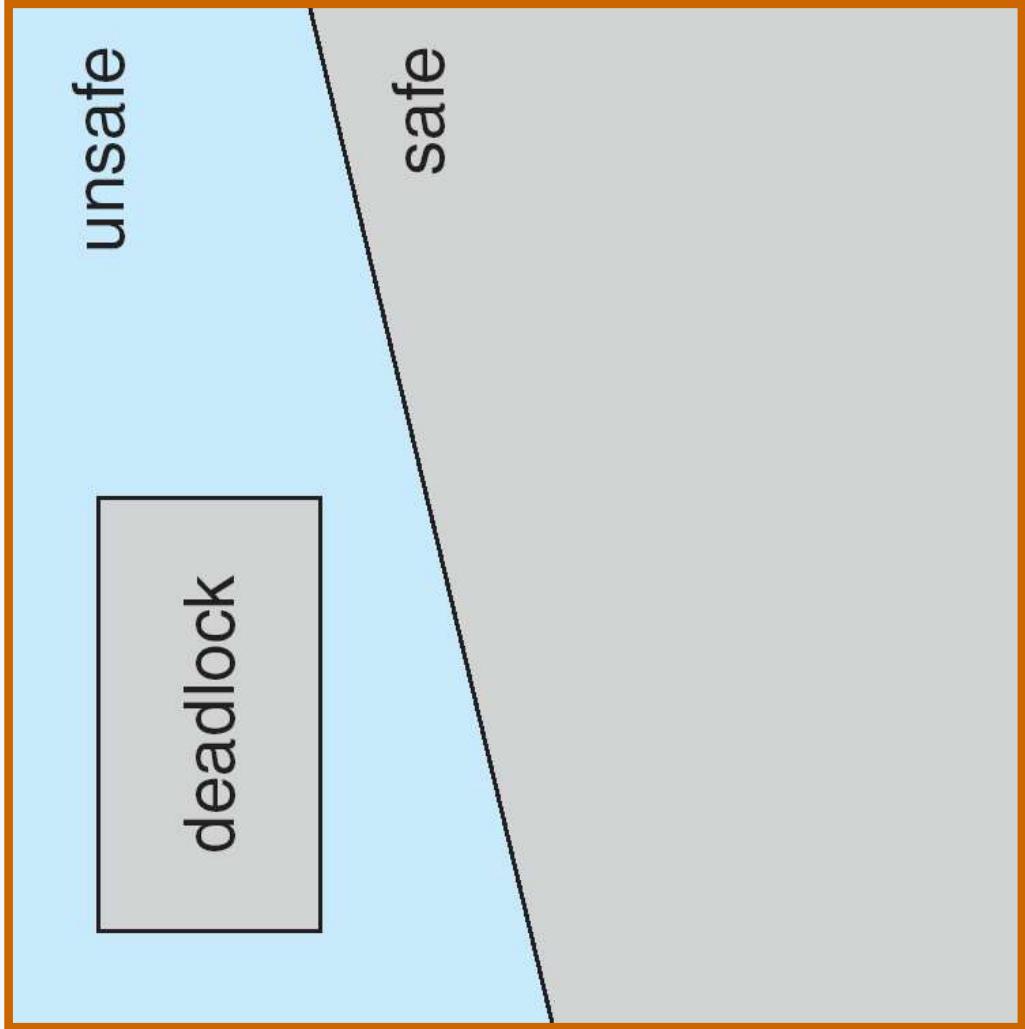


Safe State (continued)

- 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



Avoidance algorithms

- For a single instance of a resource type, use a resource-allocation graph
- For multiple instances of a resource type, use the banker's algorithm

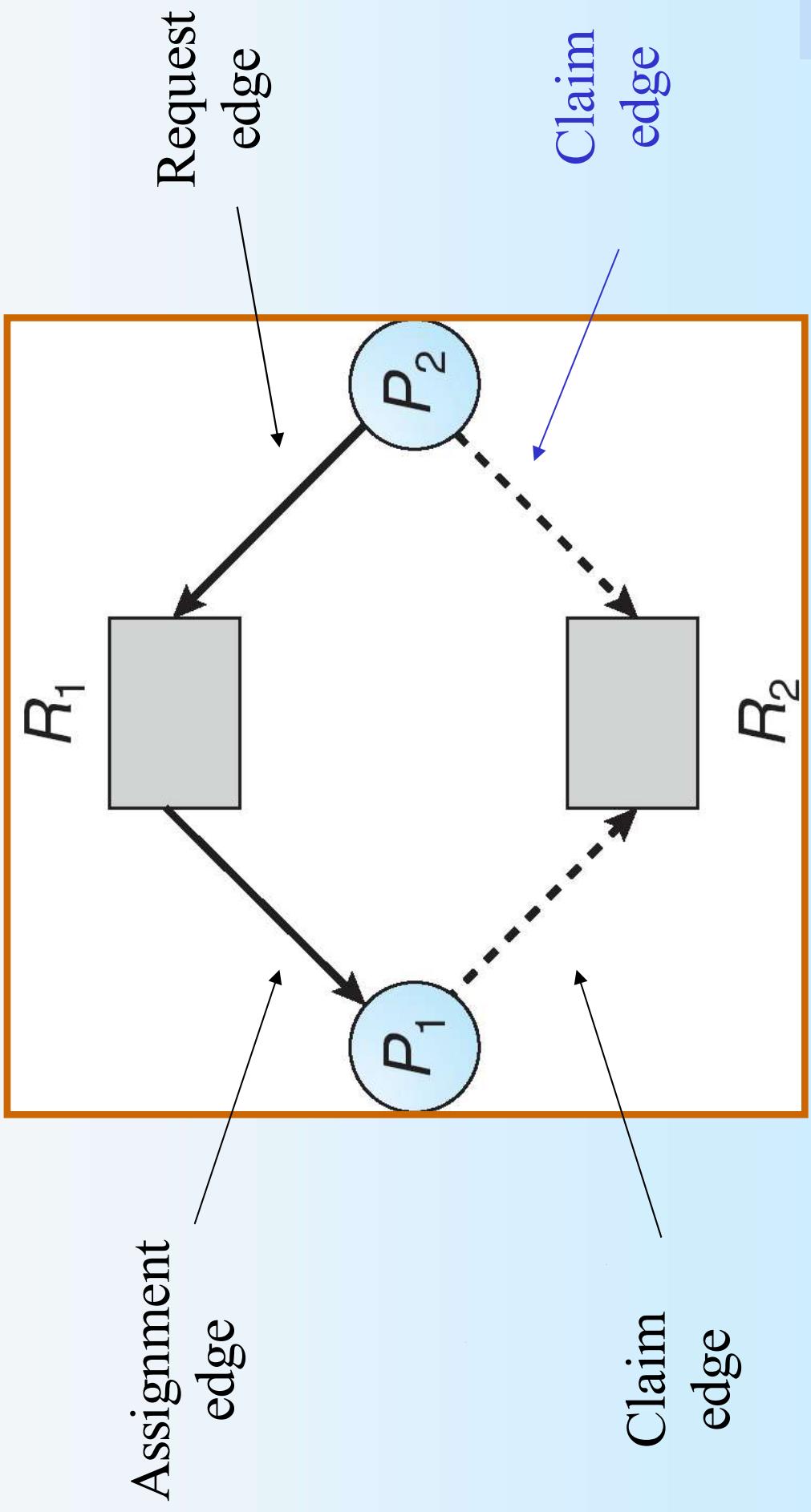


Resource-Allocation Graph Scheme

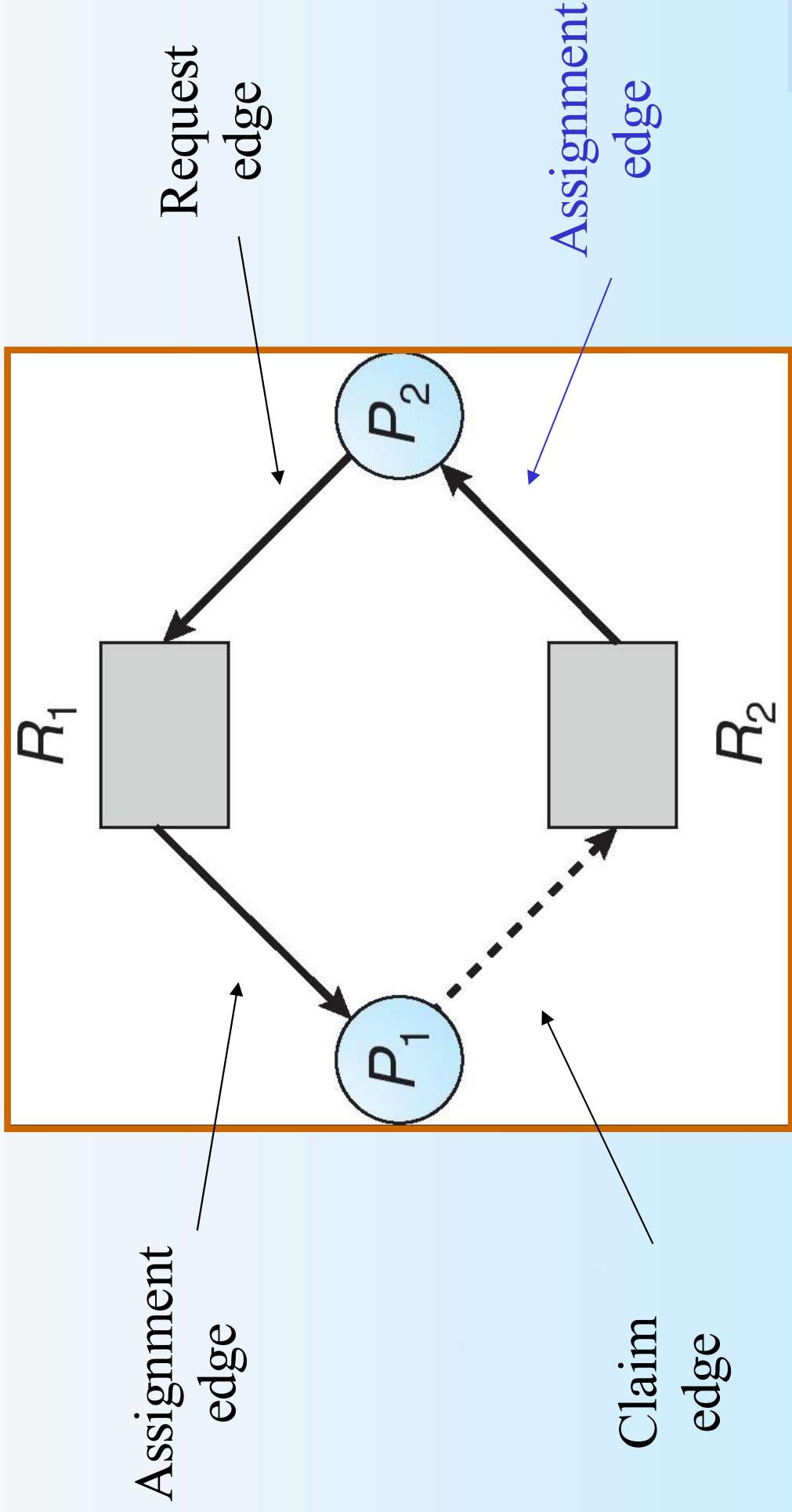
- Introduce a new kind of edge called a claim edge
- *Claim edge* $P_i \dashrightarrow R_j$ indicates that process P_i may request resource R_j , which is represented by a dashed line
- A claim edge converts to a request edge when a process **requests** a resource
- A request edge converts to an assignment edge when the resource is **allocated** to the process
- When a resource is **released** by a process, an assignment edge reconverts to a claim edge
- Resources must be **claimed *a priori*** in the system



Resource-Allocation Graph with Claim Edges



Unsafe State In Resource-Allocation Graph



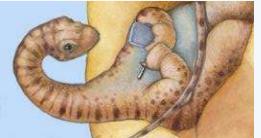
Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource 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

- Used when there exists **multiple** instances of a resource type
- 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

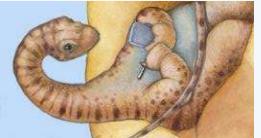


Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available:** Vector of length m . If $\text{available}[j] = k$, there are k instances of resource type R_j available.
- **Max:** $n \times m$ matrix. If $\text{Max}[i,j] = k$, 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 $\text{Need}[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

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



7.6 Deadlock Detection



Deadlock Detection

- For deadlock detection, the system must provide
 - An algorithm that examines the state of the system to detect whether a deadlock has occurred
 - And an algorithm to recover from the deadlock
- A detection-and-recovery scheme requires various kinds of overhead
 - Run-time costs of maintaining necessary information and executing the detection algorithm
 - Potential losses inherent in recovering from a deadlock

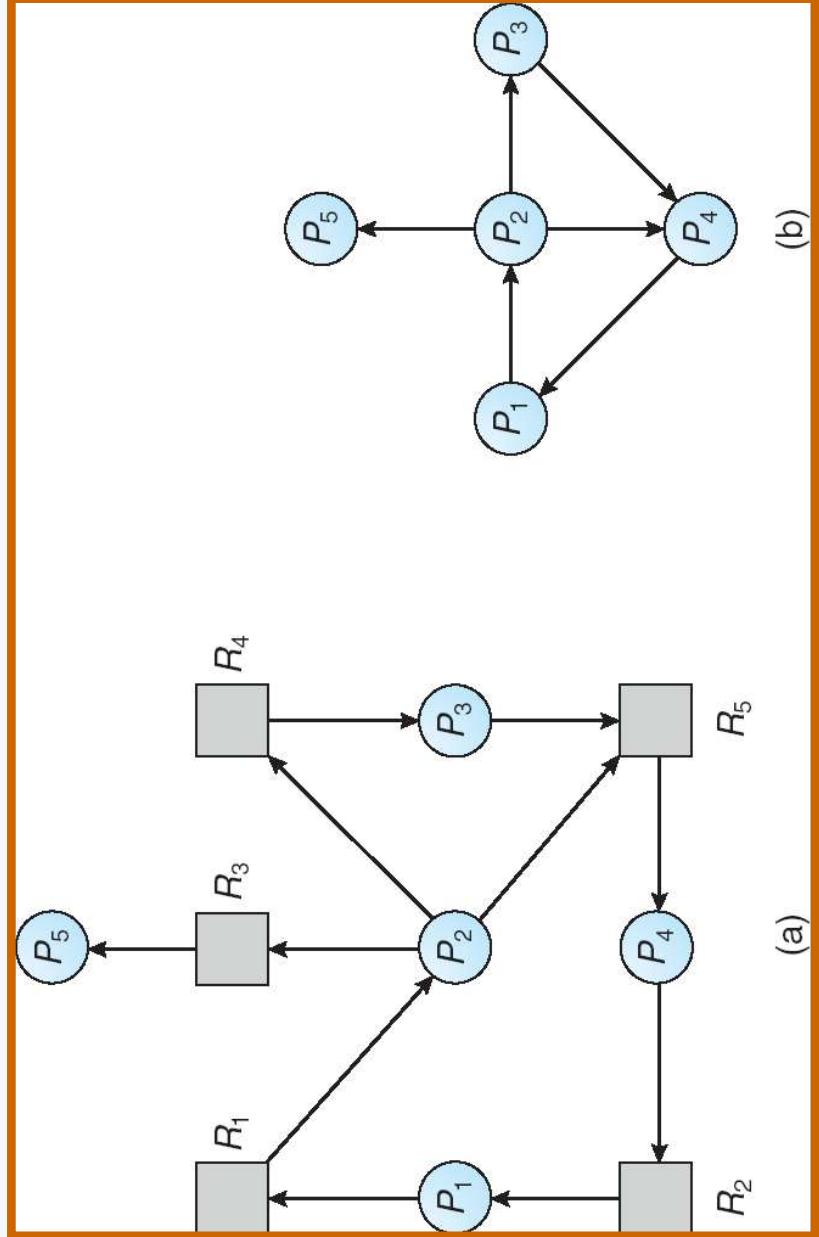


Single Instance of Each Resource Type

- Requires the creation and maintenance of a wait-for graph
 - Consists of a variant of the resource-allocation graph
 - The graph is obtained by **removing** the resource nodes from a resource-allocation graph and **collapsing** the appropriate edges
 - Consequently; all 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



Multiple Instances of a Resource Type

Required data structures:

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



Detection-Algorithm Usage

- When, and how often, to invoke the detection algorithm depends on:
 - How often is a deadlock likely to occur?
 - How many processes will be affected by deadlock when it happens?
- If the detection algorithm is invoked arbitrarily, there may be **many cycles** in the resource graph and so we would not be able to tell **which one** of the many deadlocked processes “caused” the deadlock
- If the detection algorithm is invoked for every resource request, such an action will incur a considerable **overhead** in computation time
- A less expensive alternative is to invoke the algorithm when CPU utilization drops **below 40%**, for example
 - This is based on the observation that a deadlock eventually cripples system throughput and causes CPU utilization to drop



7.7 Recovery From Deadlock



Recovery from Deadlock

- Two Approaches
 - Process termination
 - Resource preemption



Recovery from Deadlock: Process Termination

- **Abort all deadlocked processes**

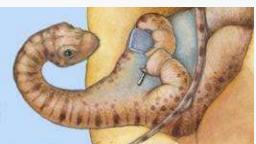
- This approach will break the deadlock, but at great expense

- **Abort one process at a time until the deadlock cycle is eliminated**

- This approach incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be re-invoked to determine whether any processes are still deadlocked

- **Many factors may affect which process is chosen for termination**

- What is the priority of the process?
- How long has the process run so far and how much longer will the process need to run before completing its task?
- How many and what type of resources has the process used?
- How many more resources does the process need in order to finish its task?
- How many processes will need to be terminated?
- Is the process interactive or batch?



Recovery from Deadlock: Resource Preemption

- With this approach, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- When preemption is required to deal with deadlocks, then three issues need to be addressed:
 - **Selecting a victim** – Which resources and which processes are to be preempted?
 - **Rollback** – If we preempt a resource from a process, what should be done with that process?
 - **Starvation** – How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?



Summary

- Four necessary conditions must hold in the system for a deadlock to occur
 - Mutual exclusion
 - Hold and wait
 - No preemption
 - Circular wait
- Four principal methods for dealing with deadlocks
 - Use some protocol to (1) **prevent** or (2) **avoid** deadlocks, ensuring that the system will never enter a deadlock state
 - Allow the system to enter a deadlock state, (3) **detect** it, and then **recover**
 - ▶ Recover by **process termination** or **resource preemption**
 - (4) **Do nothing**; ignore the problem altogether and pretend that deadlocks never occur in the system (used by Windows and Unix)
- To prevent deadlocks, we can ensure that **at least one** of the four necessary conditions **never holds**



End of Chapter 7

