CS420: Operating Systems
Deadlocks & Deadlock Prevention

James Moscola
Department of Engineering & Computer Science
York College of Pennsylvania

The Deadlock Problem

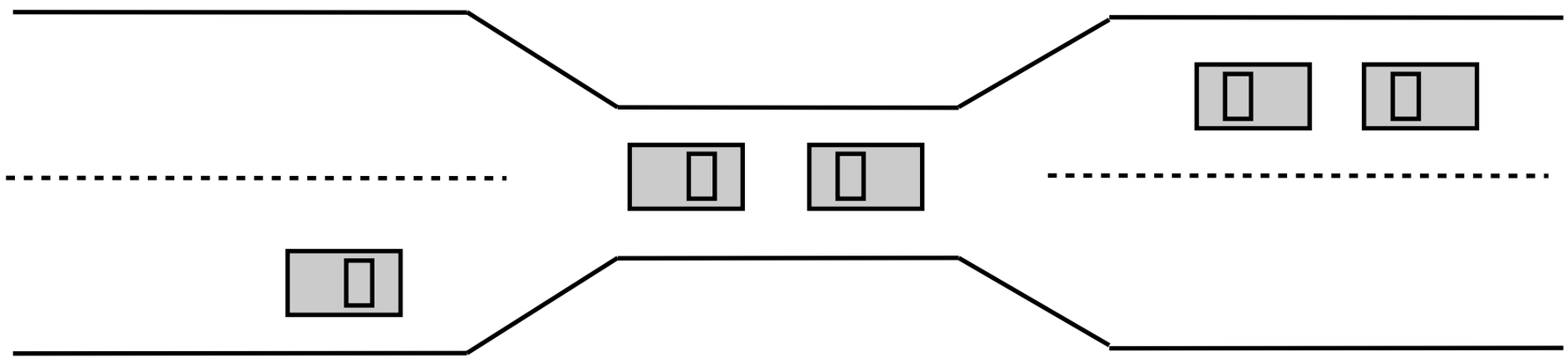
- **Deadlock - A condition that arises when two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes**
  - The events that the processes are waiting for will never happen :-(
  - Other processes in the system that need the resources that are currently locked will never get access to them

- **Operating systems do not typically provide deadlock-prevention facilities**
  - It is up to the programmer to design deadlock-free programs

- **Deadlock will become an even greater problem with the trend toward more processor cores and more threads**

Bridge Crossing Example

- **Traffic can only pass bridge 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**



Deadlock Characterization

- **Deadlock can arise if the 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 of resources - 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 $\{P_0, P_1, \ldots, P_n\}$ 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

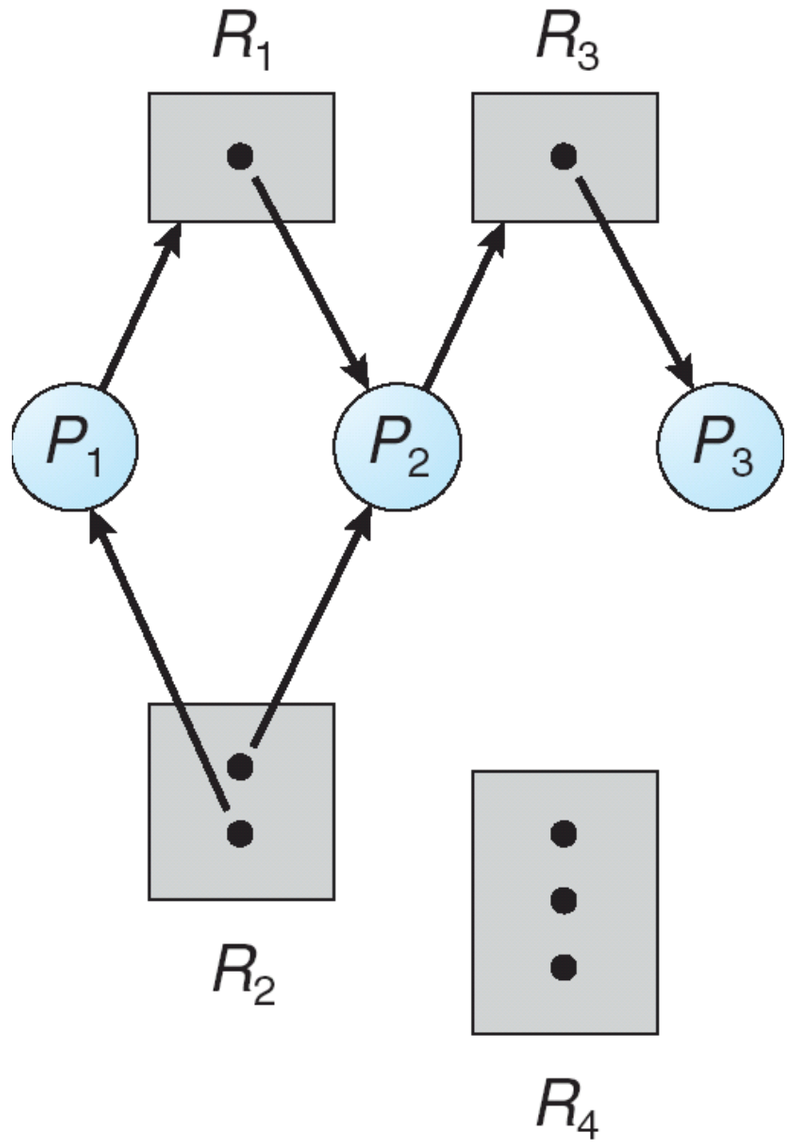
Resource-Allocation Graph

- **Deadlocks can be described precisely using system resource-allocation graphs**

- **A resource-allocation graph consists of a set of vertices V and a set of edges E**
  - V is partitioned into two sets:
    - $P = \{P_1, P_2, \ldots, P_n\}$, 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
  - There are also two different types of edges:
    - Request edge – directed edge $P_i \rightarrow R_j$, a process P_i is requesting a resource of type R_j
    - Assignment edge – directed edge $R_j \rightarrow P_i$, a resource of type R_j has been assigned to a process P_i

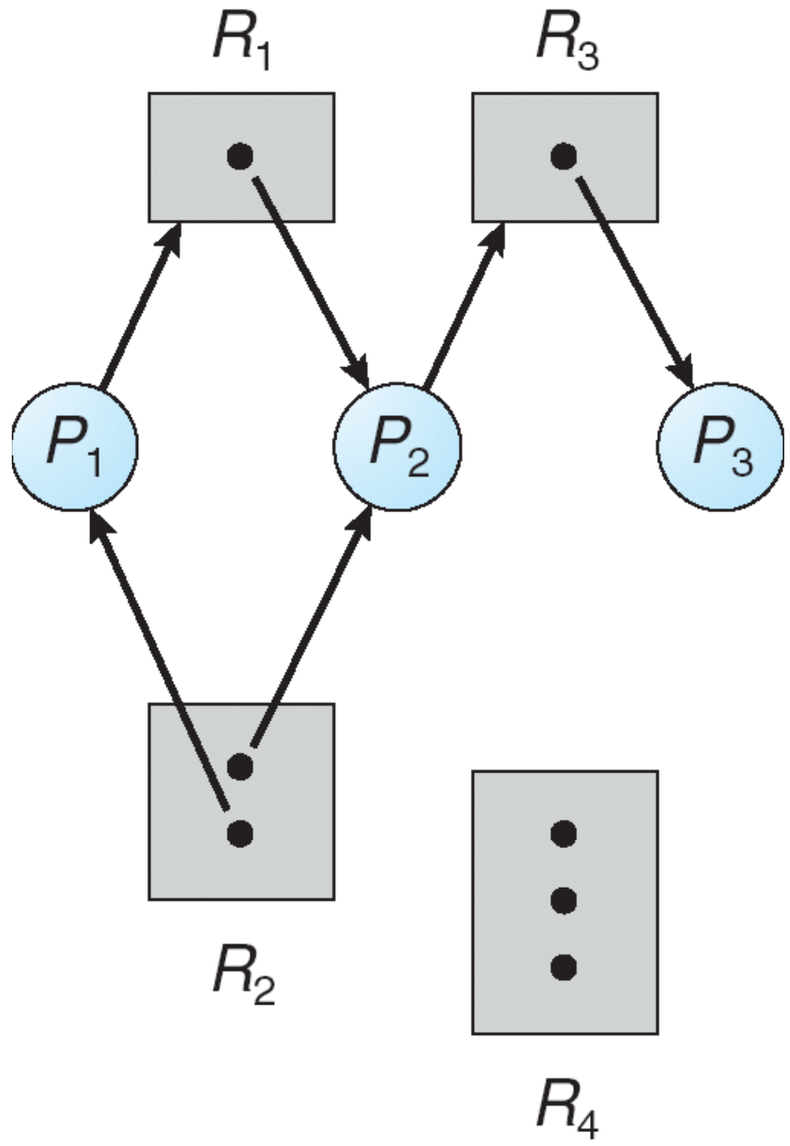
Resource-Allocation Graph (Cont.)

- **A process is represented as a circular vertex in the graph**
- **A resource type is represented as a square vertex in the graph**
  - Because there may be multiple instances of some resources (i.e. two disks), each instance of a resource is represented using a dot
- **A directed edge from a process P_i to a resource R_j is a request from P_i for a resource of type R_j**
- **A directed edge from a resource instance to a process P_i indicates that the resource has been allocated to P_i**



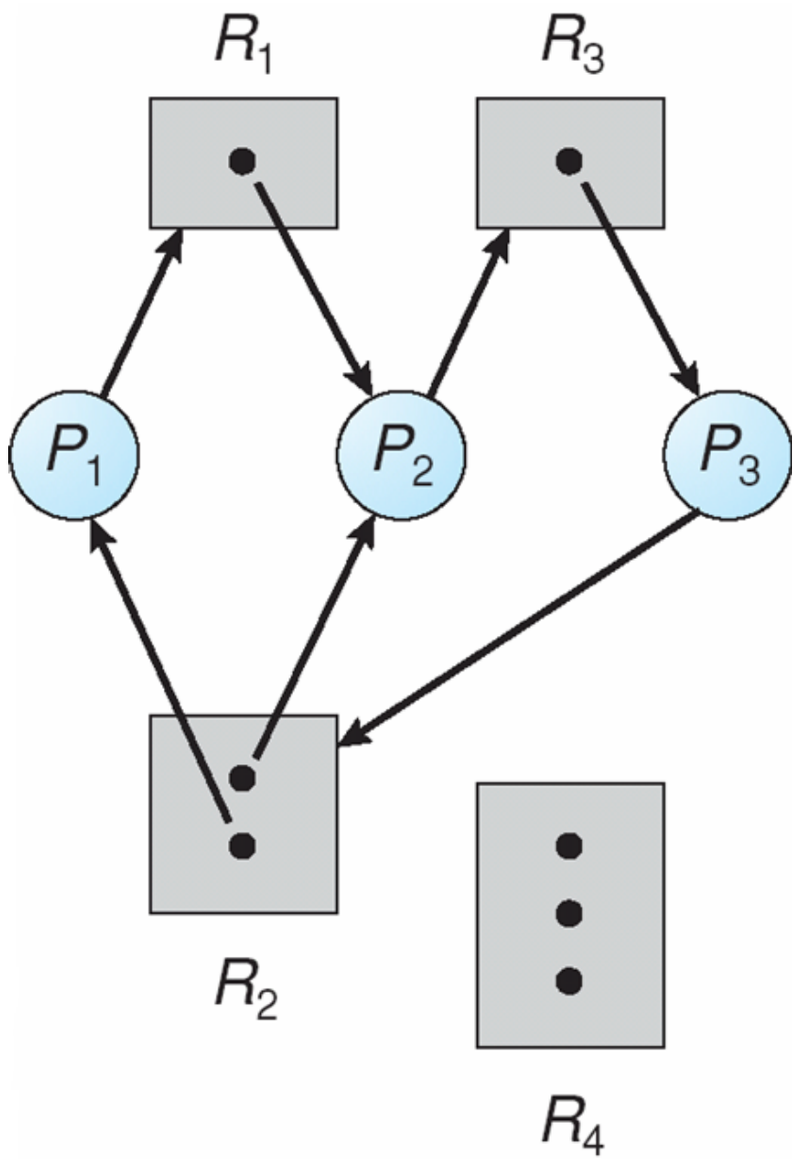
Resource-Allocation Graph (Cont.)

- **If the graph contains no cycles, then no process in the system is deadlocked**
- **If the graph does contain cycles, then deadlock may exist, but is not guaranteed to exist**
  - If a resource type has exactly one instance then a cycle implies that a deadlock has occurred
  - If a resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred



Resource Allocation Graph With A Deadlock

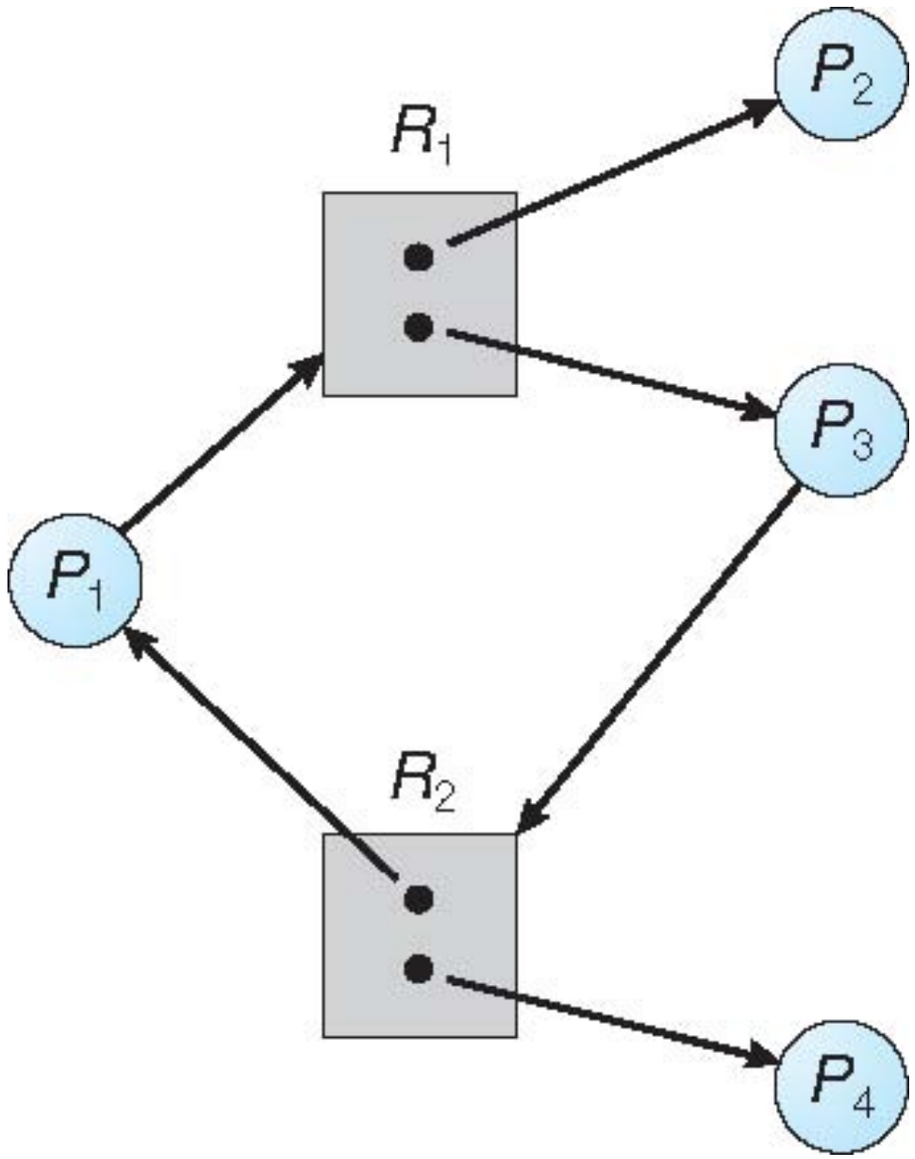
- **Two cycles exist in the graph to the right**
  - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2$
  - $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$



- **Processes P_1, P_2 and P_3 are deadlocked**
  - P_2 is waiting for R_3 (held by P_3)
  - P_3 is waiting for R_2 (held by P_1 and P_2)
  - P_1 is waiting for R_1 (held by P_2)

Graph With a Cycle But No Deadlock

- **A cycle exists in the graph to the right**
  - $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- **Processes P_3 is waiting for resource R_2**
  - P_4 should eventually release R_2
- **Process P_1 is waiting for resource R_1**
  - P_2 should eventually release R_1



Resource Allocation Graph Recap

- **If graph contains no cycles then no deadlock possible**

- **If graph contains a cycle then,**
  - If only one instance per resource type, then deadlock exists
  - If several instances per resource type, then deadlock is possible

Methods for Handling Deadlocks

- **Possible for OS to deal with deadlock in one of three ways:**
  - 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)
    - Requires that application developers write programs that avoid deadlock

Methods for Handling Deadlocks

- **To ensure that deadlocks never occur, a system can use either deadlock-prevention or deadlock-avoidance**

- **Deadlock Prevention - ensure that at least one of the four necessary conditions for deadlock cannot hold**

- **Deadlock Avoidance - requires that the operating system be given comprehensive information about which resources a process will request during its lifetime**
  - Operating system can then make intelligent decisions about when a process should be allocated a resource

Methods for Handling Deadlocks

- **If a system does not use either deadlock-prevention or deadlock-avoidance, a deadlock situation may eventually arise**
  - Need some way to detect these situations -- deadlock detection
  - Need some way to recover from these situations -- deadlock recovery

- **If a system does not use deadlock-detection/deadlock-avoidance, and does not use deadlock-detection/deadlock-recovery, then the system performance may eventually deteriorate**
  - Reboot

Deadlock Prevention

- **For deadlock to occur, each of the four necessary conditions must hold true**
- **To prevent deadlock, ensure that at least one of these four conditions does not hold true**

(1) Mutual Exclusion

- Not required for sharable resources (i.e. read-only files)
- Must hold true for non-sharable resources :-(
- In general, cannot prevent deadlock by denying the mutual-exclusion condition because some resources are non-sharable

Deadlock Prevention (Cont.)

- **For deadlock to occur, each of the four necessary conditions must hold true**
- **To prevent deadlock, ensure that at least one of these four conditions does not hold true**

(2) Hold-and-wait

- To ensure the hold-and-wait condition never occurs, must guarantee that whenever a process requests a resource, it does not hold any other resources

- One approach requires a process to request and be allocated all its resources before it begins execution
  - Low resource utilization because resources may allocated but unused for a long time
  - Starvation is possible if at least one of the resources that a process needs is always allocated to some other process
- Another approach allows a process to request resources only when it has none

Deadlock Prevention (Cont.)

- **For deadlock to occur, each of the four necessary conditions must hold true**
- **To prevent deadlock, ensure that at least one of these four conditions does not hold true**

(3) No Preemption of Resources

- To ensure that the no preemption condition never occurs:
  - 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
  - All 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

Deadlock Prevention (Cont.)

- **For deadlock to occur, each of the four necessary conditions must hold true**
- **To prevent deadlock, ensure that at least one of these four conditions does not hold true**

(4) Circular Wait

- To ensure that a circular waiting condition never occurs, impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

- Each resource type is given an integer value, must request desired resource in increasing order

LAN Port = 1
Disk Drive = 2
Printer = 3

If a process needs both the LAN and the printer, it must request the LAN first, and then request the printer

Deadlock Avoidance

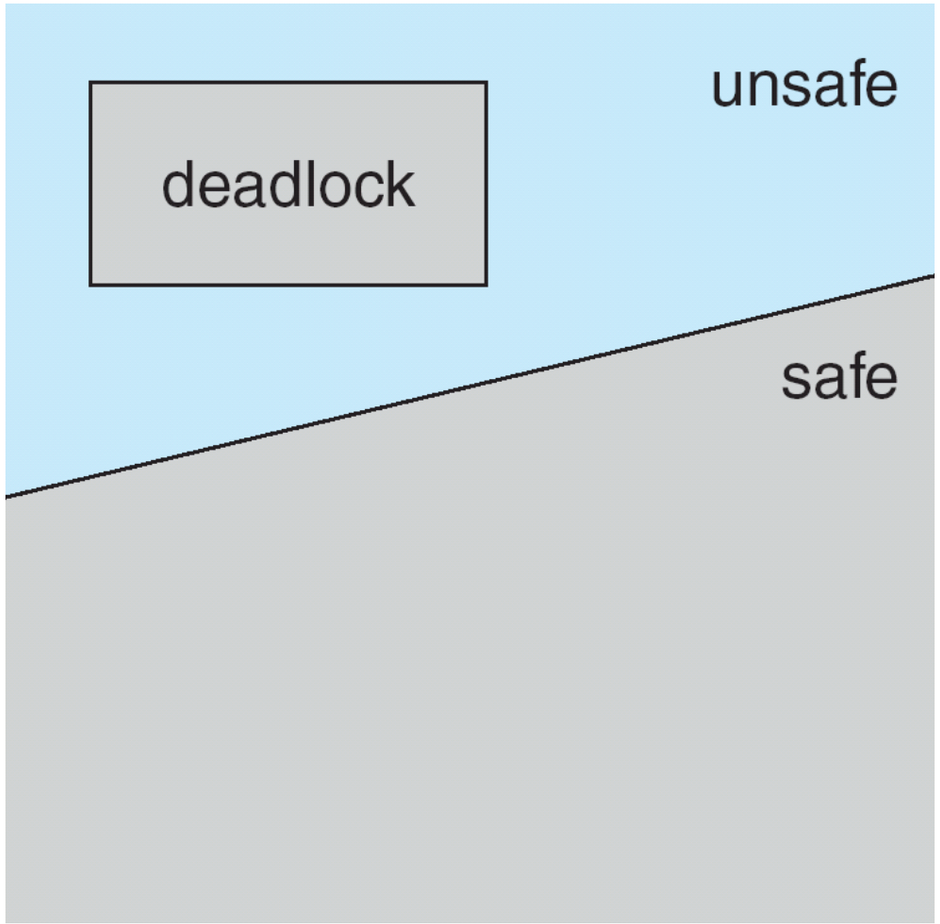
- **Requires that the system has some additional *a priori* information available**
  - Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
  - A deadlock-avoidance algorithm dynamically examines the resource-allocation *state* 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, the system must decide if immediate allocation leaves the system in a safe state**
- **System is in a safe state if it can allocate resources to each process (up to the processes maximum) in some order and still avoid deadlock**
- **System is in a safe state if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources plus the resources held by all the P_j, with j < i**
- **That is:**
  - If the resources needed by P_i are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain the resources it needs, execute, return allocated resources, and terminate
  - When P_i terminates, P_i +1 can obtain the resources it needs, and so on

Safe, Unsafe, Deadlock State

- **If a system is in safe state ⇒ no deadlocks**
- **If a system is in unsafe state ⇒ possibility of deadlock**
- **To avoid deadlock, ensure that a system never enters an unsafe state**
  - Cannot transition from safe state directly to deadlocked stack
  - Can only transition to deadlocked state from unsafe state
  - By avoiding unsafe state, system can avoid deadlock



Avoidance Algorithms

- **Two main deadlock avoidance algorithms ... choice depends on how many instances of available resource types exist**
  - Single instance of a resource type
    - Use a resource-allocation graph
  - Multiple instances of a resource type
    - Use the Banker's Algorithm

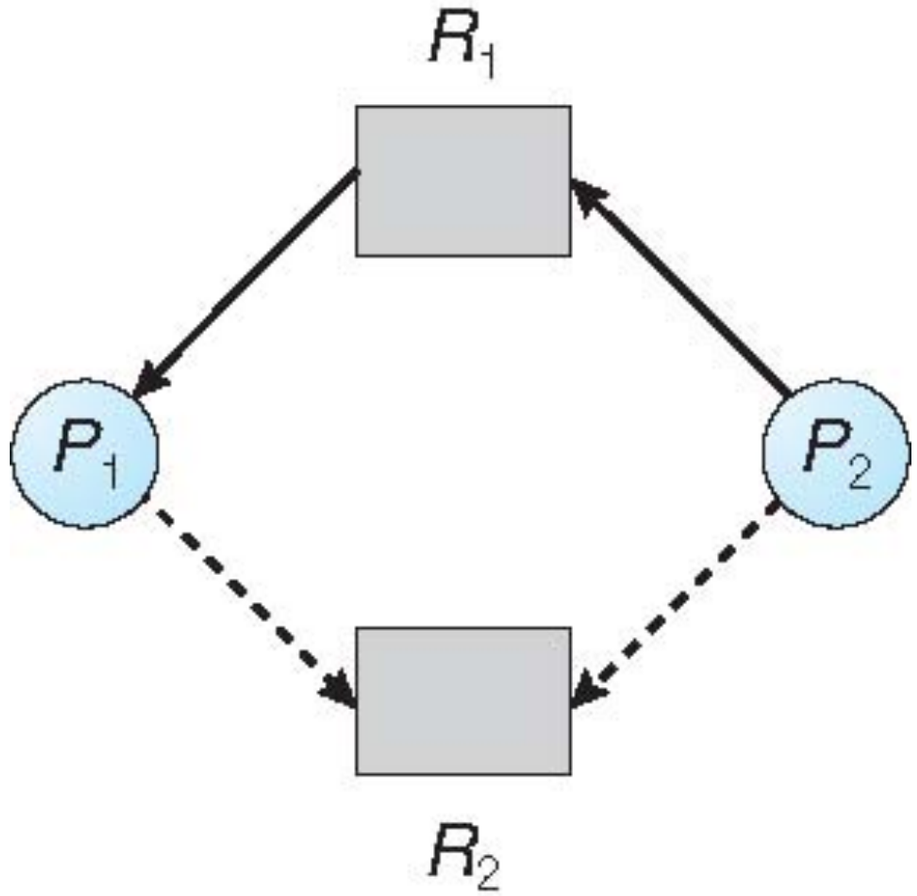
Resource-Allocation Graphs for Deadlock Avoidance

- **Introduce a new type of edge to the resource-allocation graph, the claim edge**
  - A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request a resource R_j
  - A claim edge is represented as dashed lines in resource-allocation graph
  - Resources must be claimed a priori in the system, so all claim edges are known

- **A claim edge converts to request edge when a process requests a resource**
- **A request edge is converted to an assignment edge when the resource is allocated to the process**
- **When a resource is released by a process, an assignment edge converts back to a claim edge**

Resource-Allocation Graph

- **In this graph, both processes P_1 and P_2 may eventually want to request resource R_2**
  - Claim edges exists, because the system knows in advance that P_1 and P_2 may both want to request R_2
- **Currently, there are no cycles in the graph so the system is in a safe state**
- **If process P_2 were to request resource R_2 and be granted that resource then ...**



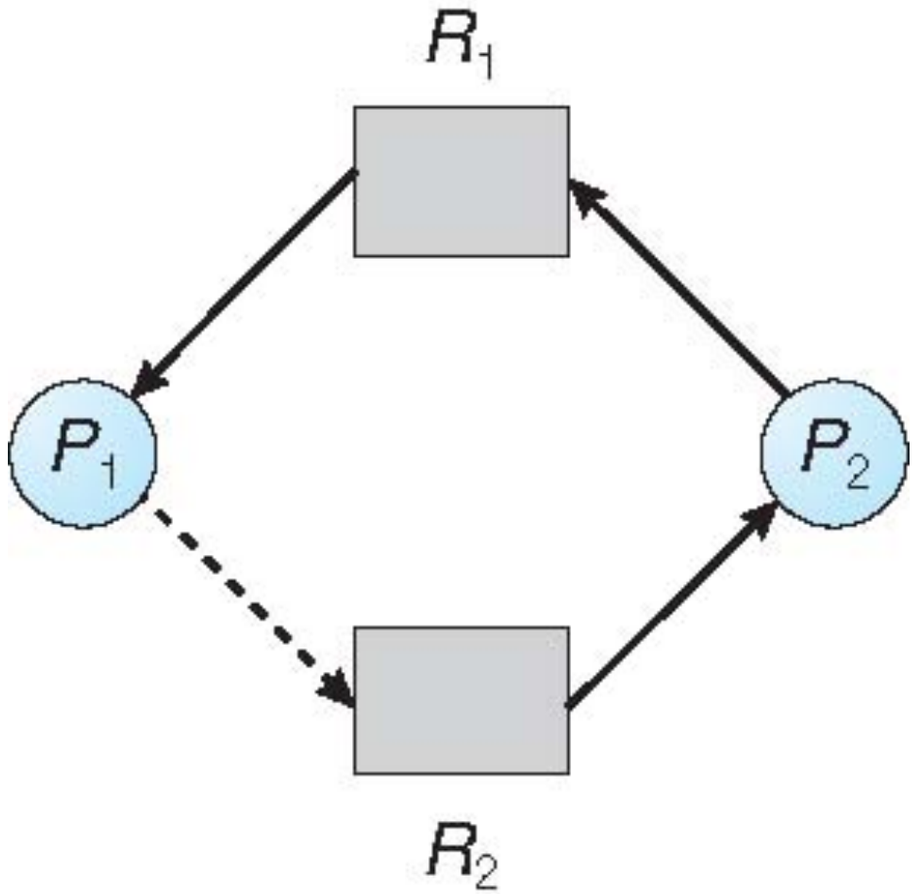
Unsafe State In Resource-Allocation Graph

- **If process P_2 were to request resource R_2 and be granted that resource then a cycle is formed in the resource-allocation graph**

- **The system is not yet deadlocked, but is in an unsafe state**

- **If process P_1 were to request resource R_2, then processes P_1 and P_2 would enter a deadlocked state**

- **To prevent an unsafe state and the possibility of deadlock, P_2 should not be allocated resource R_2 when it makes the request — process P_2 must wait for P_1 to release R_1 so that no cycles are created**



Banker's Algorithm

- **Use for deadlock avoidance when multiple instances of resources exist**
- **Each process must *a priori* claim maximum use (at process startup)**
- **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**
- **Banker's algorithm needs to know three things to work:**
  - How much of each resource each process could possibly request (max #)
  - How much of each resource each process is currently allocated
  - How much of each resource the system currently has available

Data Structures for the Banker's Algorithm

- **Let *n* = number of processes, and *m* = number of resource types**
  - *Available* -- Vector of length *m* that indicates the number of available resources of each type. If *Available[j]* = *k*, there are *k* instances of resource type R_j available
  - *Max* -- *n x m* matrix that defines the maximum demand of each process on each resource type. 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 that defines the number of resources of each type currently allocated to each process. If *Allocation[i,j]* = *k* then P_i is currently allocated *k* instances of R_j
  - *Need* -- *n x m* matrix that indicates the remaining resource needs of each process. 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]

Banker's Algorithm -- Safety Procedure

- **Used to determine whether or not a system is in a safe state**
- **Local Data Structures**
  - *Finish* -- a vector[*1 .. n*] , initialize as *false* for each process P_i
- **The procedure to check for a safe state:**

(1) Find a process P_i such that $Finish[i] = false$ and $Need_i \leq Available$

If P_i exists do the following:
$Available = Available + Allocation_i$
$Finish[i] = true$
Go back to Step #1

(2) If no such process P_i exists

If $Finish[i] = true$ for all processes P_i for $i=1 .. n$, then the system is in a safe state

Otherwise, processes with a *Finish[i]* value of *false* are in an unsafe state and can potentially deadlock

Example of Banker's Algorithm-Check Safe State

- **Consider a system with**
  - Five processes P_0 - P_4
  - Three resource types, A (10 instance), B (5 instances), and C (7 instances)
- **At some time t_0, the system looks like the following:**

	Allocation			Max		
	A	B	C	A	B	C
P_0	0	1	0	7	5	3
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

Available		
A	B	C
3	3	2

- **Is this a safe state?**

Example of Banker's Algorithm-Check Safe State (Cont.)

- **The *n x m* matrix *Need* is defined as**
  - *Need[i,j] = Max[i,j] – Allocation[i,j]*
- **The sequence <P_1, P_3, P_0, P_2, P_4> is a safe sequence (there are other safe sequences as well)**

	Allocation			*Max*			*Need*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	0	1	0	7	5	3	7	4	3
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

Available		
A	*B*	*C*
3	3	2

Banker's Algorithm -- Resource-Request

- **Used to determine whether requests for resources can be safely granted**
- **Local Data Structure**
  - $Request_i$ -- a vector[*1 .. m*], is a request vector for process P_i , the number of additional resources, of each type *1..m*, that process P_i is requesting
- **The procedure to determine if resource can be safely granted**

(1) If $(Request_i > Need_i)$ then raise an error -- process P_i is trying to acquire more resources than it initially declared it would need

(2) If $(Request_i > Available_i)$ then process P_i must wait because there are not currently sufficient resources available to satisfy the request

(3) Pretend to allocate the requested resources to process P_i

$Available = Available - Request_i$
$Allocation_i = Allocation_i + Request_i$
$Need_i = Need_i - Request_i$

(4) Check the safety of the state *that would result* from step #3
If step#3 produces a safe state, then the resources are allocated for real
If step#3 produces an unsafe state, then don't allocate the resources to process P_i, the process must wait

Example of Banker's Algorithm - P_1 Request

- **Since the system is in a safe state, use the Resource Request procedure to determine if the following request can be granted**
  - P_1 requests one instance of resource A, and two instances of resource C
    The request vector looks like --> $Request_1 = (1, 0, 2)$
  - First note if the resources are available to fulfill this request: $Request_1 \leq Available$

	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3
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

Available		
A	B	C
3	3	2

(1,0,2) ≤ (3,3,2)

(1,0,2) ≤ (1,2,2)

Example of Banker's Algorithm - P_1 Request (Cont.)

- **After determining that the resources are actually available, pretend to allocate them to process P_1**
- **Update the *Allocation*, *Need* and *Available* values**
- **Check to see if this new state is safe, if yes, then allocate resources for real**

	Allocation	*Max*	*Need*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3
~~P_1~~	~~2 0 0~~	~~3 2 2~~	~~1 2 2~~
P_1	3 0 2	3 2 2	0 2 0
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

Available
A B C
~~3 3 2~~
2 3 0

- **Is this a safe state?**

The sequence <P_1, P_3, P_4, P_0, P_2> is a safe sequence

Example of Banker's Algorithm - P_1 Request (Cont.)

- **What if P_4 requests (3, 3, 0) ?**

  Request cannot be granted, not enough resources available

- **What if P_0 requests (0, 2, 0) ?**

  Request cannot be granted, system ends up in an unsafe state

	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	2 3 0
P_1	3 0 2	3 2 2	0 2 0	
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	

Deadlock Detection

- **In systems that don't use deadlock-prevention or deadlock-avoidance, deadlock may occur**

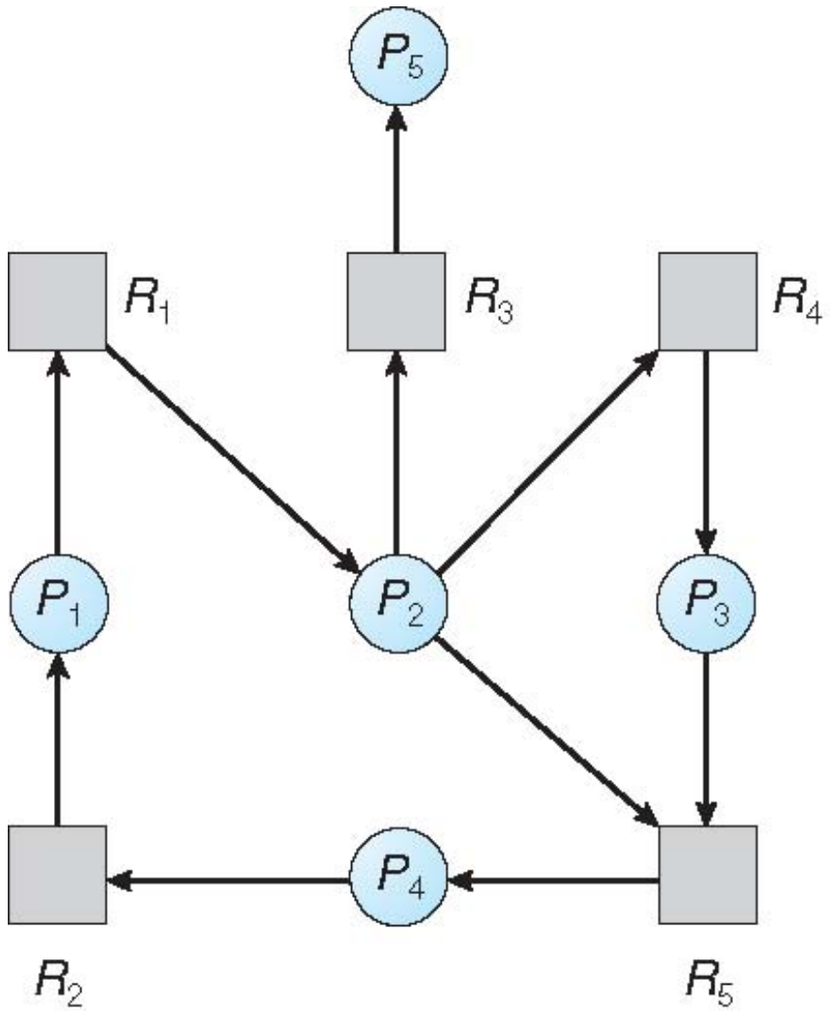
- **Such a system may provide:**
  - A deadlock detection algorithm that determines if a deadlock has occurred in the system
  - A deadlock recovery algorithm to recover from deadlock if it has occurred

- **As with deadlock avoidance, systems with multiple instances of a single resource type must be considered differently than systems with only single instance resources**

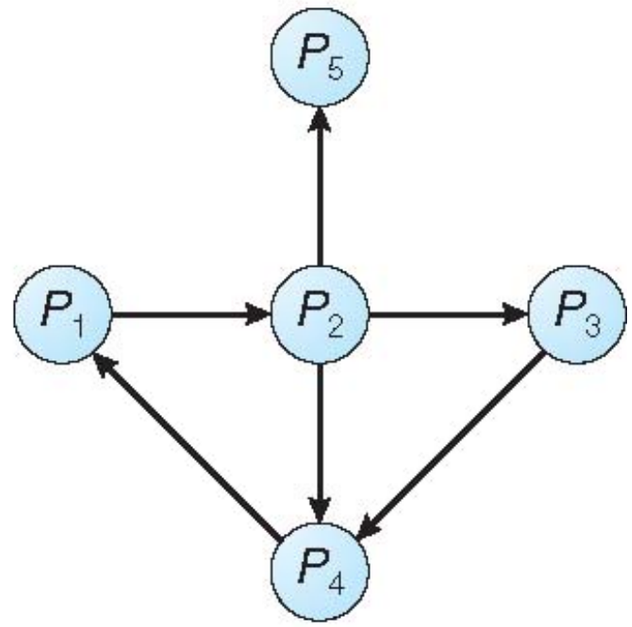
Single Instance of Each Resource Type

- **If all resources have only a single instance, use a wait-for graph for deadlock detection**
  - All Nodes are processes
  - An edge $P_i \rightarrow P_j$ exists if P_i is waiting for P_j to release a resource
- **Deadlock exists in the system if and only if the wait-for graph contains a cycle**
- **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

- **The wait-for graph cannot be used in systems with multiple instances of each resource type**
- **The deadlock detection algorithm for several instance contains similar data structures to the Banker's Algorithm**
- **Let *n* = number of processes, and *m* = number of resource types**
  - *Available* -- Vector of length *m* that indicates the number of available resource of each type. If available [*j*] = *k*, there are *k* instances of resource type R_j available
  - *Allocation* -- *n x m* matrix that defines the number of resources of each type currently allocated to each process. If *Allocation[i,j] = k* then P_i is currently allocated *k* instances of R_j
  - *Request* -- *n x m* matrix that 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

Deadlock Detection Algorithm

- **Used to determine which processes in a system are in a deadlocked state (very similar to Banker's Algorithm safety procedure)**

- **Local Data Structures**

  - *Finish* -- a vector[*1 .. n*], initialize as *false* for each process P_i where $Allocation_i \neq 0$, *true* otherwise

- **The procedure to check for a deadlocked state:**

  (1) Find a process P_i such that *Finish[i] = false* and $Request_i \leq Available$

  If P_i exists do the following:
  $Available = Available + Allocation_i$
  Finish[i] = true
  Go back to Step #1

  (2) If no such process P_i exists

  If *Finish[i] = false* for a processes P_i for *i=1 .. n*, then P_i is in a deadlocked state

Example of Deadlock Detection Algorithm

- **Consider a system with**
  - Five processes P_0 - P_4
  - Three resource types, A (7 instance), B (2 instances), and C (6 instances)
- **At some time T_0, the system looks like the following:**

	Allocation			Request		
	A	B	C	A	B	C
P_0	0	1	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

Available		
A	B	C
0	0	0

- **Is the system deadlocked?**

The sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish[i] = true* for all processes

Example of Deadlock Detection Algorithm (Cont.)

- **Continuing the previous example, suppose that process P_2 now requests one additional resource of type C**

- **At some time T_0, the system looks like the following:**

	Allocation			Request		
	A	B	C	A	B	C
P_0	0	1	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

Available		
A	B	C
0	0	0

- **Is the system deadlocked?**

Can only reclaim resources held by P_0. Insufficient resources to handle the requests from other processes. Deadlock exists consisting of processes P_1, P_2, P_3, and P_4

Deadlock Detection Algorithm Usage

- **When, and how often, to invoke the detection algorithm depends on:**
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?

- **If the detection algorithm is invoked every time a request for allocation cannot be granted:**
  - The algorithm can detect specifically which processes caused the deadlock
  - There is a lot of computational overhead when running the algorithm so frequently

- **If the detection algorithm is invoked less frequently (e.g. once per hour):**
  - It is difficult to determine which processes caused the deadlock
  - The computational overhead can be reduced

Deadlock Recovery

- **Once deadlock has been detected in a system there are several options for dealing with deadlock**
  - Notify the system user and let them deal with it
  - Have the system terminate one or more processes
  - Have the system preempt resources from one or more of the deadlocked processes

Deadlock Recovery: Process Termination

- **When terminating processes to recover from deadlock, resources are reclaimed from any terminated process**
  - Terminating a process means that work done by that process may be lost

- **Two options:**
  - Abort all deadlocked processes
    - Potential to lose a lot of work / computation time
  - Abort one process at a time until the deadlock cycle is eliminated
    - Lots of overhead since the deadlock-detection algorithm must be run after each process is terminated

Deadlock Recovery: Process Termination (Cont.)

- **If terminating processes, in what order should processes be terminated?**
- **Consider:**
  - 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?

Deadlock Recovery: Resource Preemption

- **Continually preempt the resources from processes and give those resources to other processes until the system is no longer deadlocked**

- **Must consider the following:**
  - How should victims be selected? And how can cost be minimized?
  - How should a preempted process get rolled back?
    - Would like to return the process to some safe state without having to completely restart the process. To do so requires some fancy bookkeeping by the system to know where the most recent safe state was prior to the deadlock.
  - How to ensure that starvation does not occur?
    - Don’t want resources to always get preempted from the same process