Deadlocks

Chapter 8

Chapter Objectives

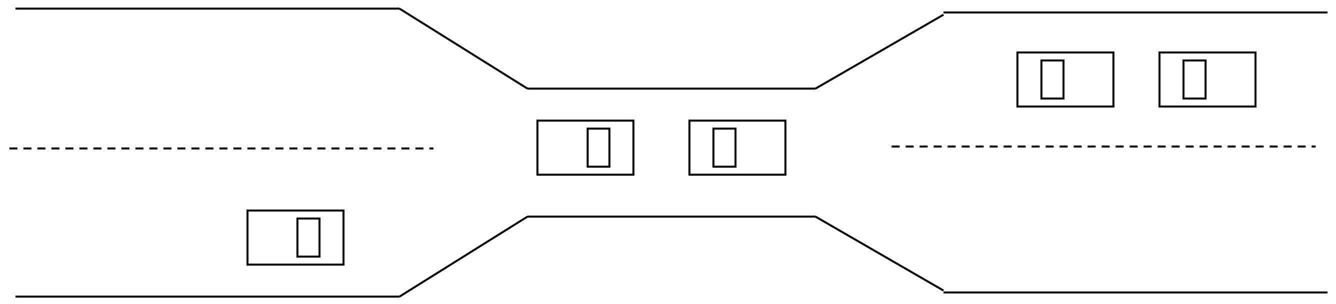
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for dealing with deadlocks in a computer system.

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 disk drives.
  - P_1 and P_2 each hold one disk drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

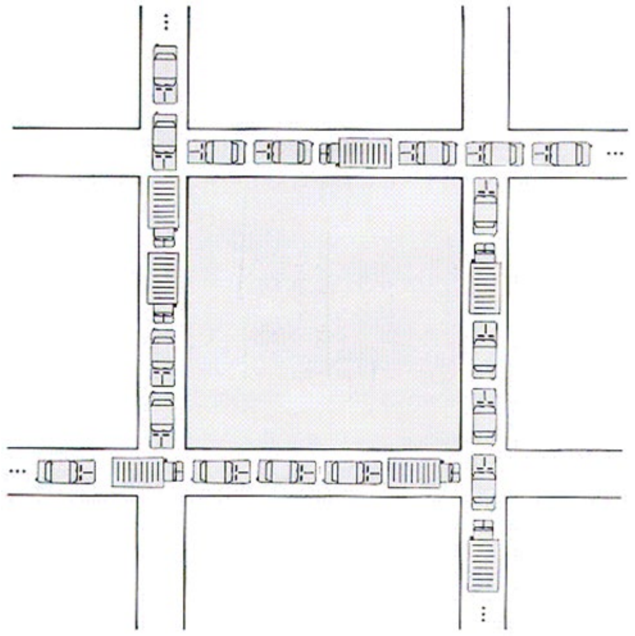
```
        P0                  P1
    wait (A);            wait(B)
    wait (B);            wait(A)
```

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.

Traffic Deadlock



System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices, locks, semaphores

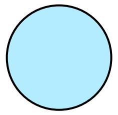
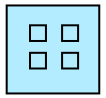
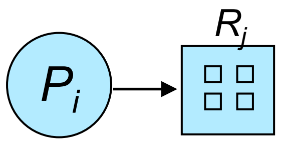
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

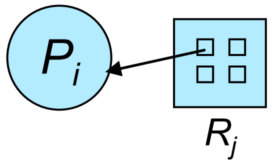
Resource-Allocation Graph

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

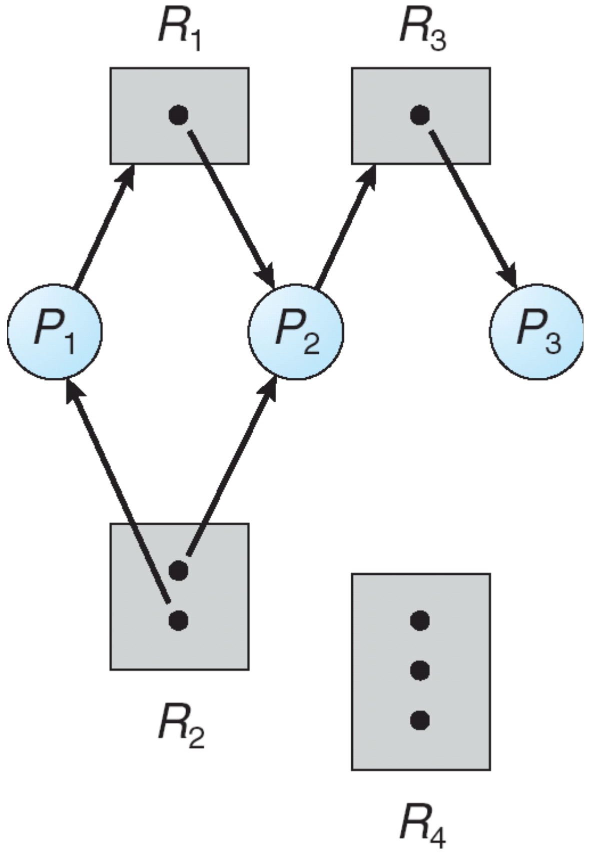
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed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

Resource-Allocation Graph (Cont.)

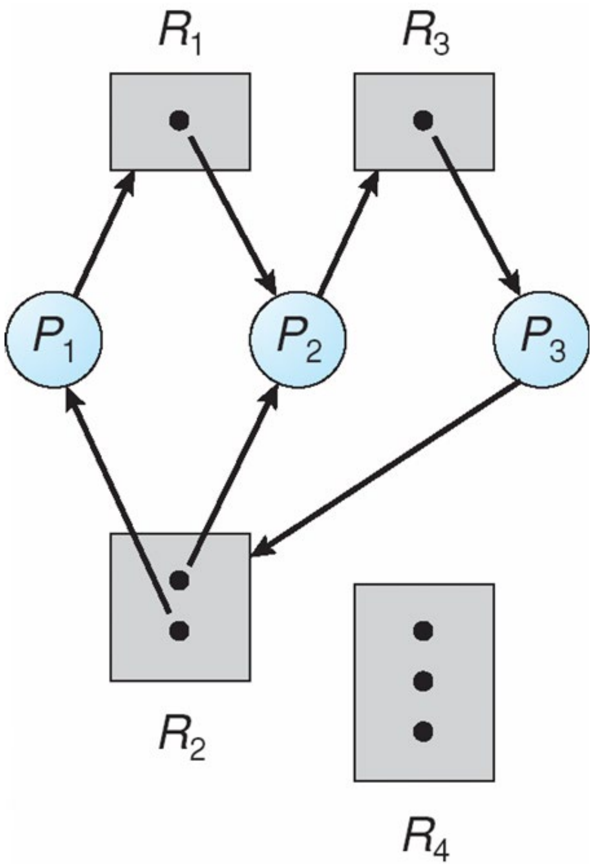
- Process 
- Resource Type with 4 instances 
- P_i requests an instance of R_j 

- P_i is holding an instance of R_j 

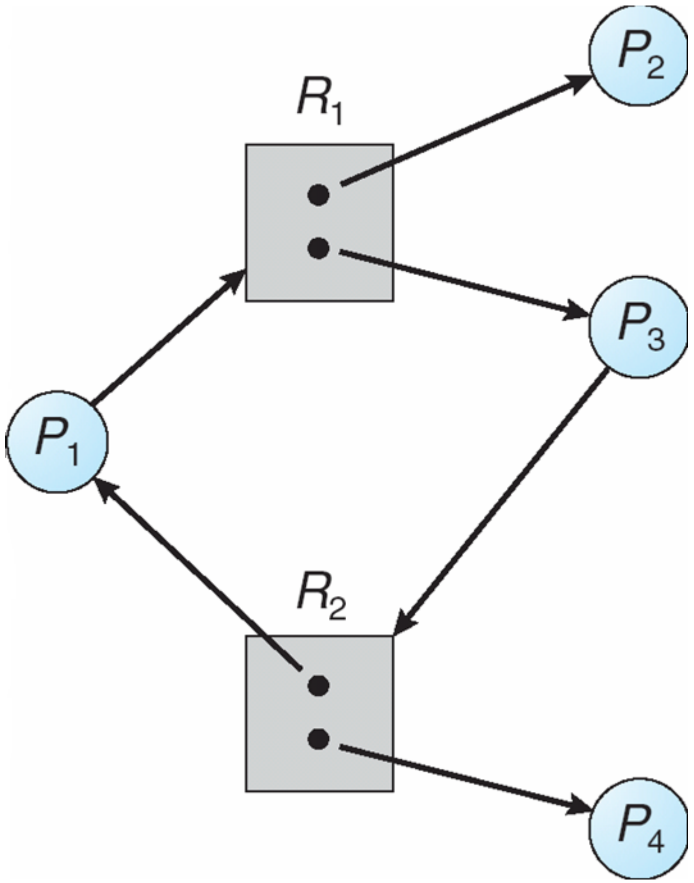
Example of a Resource Allocation Graph



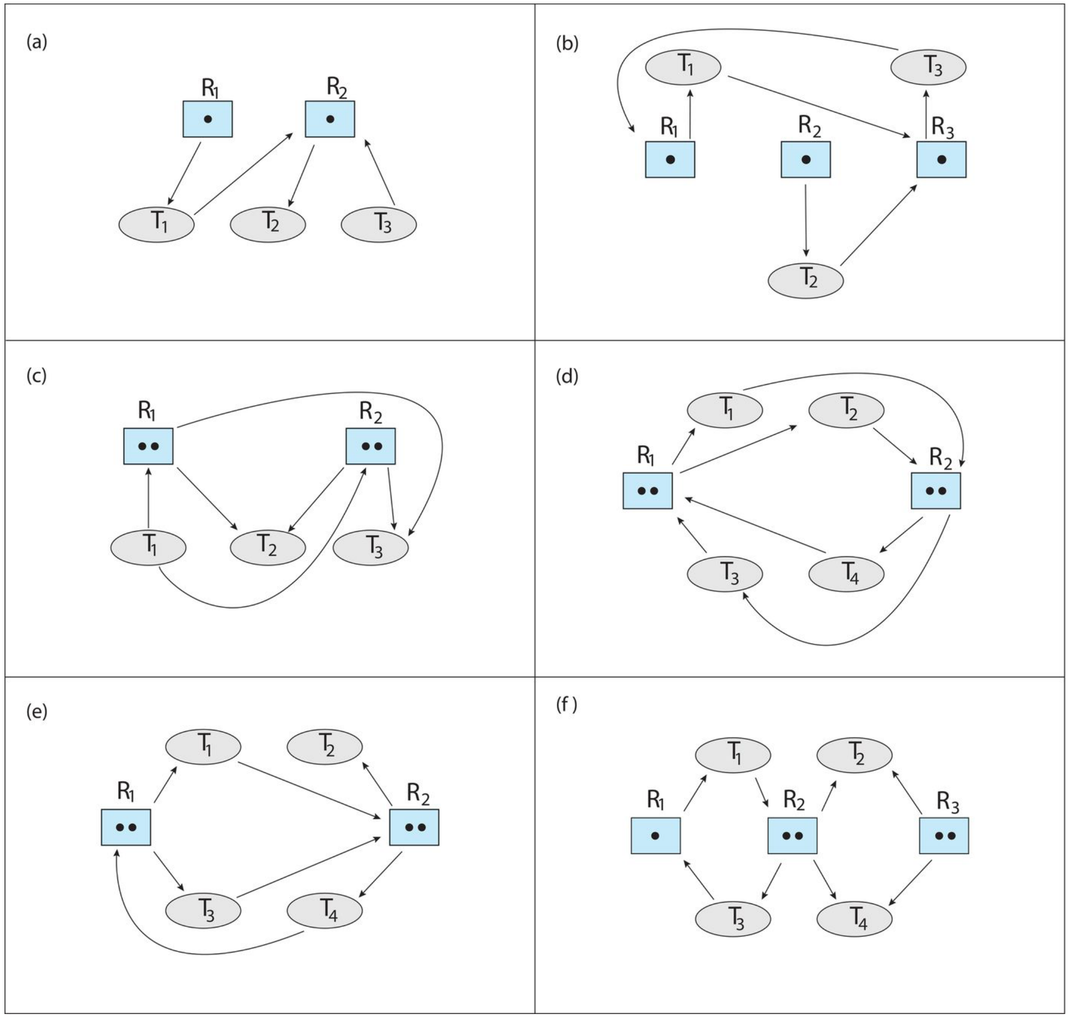
Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Examples



Which of those resource allocation graphs are deadlocked?

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.

Deadlock Necessary Conditions

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

Deadlock Prevention

Restrain the ways request can be made.

Make sure at least one of the four conditions for deadlock cannot hold:

- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable 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 (Cont.)

- **No Preemption**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Prevention

Prevent deadlock by making sure at least one of the four conditions for deadlock cannot hold.

Which of these conditions is easiest to give up to prevent deadlocks?

A. Mutual exclusion (make everything sharable)
B. Hold and wait (must get all resources at once)
C. No preemption (resources can be taken away)
D. Circular wait (total order on resource requests)
E. None of the above

Deadlock Avoidance

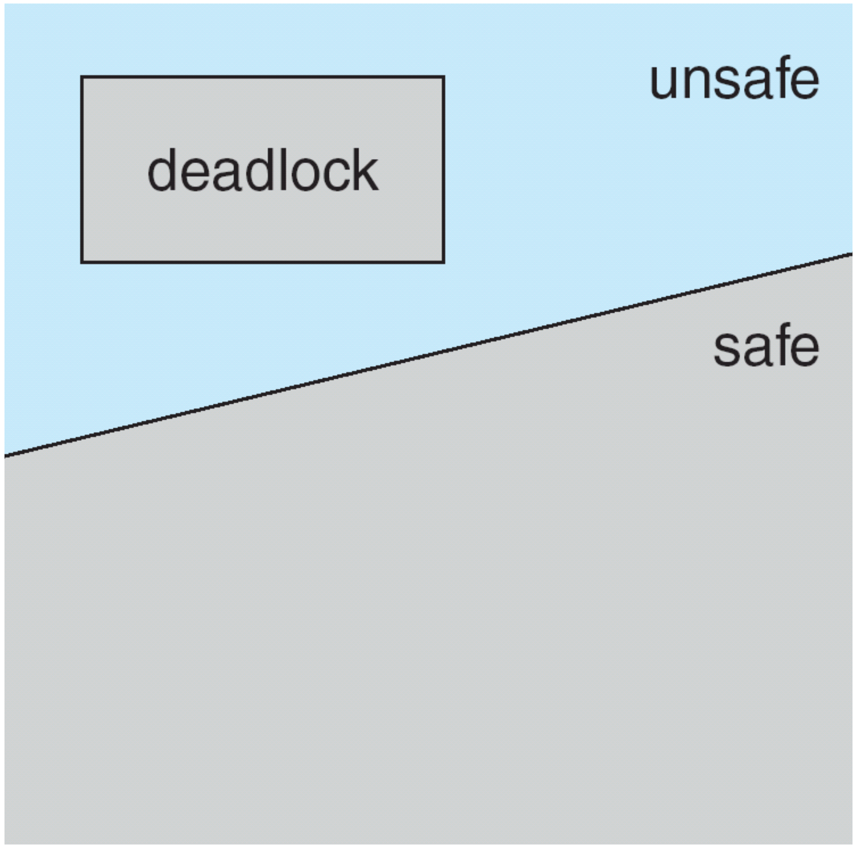
- Keep the system in a safe state that deadlock cannot happen
- Request for more resources can be granted only if the system remains safe
  - Requires that the system has some additional ***a priori*** information available.
  - Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need.

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 sequence $<P_1, P_2, \ldots, P_n>$ 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 + resources held by all the P_j, with $j < i$.
- That is:
  - 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, Unsafe , Deadlock State

- 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.



Avoidance Algorithms

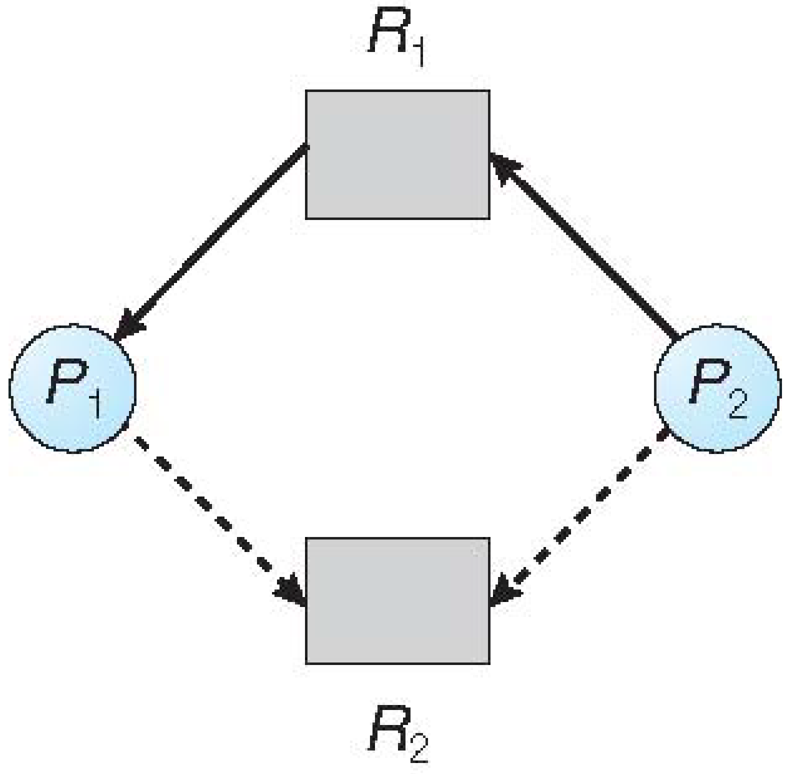
- Single instance of a resource type. Use a resource-allocation graph

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

Resource-Allocation Graph

Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.

Resources must be *claimed a priori* in the system.

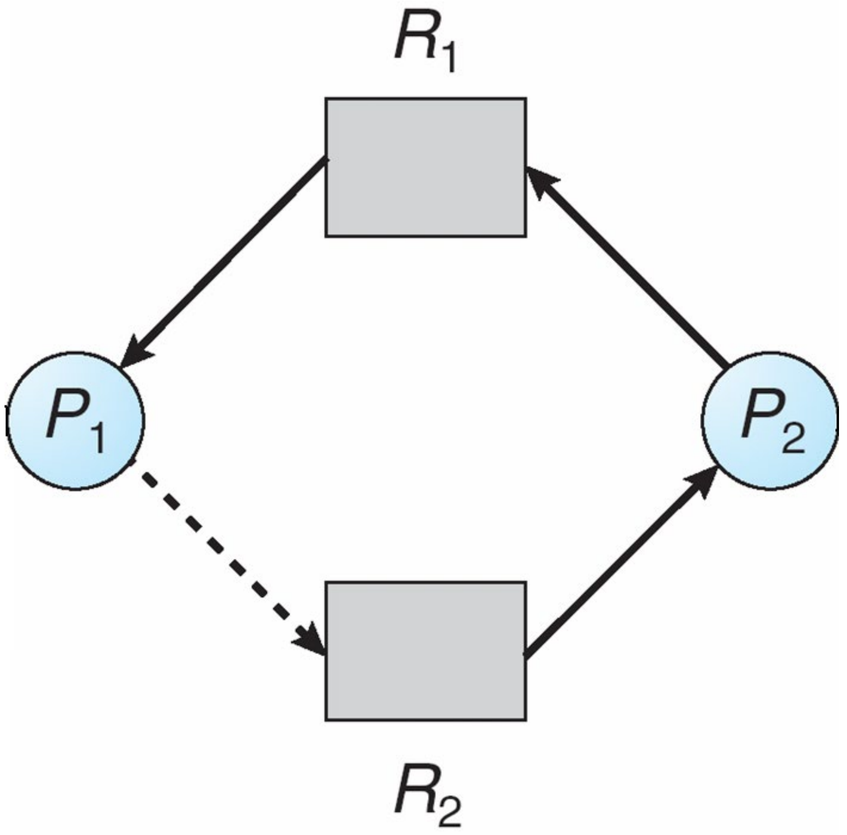


Is this safe?

Resource-Allocation Graph

Claim edge converts to request edge when a process requests a resource.

Request edge converted to an assignment edge when the resource is allocated to the process.



Is this safe?

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

- 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.

Example

Suppose a system has 12 tape drives and 3 processes.
At time t0, the system is as follows:

Process	Max need	Current allocated
P0	10	5
P1	4	2
P2	9	2

3 tape drives are unallocated.
Is the system safe?

Suppose process P2 is allocated another tape drive at time t1.
Is the system safe?

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 ***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_].

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	

Is the system safe?

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 and *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.

What is the complexity of this algorithm?

Available
A B C
3 3 2

- Snapshot at time T_0:

	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

Example (Cont.)

- 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

Available

A	*B*	*C*
3	*3*	*2*

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

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i.

If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

   Available = Available – Request;

   $Allocation_i = Allocation_i + Request_i$;

   $Need_i = Need_i – Request_i$;

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

Example: P_1 Request (1,0,2)

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ 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 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

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

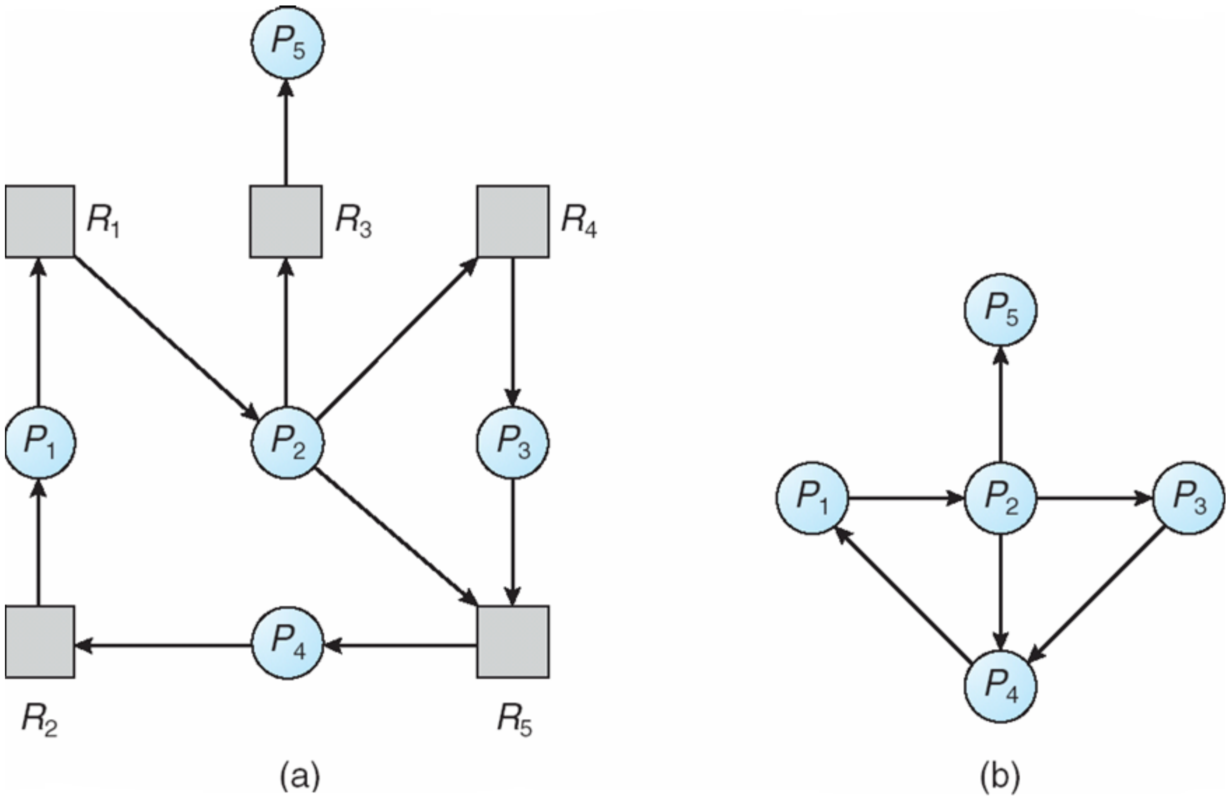
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
  - Do not need a prior claims
  - Need current allocations and requests
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph and Wait-for Graph



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, …, *n*, *Finish*[i] = false.
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. $Work = Work + Allocation_i$
   $Finish[i] = true$
   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 (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 process 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 need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

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
- Rollback – return to some safe state, restart process from that state
  - Database transactions
- Starvation – same proves may always be picked as victim
  - Include number of rollback in cost factor
  - Prefer the oldest process / transaction

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
  - Deadlock prevention
  - Deadlock avoidance
- 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 current operating systems, including UNIX.