The University of Texas at Arlington

Department of Computer Science and Engineering

CSE@UTA

CSE 3320: Operating Systems

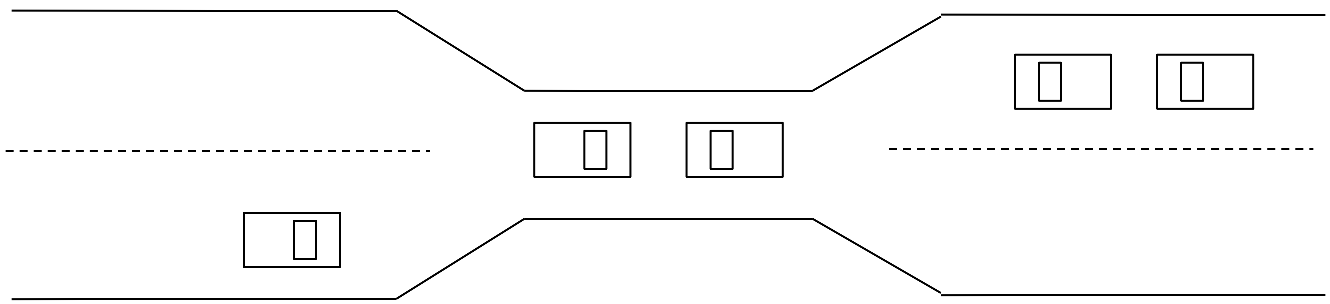
Deadlocks

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 1
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another 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.
- 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 R1, R2, . . ., Rm
  - CPU cycles, memory space, I/O devices
- Each resource type Ri has Wi instances
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

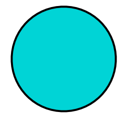
- Deadlock can arise if the following four conditions hold simultaneously.
  - Mutual exclusion
    - only one process at a time can use a resource.
  - Hold and wait
    - a process holding at least one resource is waiting to acquire additional resources held by other processes.
  - No preemption
    - 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 {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

Resource-Allocation Graph

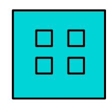
- A set of vertices V
  - V is partitioned into two types:
    - P = {P1, P2, …, Pn}, the set consisting of all the processes in the system.
    - R = {R1, R2, …, Rm}, the set consisting of all resource types in the system.
- A set of edges E
  - Request edge – directed edge P1 → Rj
  - Assignment edge – directed edge Rj → Pi

Resource-Allocation Graph (Cont.)

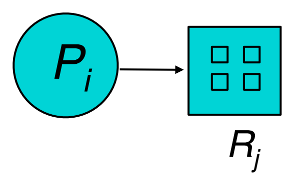
- Process



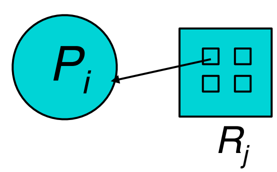
- Resource Type with 4 instances



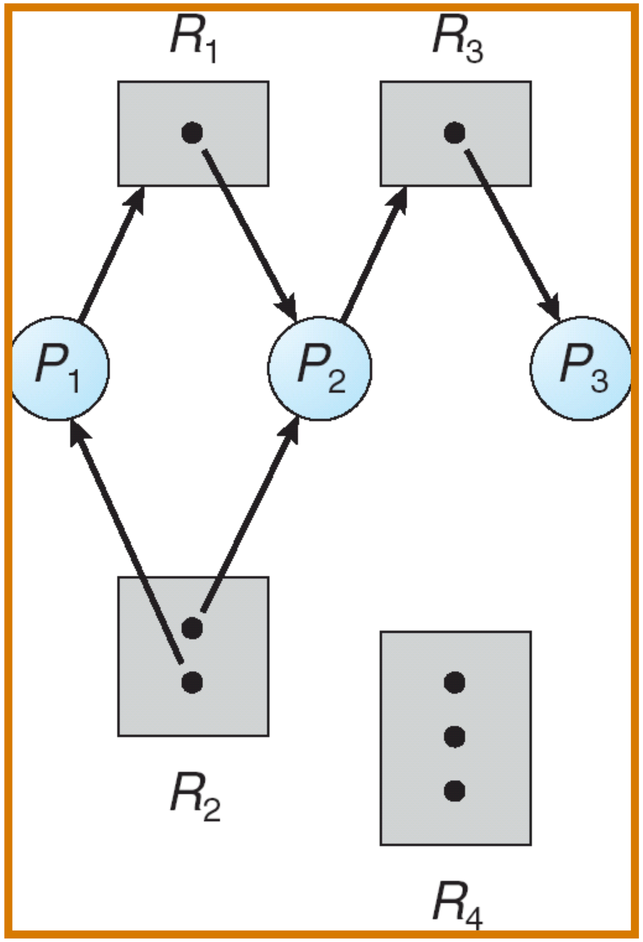
- P_i requests instance of R_j



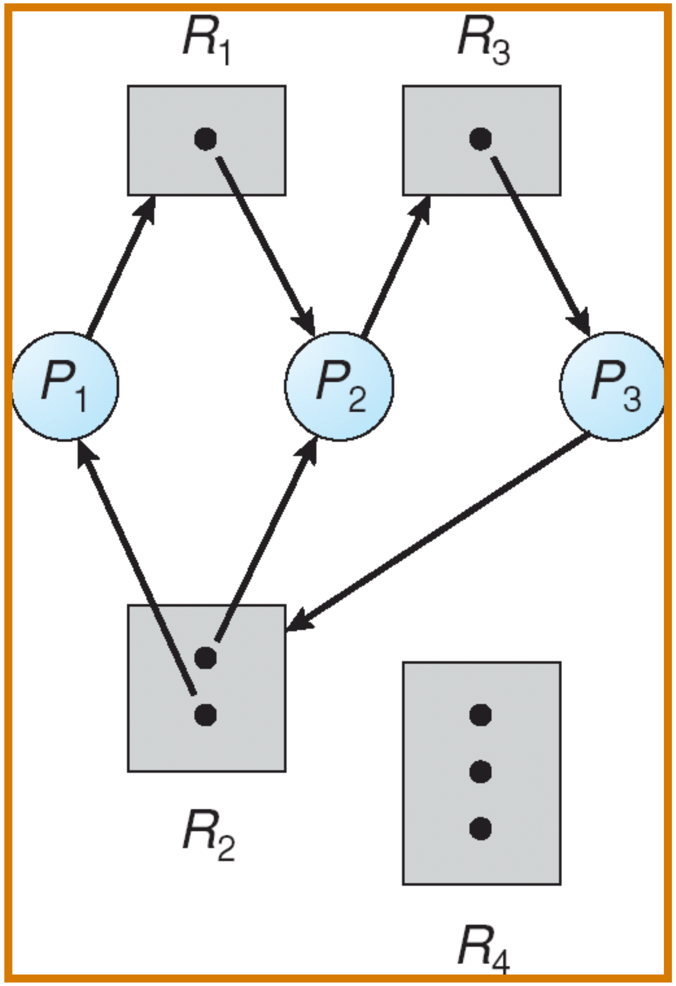
- P_i is holding an instance of R_j



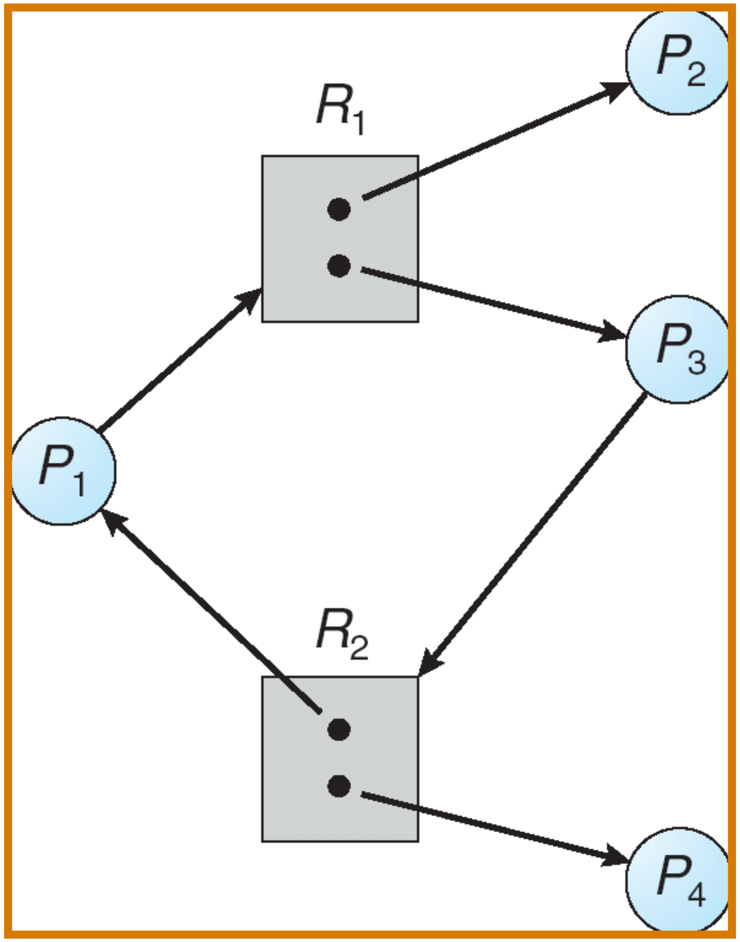
Resource Allocation Graph Example



Example of Deadlock



A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- 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.

Deadlock Prevention

- Restrain the ways request can be made
- 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 (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 Avoidance

- Requires that the system has some additional a priori information
- 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 <P1, P2, …, Pn> is safe if for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, 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.

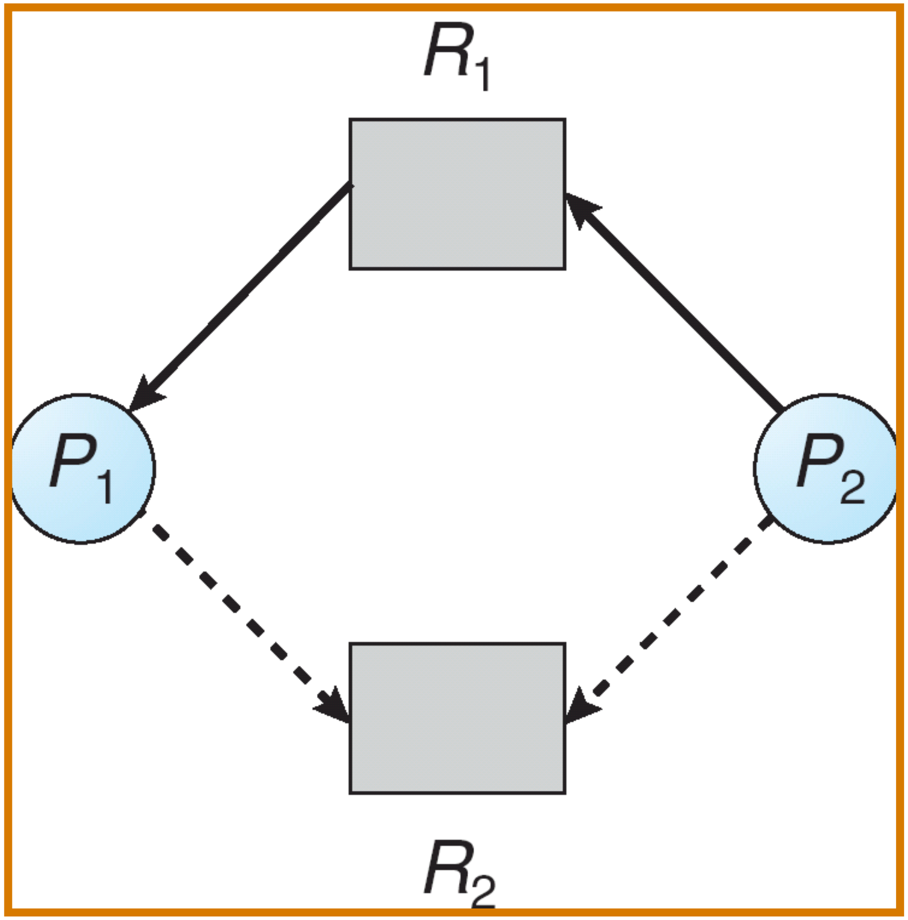
Basic Facts

- 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.

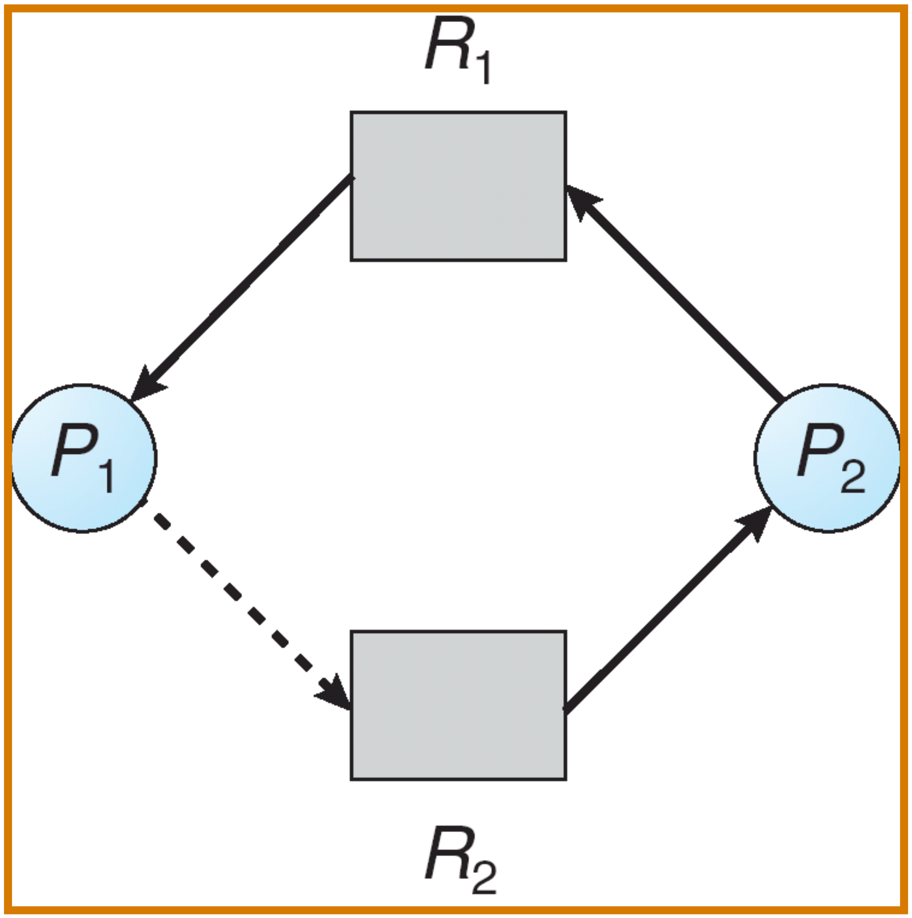
Deadlock Avoidance Algorithm

- Assume each resource has one instance
- Resource-allocation graph algorithm
  - Claim edge Pi → Rj indicated that process Pj may request resource Rj (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
  - The requesting process will need to wait if converting its request edge to an assignment edge will create a cycle in the graph
- Resources must be claimed a priori in the system.

Safe State



Unsafe State



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

- 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 Rj available.
  - **Max**: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj.
  - **Allocation**: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj.
  - **Need**: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj 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:
   1) Work = Available
   2) Finish [i] = false for i = 0,1, …, n-1.
2. Find an i such that both:
   1) (a) Finish[i] = false
   2) (b) Need[i] ≤ 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.

Example of Banker's Algorithm

- 5 processes $P_{0,\dots,}P_4$; 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 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.

Resource-Request Algorithm for Pi

- Request[i] = request vector for process Pi.
  - If Request[i,j] = k then process Pi wants k instances of resource type Rj.
- Description of the algorithm
  - 1. If Request[i] ≤ Need[i] go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
  - 2. If Request[i] ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available.
  - 3. Pretend to allocate requested resources to Pi by modifying the state as follows:
    - Available = Available - Request[i];
    - Allocation[i] = Allocation[i] + Request[i];
    - Need[i] = Need[i]– Request[i];
    - If safe ⇒ the resources are allocated to Pi.
    - If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Example P_1 Request (1,0,2) (Cont.)

- Check that Request ≤ Available (that is, $(1,0,2) \le (3,3,2) \Rightarrow$ 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 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> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

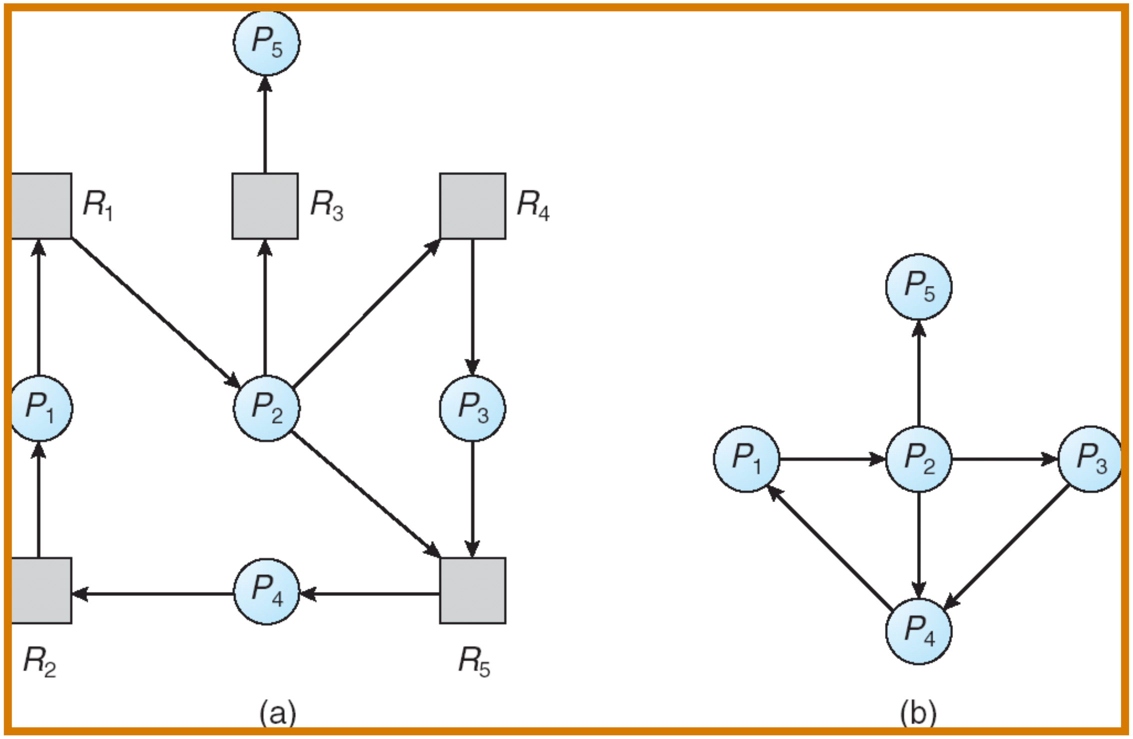
One Instance for Each Resource Type

- 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 Pi is requesting k more instances of resource type Rj.

Detection Algorithm

1. Let Work and Finish be vectors of length m and n, respectively Initialize:
   - (a) Work = Available
   - (b) For i = 0,1, …, n-1, if Allocation[i] ≠ 0, then Finish[i] = false;otherwise, Finish[i] = true.
2. Find an index i such that both:
   - (a) Finish[i] == false
   - (b) Request[i] ≤ 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 $P_0, \ldots, P_4$; 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 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 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: 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: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.