CSI 4500
Operating Systems

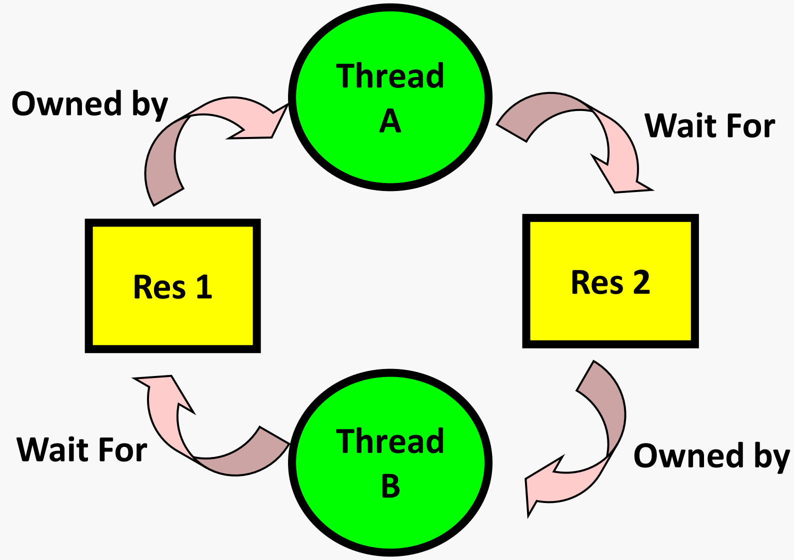
Deadlock

Goals for Today

- Understand Deadlock
- Understand Resource Allocation Graph
- Study techniques for
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection

Definitions

- Starvation: thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2
    Thread B owns Res 2 and is waiting for Res 1



- Deadlock $\Rightarrow$ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

Conditions for Deadlock

```
   Thread A          Thread B
semX.P();           semY.P();
semY.P();           semX.P();
semY.V();           semX.V();
semX.V();           semY.V();
```

- Deadlock won't always happen with this code
  - Have to have exactly the right timing order
- Deadlocks occur with multiple resources
  - Can't solve deadlock for each resource independently
- Example: System with 1 disk drive,1 printer and two threads
  - Each thread needs both of them to function
  - Thread 1 gets disk and Thread 2 gets the printer

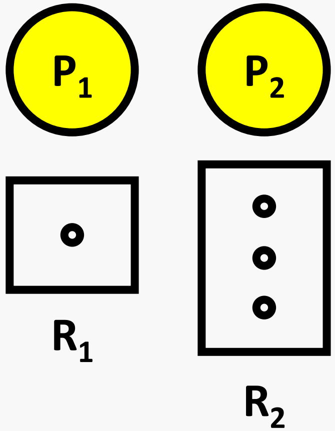
Four requirements for Deadlock

- **Mutual exclusion**
  - Only one thread at a time can use a resource.
- **Hold and wait**
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption**
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- **Circular wait**
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - T_1 is waiting for a resource that is held by T_2
    - T_2 is waiting for a resource that is held by T_3
    - …
    - T_n is waiting for a resource that is held by T_1

System Model & Resource-Allocation Graph

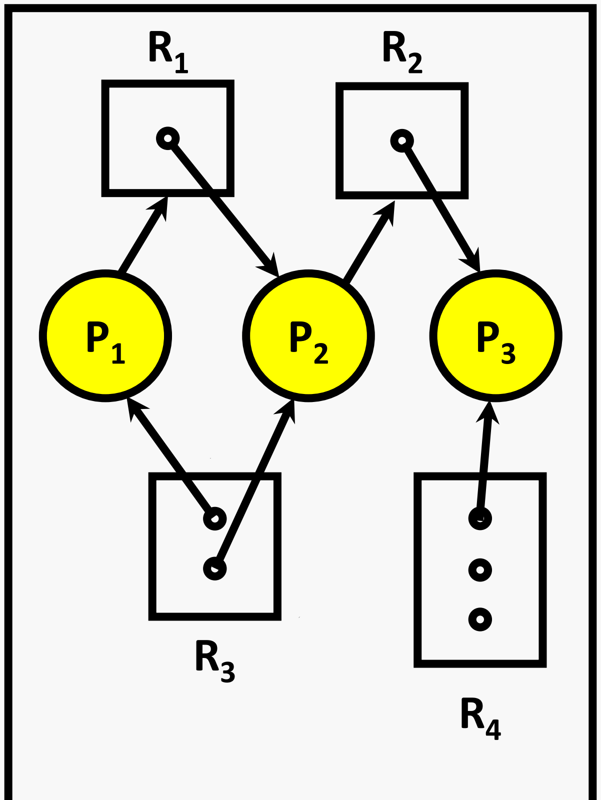
- **System Model**
  - A set of processes $P_1, P_2, \ldots, P_n$
  - Resource types $R_1, R_2, \ldots, R_m$
    - E.g., CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances.
  - Each process utilizes a resource as follows:
    - `Request() -> Use() -> Release()`
- **Resource-Allocation Graph:**
  - V is partitioned into two types:
    - P={$P_1, P_2, \ldots, P_n$}, R={$R_1, R_2, \ldots, R_m$}
  - E is partitioned into two types:
    - request edge – directed edge
      - $P_1 \rightarrow R_j$
    - assignment edge – directed edge $R_j \rightarrow P_i$

Symbols

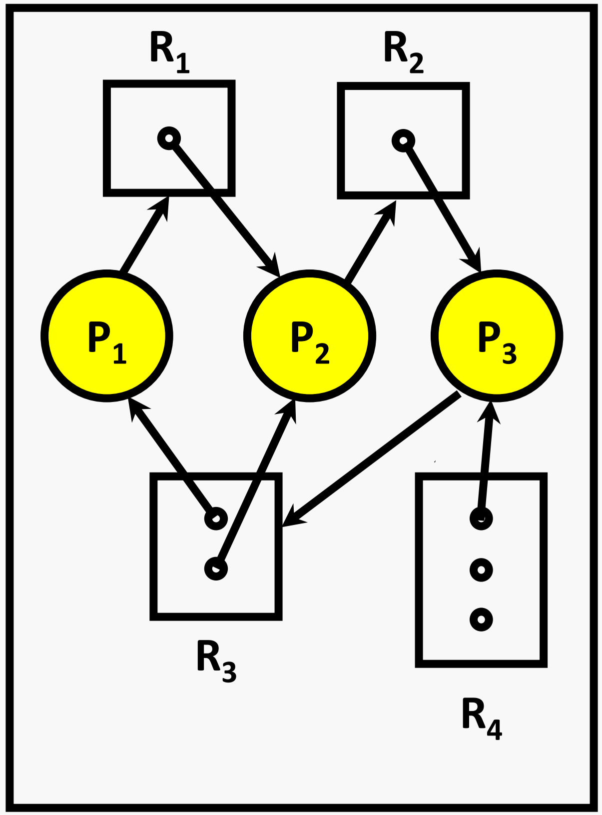


Resource Allocation Graph Examples

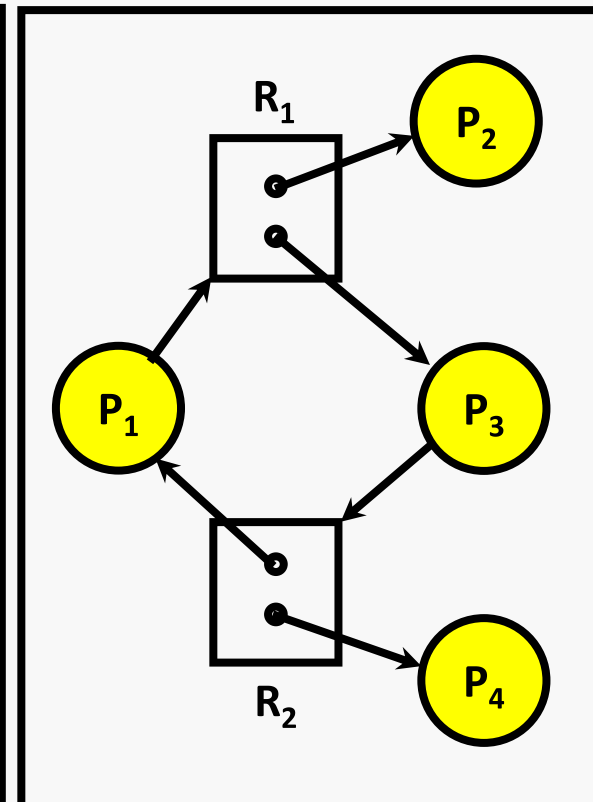
- Recall:
  - request edge – directed edge $P_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

Initial Found

- 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

- Ensure that system will NEVER enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that ***MIGHT*** lead to deadlock
- ALLOW system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempting resources and/or terminating tasks
- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX, windows

Oakland UNIVERSITY

Deadlock Prevention

- **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
  - Allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.
- **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**
  - a total ordering of all resource types
  - require that each process requests resources in an ordering strategy

Deadlock Avoidance

- **Resource-allocation state**
  - the number of available and allocated resources
  - the maximum demands of the processes.

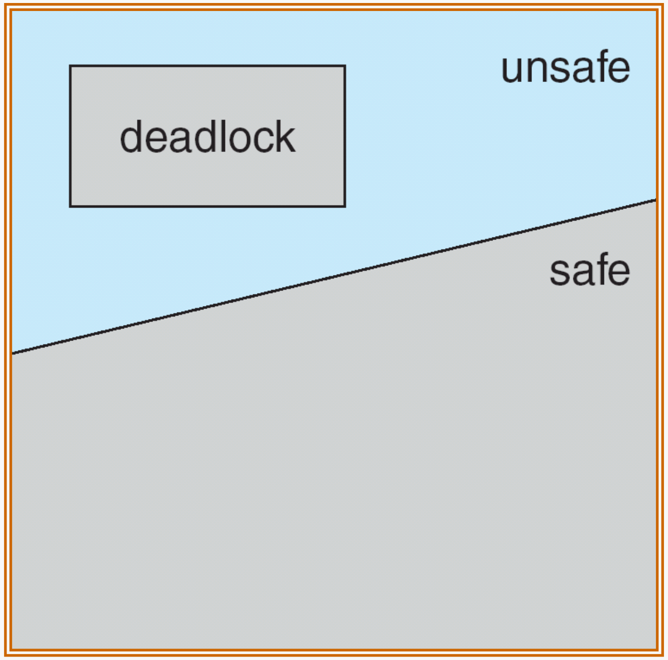
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

Safe State

- System is in 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 the 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.

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.



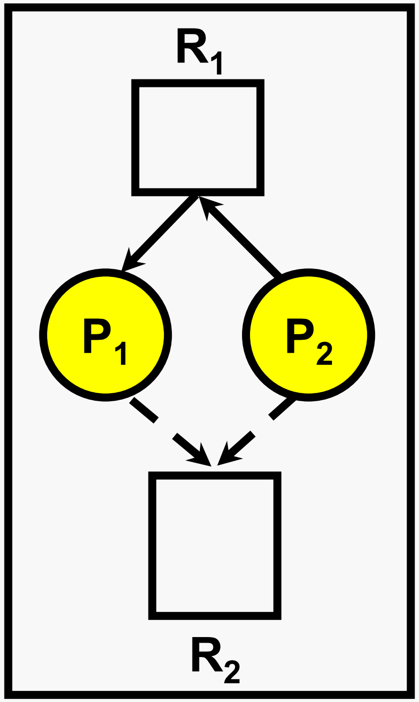
Deadlock Avoidance algorithms

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

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

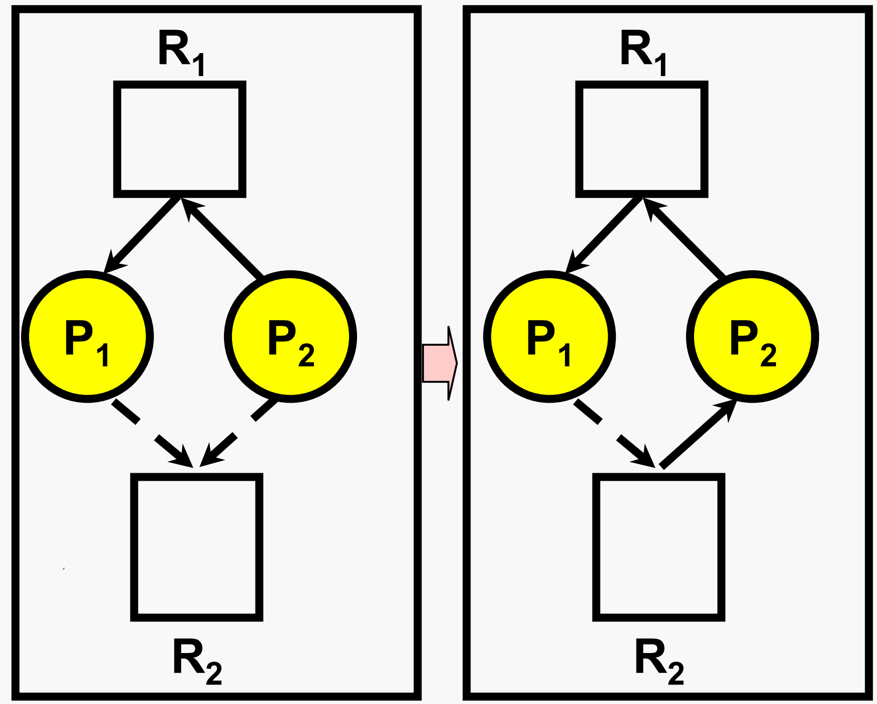
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j in the future.
  - represented by a dashed line.
- **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.
- When a resource is released by a process, **assignment edge** reconverts to a **claim edge**.
- Resources must be claimed *a priori* in the system.



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
  - P_i->R_j ➔ R_j->P_i



Resource Allocation leading to an Unsafe State
In Resource-Allocation Graph

Banker's Algorithm

- For one resource type, there are multiple instances.
- Each process must a priori claim maximum use.
  - This # <= total # of resources in the system
- 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.
- When system will determine whether the allocation of these resources will leave the system in a safe state.

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 x m matrix. 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. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.
- ***Need***: n x 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* = 0, 1, …, *n*- 1.

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

- Definition: $Request_i$ = 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:

   $Available = Available - Request;$

   $Allocation_i = Allocation_i + Request_i;$

   $Need_i = Need_i - Request_i;$

4. *Run Safety algorithm*
   - *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 (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	

Need

	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

- The content of the matrix *Need* is defined as follows.

  Need = Max – Allocation.

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

Example: P_1 Request (1,0,2)

- Check that Request $\leq$ Available, True or False?

	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?

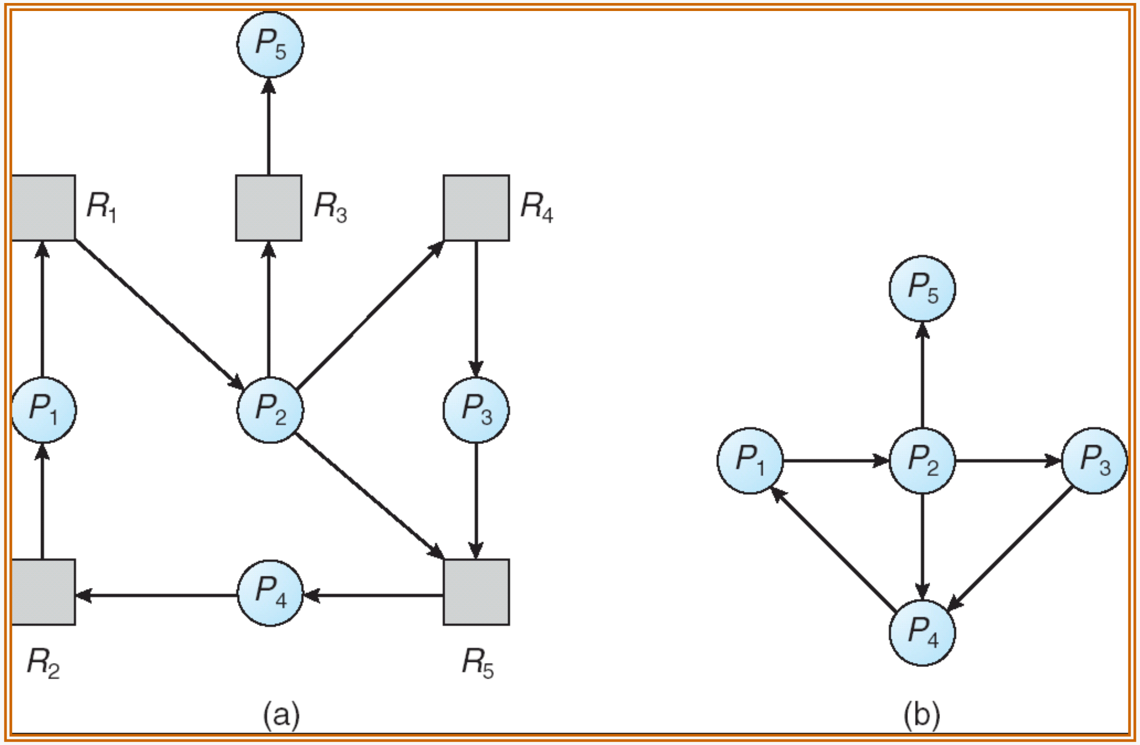
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
  - Single Instance of Resource Type
  - Multiple Instances of Resource Type
- Recovery scheme

Single Instance of Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
    - Rmove the resource nodes from resource-allocation graph
  - $P_i \rightarrow P_j$ 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

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.

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 1
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 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 for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Summary

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- Four conditions for deadlocks
  - Mutual exclusion
    - Only one thread at a time can use a resource
  - Hold and wait
    - Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - Resources are released only voluntarily by the threads
  - Circular wait
    - $\exists$ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern
- Techniques for addressing Deadlock
  - Allow system to enter deadlock and then recover
  - Ensure that system will *never* enter a deadlock
  - Ignore the problem and pretend that deadlocks never occur in the system
- Deadlock detection
  - Attempts to assess whether waiting graph can ever make progress