KOÇ UNIVERSITY

Deadlocks - II

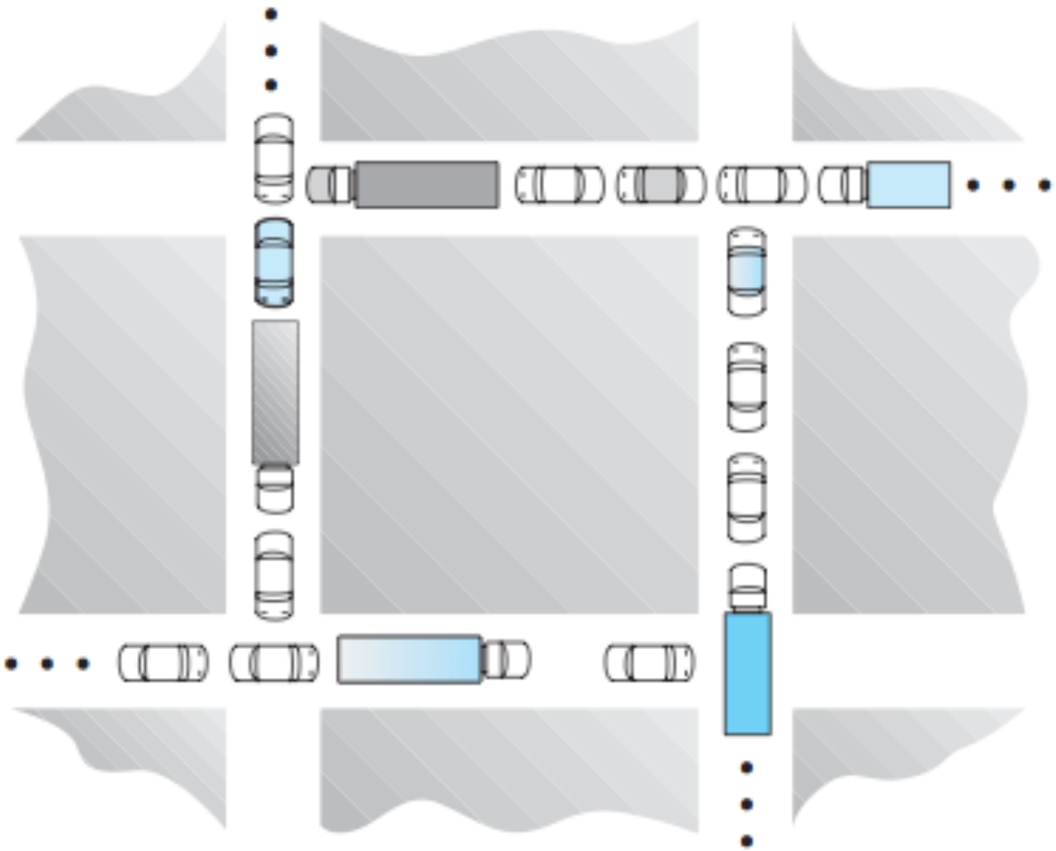
Didem Unat

Lecture 16

COMP304 - Operating Systems (OS)

Traffic Example

- Traffic only in one direction.
- Each street 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.



Resource-Allocation Graph

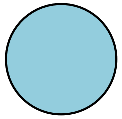
Deadlocks can be described in terms of a directed graph

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

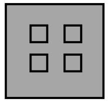
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- **request edge:** directed edge $P_i \rightarrow R_j$
- **assignment edge:** directed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

Resource-Allocation Graph

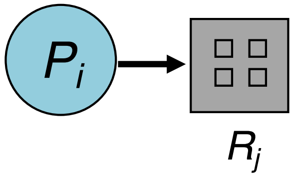
- Process



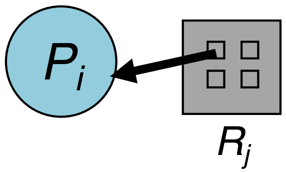
- Resource type with 4 instances



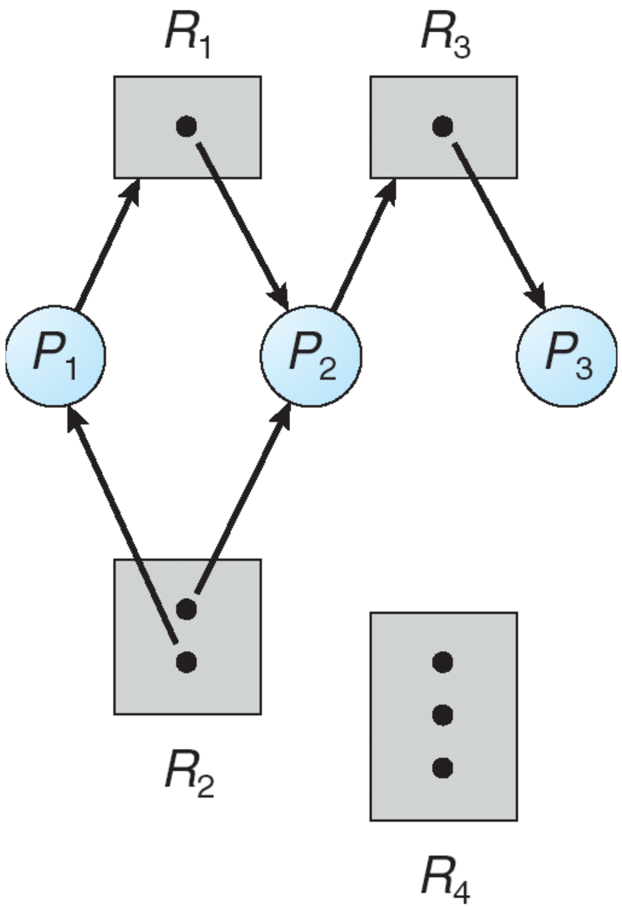
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example Resource Allocation Graph



Resource instances:

- One instance of resource type *R1*
- Two instances of resource type *R2*
- One instance of resource type *R3*
- Three instances of resource type *R4*

Process states:

- Process *P1* is holding an instance of resource type *R2* and is waiting for an instance of resource type *R1*.
- Process *P2* is holding an instance of *R1* and an instance of *R2* and is waiting for an instance of *R3*.
- Process *P3* is holding an instance of *R3*.

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 Avoidance

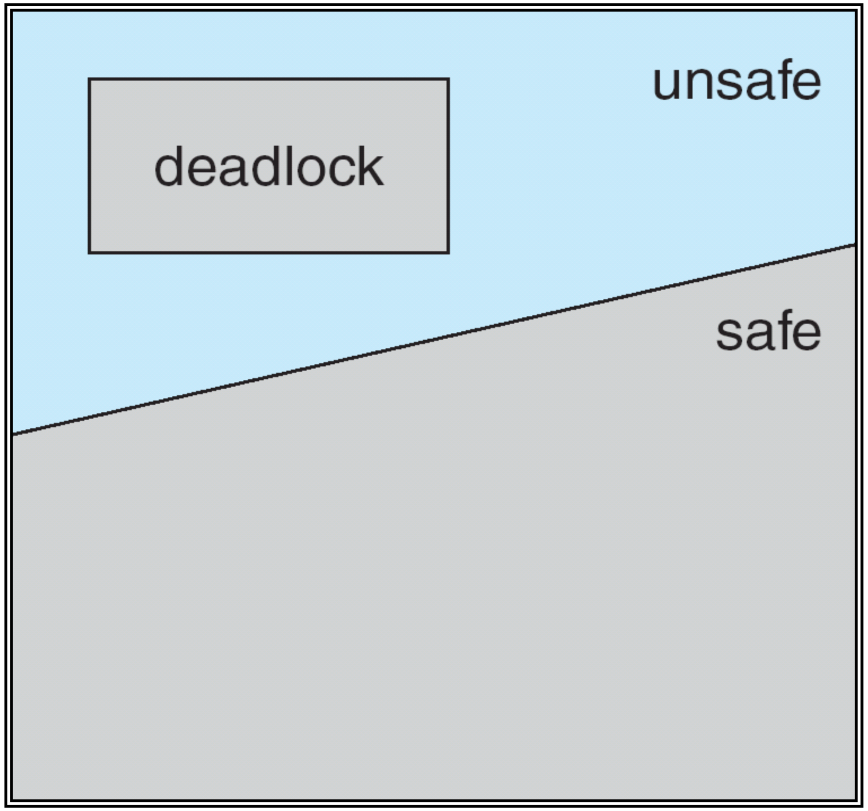
- The **system knows** the complete sequence of requests and releases for each process.
  - Priori information is available

- The **system decides** for each request whether or not the process should wait in order to avoid a deadlock.

- Each **process declares** the maximum number of resources of each type that it may need.

- The system should always be at a **safe state**.

Safe, Unsafe , Deadlock State



We can define avoidance algorithms that ensure the system will never deadlock.

A resource is granted only if the allocation leaves the system in a **safe state.**

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.

Safe State

- System is in safe state if there exists a safe sequence of all processes.

- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if 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 $j<i$.**
  - If P_i's 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 State Sequence Example

- 12 resources, three processes ($P_{0,}$ $P_{1,}$ and P_2)

	Max Needs	Currently Held #Resources at t_0
P0	10	5
P1	4	2
P2	9	2

- 9 resources are currently used, 12- 9 = 3 available
- Current state is safe because a safe sequence exists *<P1, P0, P2 >*
  - p1 can complete with current resources
  - p0 can complete with current+p1
  - p2 can complete with current +p1+p0

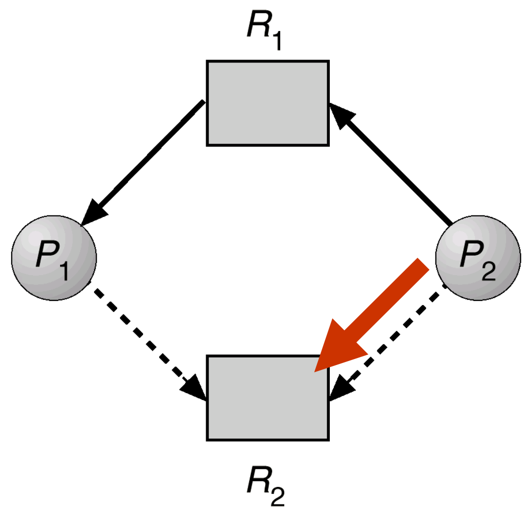
Deadlock Avoidance Algorithms

- Single instance of a resource type: Use a **resource-allocation graph**

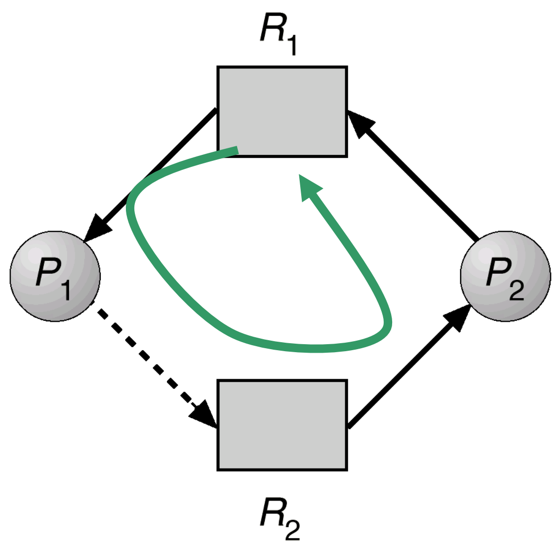
- Multiple instances of a resource type: Use the **banker's algorithm**

Resource-Allocation Graph Algorithm

- Works only if each resource type has one instance
- Algorithm
  - **Add a claim edge** $P_i \rightarrow R_j$ indicating that process P_i may request resource R_j;
  - Represented by a dashed line.



- A request $P_i \rightarrow R_j$ can be granted only if
  - Adding assignment edge $R_j \rightarrow P_i$ does not result in a cycle in the graph



A cycle indicates an unsafe state.

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

Convert P_i -> R_j *to* R_j -> P_i only if no cycle

Banker’s Algorithm

- Multiple instances of resource types.
- Each process must a priori claim **maximum use.**
- When a process requests a resource it may have to wait.
- When a process requests a set of resources:
  - Will the system be at a safe state after the allocation?
  - Yes → Grant the resources to the process.
  - No → Block the process until the resources are released by some other process.

Banker's Algorithm

```
n: integer    # of processes
m: integer    # of resource-types


Available[1:m]
 #Available[j] is # of avail resources of type j
Max[1:n,1:m]
 #Max demand of each Pi for each Rj
Allocation[1:n,1:m]
 #current allocation of resource Rj to Pi
Need[1:n,1:m]
 #max # resource Rj that Pi may still request
 #Need[i,j] = Max[i,j] - Allocation[i,j]
```

Banker's Algorithm

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

If $request_i$ > $need_i$
    error (asked for too much)
If $request_i$ > available
    wait(can't supply it now)
Resources are available to satisfy the request
    Let's assume that we satisfy the request, then
        available = available - $request_i$
        $allocation_i$ = $allocation_i$ + $request_i$
        $need_i$ = $need_i$ – $request_i$

Now check if this would leave us in a safe state:
    If yes, grant the request
    If no, leave the state as is and cause process to wait

Banker's Safety Algorithm

Algorithm for finding out whether or not a system is in a safe state

```
Initialize:
  Work[1:m] = Available[1:m] //how many resources available
  Finish[1:n] = false        //none of the processes finished yet

Step 1:Find a process i such that both:
    (a) Finish[i] = false
    (b) Need_i ≤ Work
    If no such i exists, go to step 3.

Step 2: Found an i
    Finish[i] = true         //done with this process
    Work = Work + Allocation_i
    go to step 1

Step 3:
    If Finish[i] == true for all i, then
       the system is in a safe state.
    Else
       Not safe
```

Requires $O(mn^2)$ operations to decide whether a state is safe

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		
	A	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3

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

Example (Cont.)

- The content of the matrix Need is defined to be
  Need = Max – Allocation.

	Allocation			Max			Need		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	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.

Example P_1 Requests (1,0,2)

- *If P_1* requests (1,0,2), should we grant it?
  - Check that $\text{Request}_1 \leq \text{Need}_1$: $(1,0,2) \leq (1,2,2) \Rightarrow$ true.
  - Check that $\text{Request}_1 \leq \text{Available}$: $(1,0,2) \leq (3,3,2) \Rightarrow$ 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 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 $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement $\Rightarrow$ grant request of P_1.
  - Can request for (0,2,0) by P_3 be granted in this state?
  - Can request for (3,3,0) by P_4 be granted in this state?
  - Can request for (0,2,0) by P_0 be granted in this state?

Example P_0 Requests (0,2,0)

- P_0 Requests (0,2,0), should we grant the resources?
  - Check that $\text{Request}_1 \leq \text{Need}_1$: $(0,2,0) \leq (7,4,3) \Rightarrow$ true.
  - Check that $\text{Request}_1 \leq \text{Available}$: $(0,2,0) \leq (2,3,0) \Rightarrow$ true.

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	**0 3 0**	**7 2 3**	**2 1 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	

- P0's request will be denied because the resulting state is unsafe

Deadlock Detection

- We saw that you can **prevent** deadlocks
  - By negating one of the four necessary conditions.
- We saw that the OS can schedule processes in a careful way so as to **avoid** deadlocks.
  - Using a resource allocation graph.
  - Banker's algorithm.

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

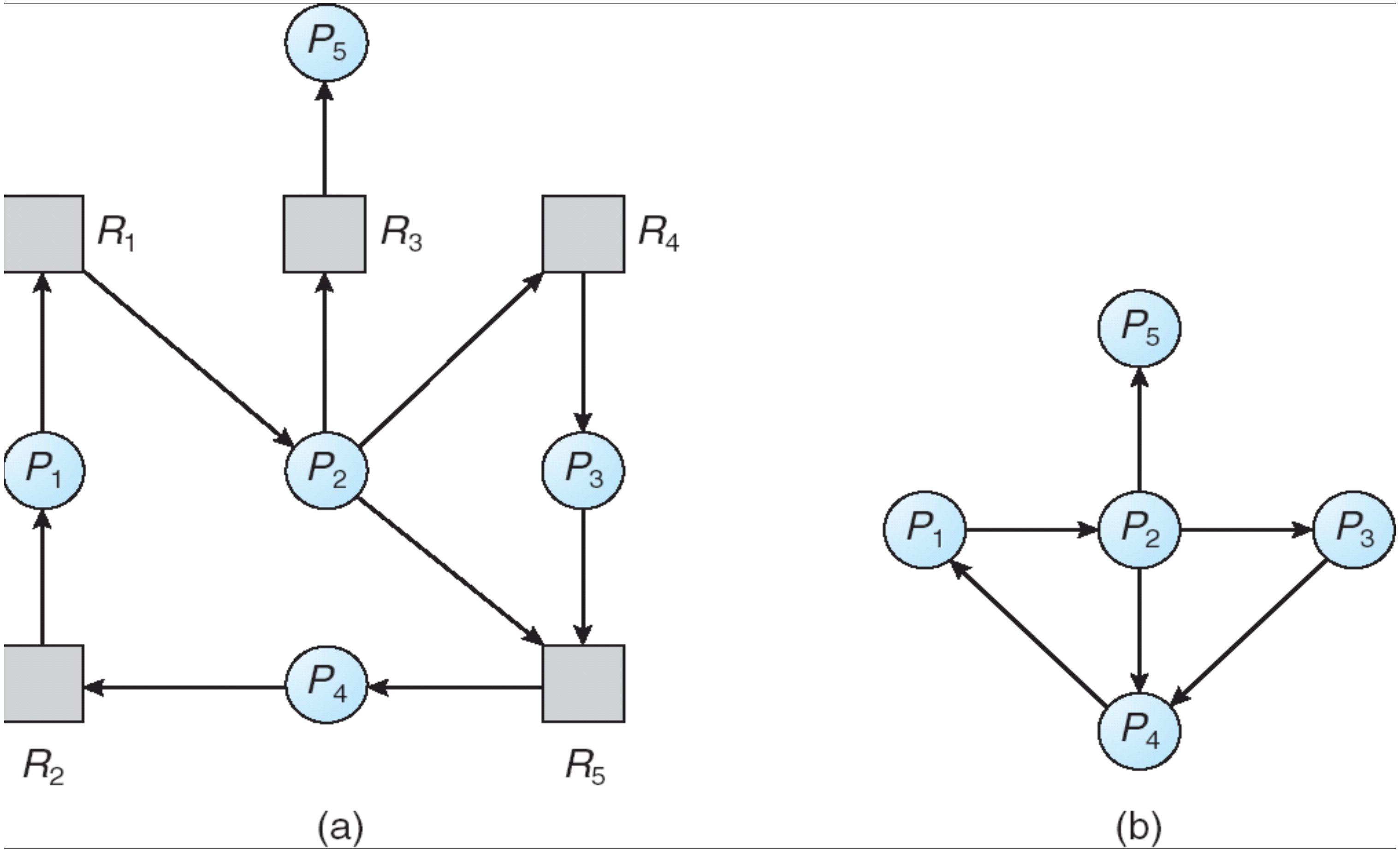
Single instance of each resource type

- Maintain **wait-for** graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j (to release a resource that P_i needs)

- 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 and Wait-for Graphs



Resource-Allocation Graph

Corresponding wait-for graph

Multiple instances of a resource type

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

```
n: integer    # of processes
m: integer    # of resource-types

Available[1:m]
 #Available[i] is # of avail resources of type i
Request[1:n,1:m]
 #Current demand of each Pi for each Rj
Allocation[1:n,1:m]
 #current allocation of resource Rj to Pi
finish[1:n]
 #true if Pi's request can be satisfied
```

Detection Algorithm

Let Work and Finish be vectors of length m and n, respectively

```
1. Initialize:
     (a)Work = Available
     (b) For i=1:n,
           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.
3. Work = Work + Allocation[i]
   Finish[i] = true
   go to step 2.

4.  If Finish[i] == false, for some i, then
      the system is in deadlock state with Pi is deadlocked.
```

Requires an order of $O(mn^{2})$ 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	

No deadlock

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

System is deadlocked

- 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.

Deadlock Detection

- How often should we call deadlock detection algorithm?
  - When there is a low CPU utilization
  - Periodically but not too often
- **Recovery from Deadlock**
- **Killing** one/all deadlocked processes
  - Keep killing processes, until deadlock broken
  - Repeat the entire computation
- **Preempt** resource/processes until deadlock broken
  - Selecting a victim (based on # resources held, how long executed)
  - 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.

Acknowledgments

- These slides are adapted from
  - Öznur Özkasap (Koç University)
  - Operating System and Concepts (9th edition) Wiley
  - Cornell University