BILKENT UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
CS342 OPERATING SYSTEMS

LECTURE 7
DEADLOCKS
(CHAPTER 7)

Dr. İbrahim Körpeoğlu
http://www.cs.bilkent.edu.tr/~korpe

References

- *The slides here are adapted/modified from the textbook and its slides: Operating System Concepts, Silberschatz et al., 7th & 8th editions, Wiley.*

REFERENCES

- Operating System Concepts, 7th and 8th editions, Silberschatz et al. Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum, 3rd edition, 2009.

Outline

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Objectives

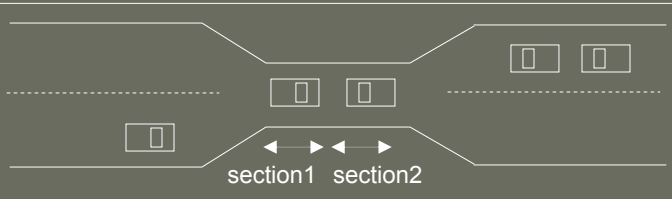
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

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

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
- Note – Most OSes do not prevent or deal with deadlocks

System Model

- Resource types R_1, R_2, . . ., R_m
  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request** (may cause the process to wait)
  - **use**
  - **release**

Deadlock Characterization

- **Deadlocks can arise if four conditions hold simultaneosly**
  - **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 $\{P_0, P_1, \ldots, P_N, P_0\}$ 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.

Resource-Allocation Graph

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

V is partitioned into two types:

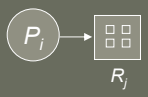
$P = \{P1, P2, \ldots, Pn\}$, the set consisting of all the processes in the system

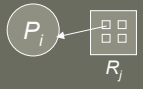
$R = \{R1, R2, \ldots, Rm\}$, the set consisting of all resource types in the system

request edge – directed edge $Pi \rightarrow Rj$

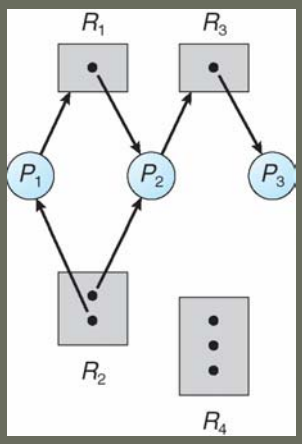
assignment edge – directed edge $Rj \rightarrow Pi$

Resource-Allocation Graph (Cont.)

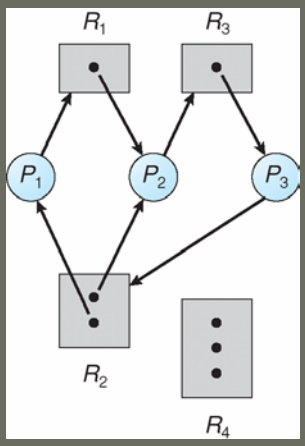
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

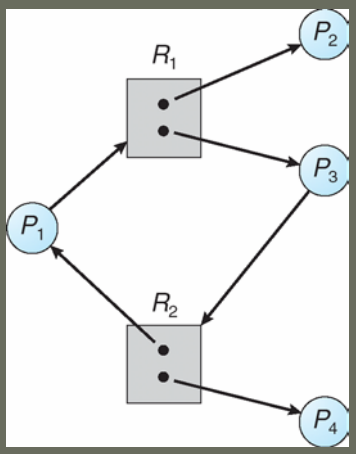
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With 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
  - Deadlock Prevention or Deadlock Avoidance methods
- Allow the system to enter a deadlock state and then recover
  - Deadlock Detection needed
- Ignore the problem and pretend that deadlocks never occur in the system;
  - Used by most operating systems, including UNIX
  - OS does not bother with deadlocks that can occur in applications

Deadlock Prevention

Basic Principle: Restrain the ways requests 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 Prevention (Cont.)

- All resources are ordered and assigned an integer number
  - A process can request resources in increasing order of enumeration

Example:

Process 1	Process 2	Process 3
Request R2	Request R1	Request R3
Request R4	Request R2	Request R4
	Request R3	

Deadlock Avoidance

Basic Principle: 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
- 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

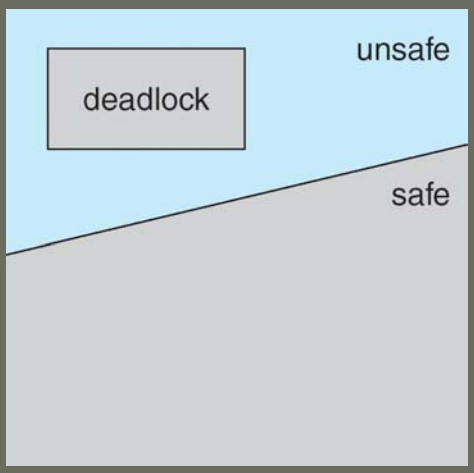
Safe state

- 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 sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes is 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.

Basic Facts

- 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.
  - When a request is done by a process for some resource(s): check before allocating resource(s); if it will leave the system in an unsafe state, then do not allocate the resource(s); process is waited and resources are not allocated to that process.

Safe, Unsafe , Deadlock State



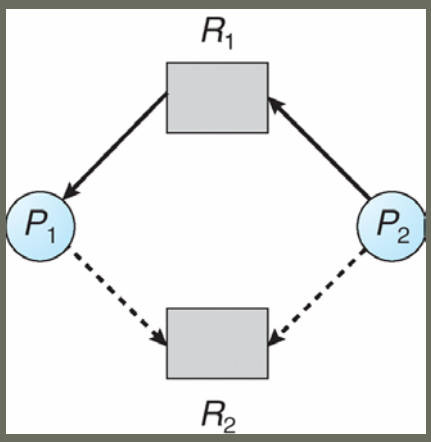
Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

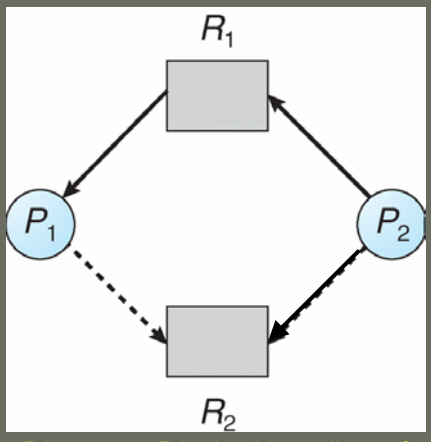
Resource-Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j; represented by a dashed line
- Claim edge is converted to a request edge when a process requests a resource
- Request edge is converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge is reconverted to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph

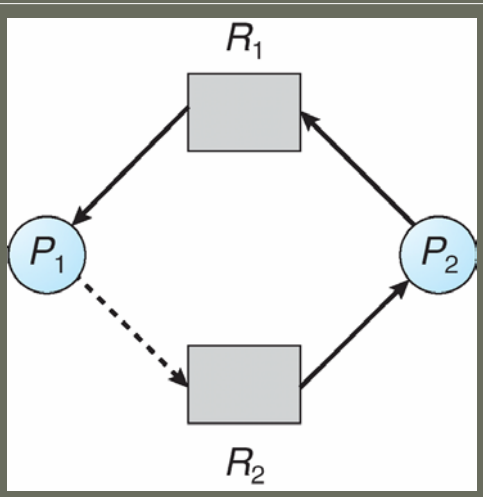


Resource-Allocation Graph



P2 requests R2; should we allocate?

Unsafe State In Resource-Allocation Graph



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

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 at that time t.
- **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

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

An example system state

Existing A B C
10 5 7

$$Available[j] = Existing[j] = \sum_{i=0}^{n-1} Allocation[i, j]$$

state changing over time

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

	Allocation A B C
P0	0 1 0
P1	2 0 0
P2	3 0 2
P3	2 1 1
P4	0 0 2

Need = Max - Allocation

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

Available A B C
3 3 2

Notation

	X A B C
P0	0 1 0
P1	2 0 0
P2	3 0 2
P3	2 1 1
P4	0 0 2

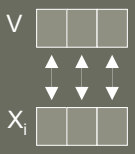
X is a matrix.

X_i is the i^{th} row of the matrix: it is a vector. For example, X_3 = [2 1 1]

V A B C
3 3 2

V is a vector; V = [3 3 2]

Compare two vectors:
Ex: compare V with X_i



V == X_i ?
V <= X_i ?
X_i <= V ?
….

Ex:Compare [3 3 2] with [2 2 1]

[2 2 1] <= [3 3 2]

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
   Work = *Available* (initialize Work temporary vector)
   Finish [*i*] = *false* for *i* = 0, 1, …, *n*-1
   (Work is a temporary vector initialized to the Available (i.e., free) resources at that time when the safety check is performed)

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.

	Allocation A B C
P0	0 1 0
P1	2 0 0
P2	3 0 2
P3	2 1 1
P4	0 0 2

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

Work [3 3 2]

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

Algorithm

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

Resource-Request Algorithm for Process P_i

- 3. Pretend to allocate requested resources to P_i by modifying the state as follows:

  $Available = Available - Request_i;$

  $Allocation_i = Allocation_i + Request_i;$

  $Need_i = Need_i - Request_i;$

  Run the Safety Check 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, B, and C

Existing Resources: *A* (10 instances), *B* (5 instances), and *C* (7 instances)

Existing = [10, 5, 7]
initially, Available = Existing.

Assume, processes indicated their maximum demand as follows:

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

Example of Banker's Algorithm

- Assume later, at an arbitrary time t, we have the following system state:

Existing = [10 5 7]

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

	Allocation A B C
P0	0 1 0
P1	2 0 0
P2	3 0 2
P3	2 1 1
P4	0 0 2

Need = Max - Allocation

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

Available A B C
3 3 2

Is it a safe state?

Example of Banker's Algorithm

	Allocation A B C
P0	0 1 0
P1	2 0 0
P2	3 0 2
P3	2 1 1
P4	0 0 2

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

Available A B C
3 3 2

Try to find a row in $Need_i$ that is <= Available.

P1. run completion. Available becomes = [3 3 2] + [2 0 0] = [5 3 2]
P3. run completion. Available becomes = [5 3 2] + [2 1 1] = [7 4 3]
P4. run completion. Available becomes = [7 4 3] + [0 0 2] = [7 4 5]
P2. run completion. Available becomes = [7 4 5] + [3 0 2] = [10 4 7]
P0. run completion. Available becomes = [10 4 7] + [0 1 0] = [10 5 7]

We found a sequence of execution: P1, P3, P4, P2, P0. State is safe

Example: P_1 requests (1,0,2)

- At that time Available is [3 3 2]
- First check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.
- Then check the new state for safety:

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

	Allocation A B C
P0	0 1 0
P1	3 0 2
P2	3 0 2
P3	2 1 1
P4	0 0 2

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

Available A B C
2 3 0

new state (we did go to that yet; we are just checking)

Example: P_1 requests (1,0,2)

	Allocation A B C
P0	0 1 0
P1	3 0 2
P2	3 0 2
P3	2 1 1
P4	0 0 2

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

Available A B C
2 3 0

new state

Can we find a sequence?
Run P1. Available becomes = [5 3 2]
Run P3. Available becomes = [7 4 3]
Run P4. Available becomes = [7 4 5]
Run P0. Available becomes = [7 5 5]
Run P2. Available becomes = [10 5 7]

Sequence is:
*P*1, *P*3, *P*4, *P*0, *P*2
Yes, New State is safe.
We can grant the request.
Allocate desired resources to process P1.

P_4 requests (3,3,0)?

	Allocation A B C
P0	0 1 0
P1	3 0 2
P2	3 0 2
P3	2 1 1
P4	0 0 2

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

Available A B C
2 3 0

Current state

If this is current state, what happens if P4 requests (3 3 0)?

There is no available resource to satisfy the request. P4 will be waited.

P_0 requests (0,2,0)? Should we grant?

	Allocation A B C
P0	0 1 0
P1	3 0 2
P2	3 0 2
P3	2 1 1
P4	0 0 2

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

Available A B C
2 3 0

Current state

System is in this state.
P0 makes a request: [0, 2, 0]. Should we grant.

P_0 requests (0,2,0)? Should we grant?

Assume we allocate 0,2,0 to P0. The new state will be as follows.

	Allocation A B C
P0	0 3 0
P1	3 0 2
P2	3 0 2
P3	2 1 1
P4	0 0 2

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

Available A B C
2 1 0

New state

Is it safe?

No process has a row in Need matrix that is less than or equal to Available. Therefore, the new state would be UNSAFE. Hence we should not go to the new state. The request is not granted. P0 is waited.

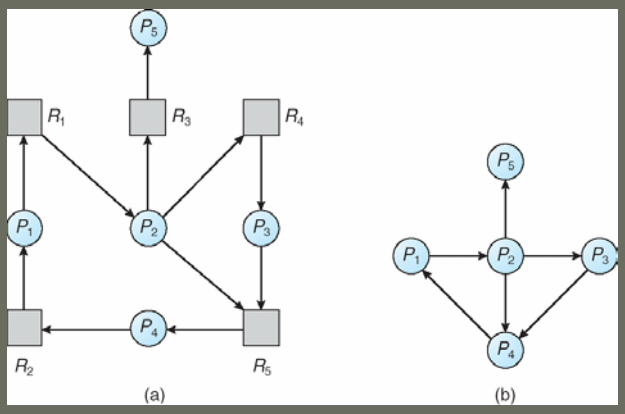
Deadlock Detection

- 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
- 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

Several Instances of a Resource Type



Several Instances of Research 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.

System state is represented by this information

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*,
   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

Detection Algorithm (Cont.)

3. *Work = Work + Allocation*$_i$
   Finish[*i*] = *true*
   go to step 2

4. If *Finish*[*i*] == false, for some *i*, $1 \le i \le n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

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*:

Existing A B C
7 2 6

	Allocation A B C
P0	0 1 0
P1	2 0 0
P2	3 0 3
P3	2 1 1
P4	0 0 2

	Request A B C
P0	0 0 0
P1	2 0 2
P2	0 0 0
P3	1 0 0
P4	0 0 2

Available A B C
0 0 0

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

Example of Detection Algorithm

	Allocation A B C
P0	0 1 0
P1	2 0 0
P2	3 0 3
P3	2 1 1
P4	0 0 2

	Request A B C
P0	0 0 0
P1	2 0 2
P2	0 0 0
P3	1 0 0
P4	0 0 2

Available A B C
0 0 0

Can we find a row i in Request that can be satisfied with Available, i.e. $Request_i$ <= Available?

P0 is not deadlocked at the moment. Run completion. Available becomes: [0 1 0]
Then P2 can be satisfied. Can run completion. Available becomes: [3 1 3]
Then P3 can be satisfied. Can run completion. Available becomes: [5 2 4]
Then P1 can be satisfied. Can run completion. Available becomes: [7 2 4]
Then P4 can be satisfied. Can run completion. Available becomes: [7 2 6]

Another example

- Lets assume at time t2, P_2 makes a request for an additional instance of type C. Then **Request matrix** becomes

	Request A B C
P0	0 0 0
P1	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

State of system? Check it.

Check

	Allocation A B C
P0	0 1 0
P1	2 0 0
P2	3 0 3
P3	2 1 1
P4	0 0 2

	Request A B C
P0	0 0 0
P1	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

Available A B C
0 0 0

We can run P0. Then Available becomes: [0 1 0]
Now, we can not find a row of Request that can be satisfied.
Hence all processes P1, P2, P3, and P4 has to wait in the request. We have a deadlock.

Processes P1, P2, P3, and P4 are deadlocks.

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

End of lecture