Deadlocks

Deadlock problem. Necessary conditions for deadlock. Resource allocation graphs. Java deadlock examples. Deadlock prevention. Deadlock avoidance (safe state and Banker’s algorithm). Deadlock detection and recovery.

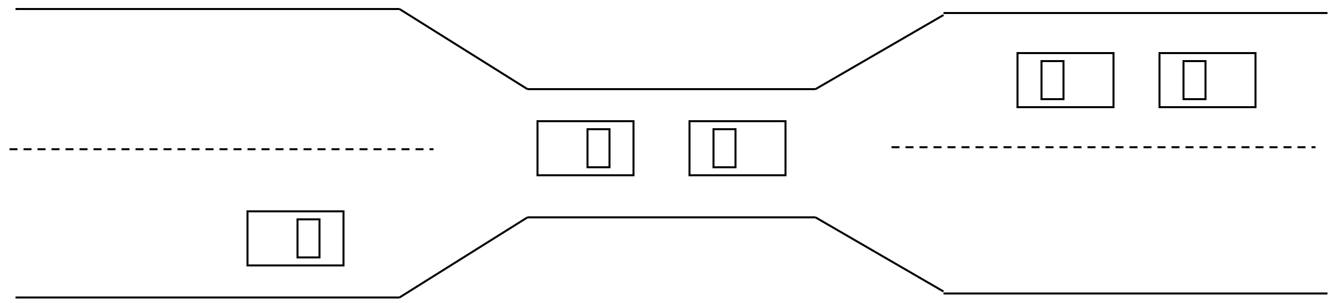
OS5: 22/2/2018
Textbook (SGG): Ch. 7.1-7.4, 7.5.1, 7.5.3, 7.6.2, 7.7

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 (*circular wait*: simplest process P waiting for Q; Q waiting for P)
- Example 1
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs the other one
- Example 2: (binary) semaphores *A* and *B*, initialized to 1

```
    P0                P1
acquire(A);       acquire(B);
acquire(B);       acquire(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 back up in general
- Starvation is possible (cars in one direction only keep going)
- Note: most operating systems do not prevent or resolve deadlock completely (users will deal with it when needed)

General System Model

- Resource types R_1, R_2, . . ., R_m
  - Physical interpretation of resource types: CPU, memory, I/O devices, etc
- OS has W_i instances (or units) of each resource type R_i that it can allocate to requesting processes
- Each process utilizes a resource as follows:
  - **request** (request one instance of a resource from OS; e.g., get access to a printer)
  - **use** (use the acquired resource privately to do its work; e.g., print a file to the acquired printer)
  - **release** (return an acquired instance of a resource back to the OS; e.g., give back the acquired printer so the OS can give this printer to another process)

Necessary Conditions for Deadlock

Deadlock *can* occur (but may not necessarily occur) if these 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 process has completed its task (hence use of the resource)
- **Circular wait:** There exists a set $\{P_0, P_1, \ldots, P_n\}$ 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

NB: These conditions are *necessary*, but *not sufficient*, for deadlock
Implication: In principle, we can solve deadlock by removing *any* of these conditions

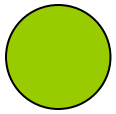
Resource Allocation Graph

A *directed graph*, with set of vertices/nodes V and set of edges E

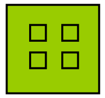
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set of all the resource types in the system
- **request edge**: directed edge $P_i \rightarrow R_j$
  - P_i wants to acquire an instance of R_j
- **assignment edge:** directed edge $R_j \rightarrow P_i$
  - An instance of R_j is being held by P_i

Resource-Allocation Graph (cont'd)

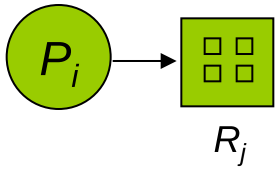
- Process (first type of nodes/vertices)



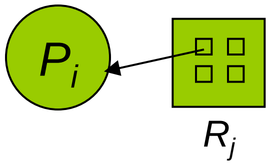
- Resource type with 4 instances (second type of nodes)



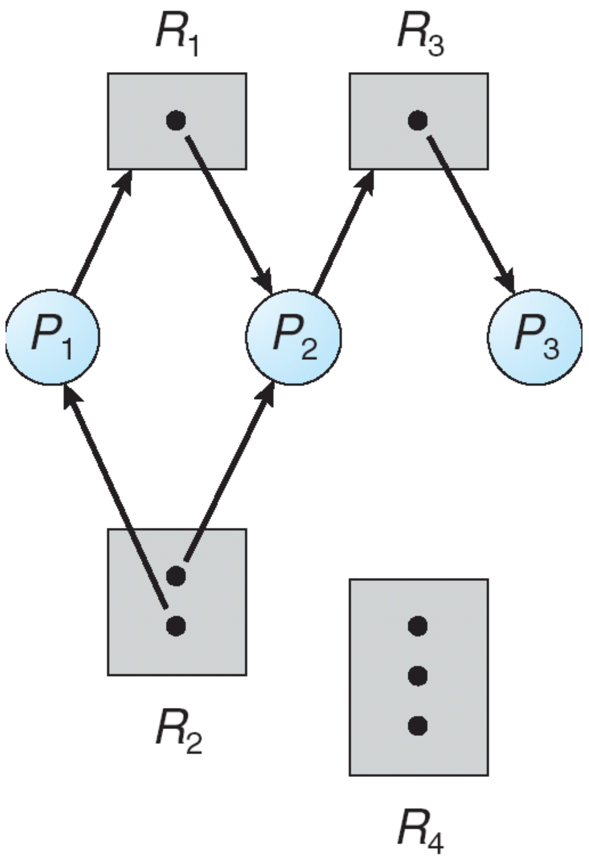
- P_i requests an instance of R_j (first type of directed edge)



- P_i is holding an instance of R_j (second type of directed edge)



Example Resource Allocation Graph

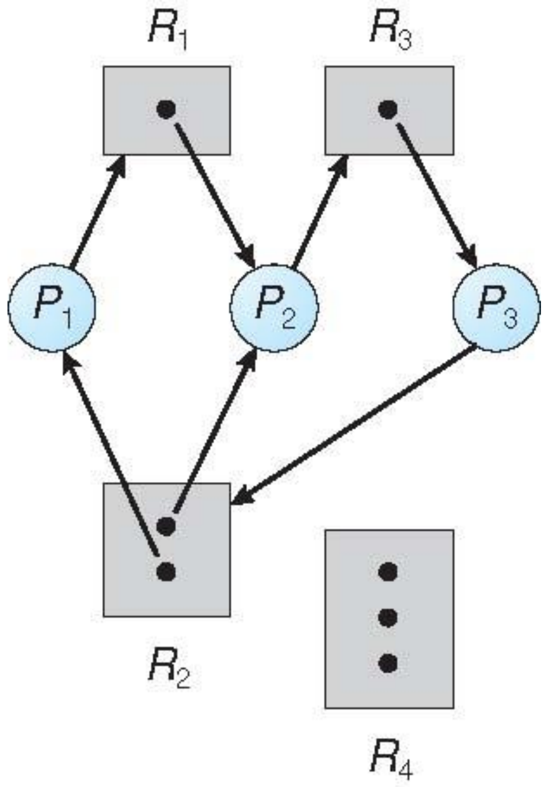


Is this system deadlocked? *Hint*: Is there circular wait?

Resource Allocation Graph with a Deadlock

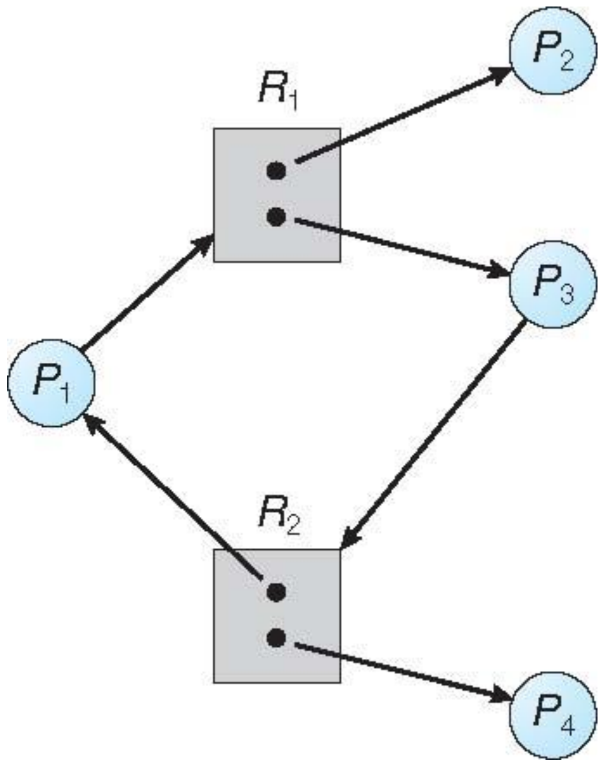
P1 holds an instance of R2, needs an instance of R1 to continue execution so that it may complete its task later

This graph *has* a cycle



Can any of P1, P2, P3 acquire the resource it wants to complete its task?
Hint: The processes won’t release the resources they are holding until they can continue execution and complete their respective tasks.

Graph with A Cycle but No Deadlock



Multiple instances of resource: Cycle is *necessary*, but *not sufficient*, condition for deadlock
This graph has a cycle. But what is a possible completion sequence of these processes?

Summary: Graph Cycles and Deadlock

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

Java Deadlock Example

```
class A implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         first.lock();
         // do something
            second.lock();
            // do something else
      }
      finally {
         first.unlock();
         second.unlock();
      }
   }
}
```

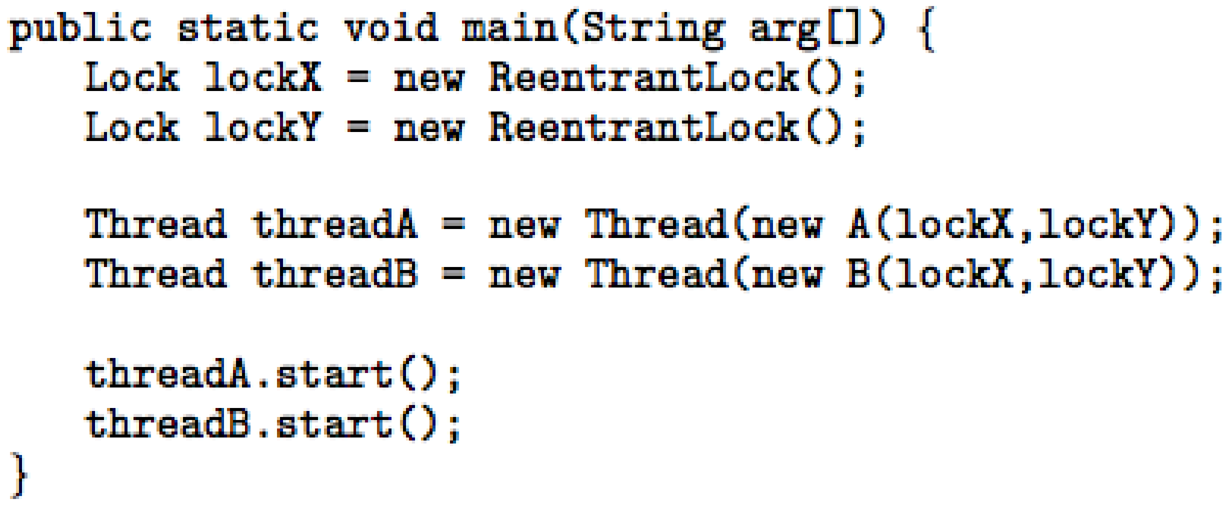
Thread A

```
class B implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         second.lock();
         // do something
            first.lock();
            // do something else
      }
      finally {
         second.unlock();
         first.unlock();
      }
   }
}
```

Thread B

Note: each runnable object needs to get *two* locks to do its work

Java Deadlock Example



```
public static void main(String arg[]) {
    Lock lockX = new ReentrantLock();
    Lock lockY = new ReentrantLock();

    Thread threadA = new Thread(new A(lockX,lockY));
    Thread threadB = new Thread(new B(lockX,lockY));

    threadA.start();
    threadB.start();
}
```

lockX and **lockY** are binary locks, initially available. How can deadlock occur? (Demonstrate an interleaving of the two threads' execution.)

Hint: Each runnable object in previous slide runs as a separate thread; These threads acquire the locks **lockX** and **lockY** in *different* orders

Methods for Handling Deadlocks

- Deadlock *avoidance*
  - Before granting a resource request (even if request is valid and the requested resources are now available), check that the request will not cause the system to enter a deadlock state (not just no deadlock immediately, but *not even later*)
  - Requires advance knowledge of *future* resource needs (e.g., Banker's algorithm in your OS Lab #3)
- Deadlock *detection and recovery*
  - Detect deadlock after the fact, then recover from it (e.g., preempting held resources and rolling back processes)
- Deadlock *prevention*
  - Impose conditions on resource requests to ensure that a valid request can *never* cause the system to enter a deadlock state *by design* (so no need to check the runtime system state as in deadlock avoidance)
- Real-world OS (e.g., UNIX) may not handle deadlocks completely (i.e., possible that users will see a deadlock; then they'll deal with it)

Deadlock Avoidance

Requests can lead to deadlock (in the future); don’t grant those requests (now) even if the requested resources are currently available

- Need additional a priori information: Simplest and most useful model requires each process to declare (in advance) the *maximum number* of resources of each type that it will ever need.
- At time of resource request, avoidance algorithm examines the system’s resource allocation state to ensure that granting the request will *never* lead to a *circular-wait* condition later.
- Resource allocation state is defined by the numbers of currently available and allocated resources, and the maximum number of resources that each process may need in the future.

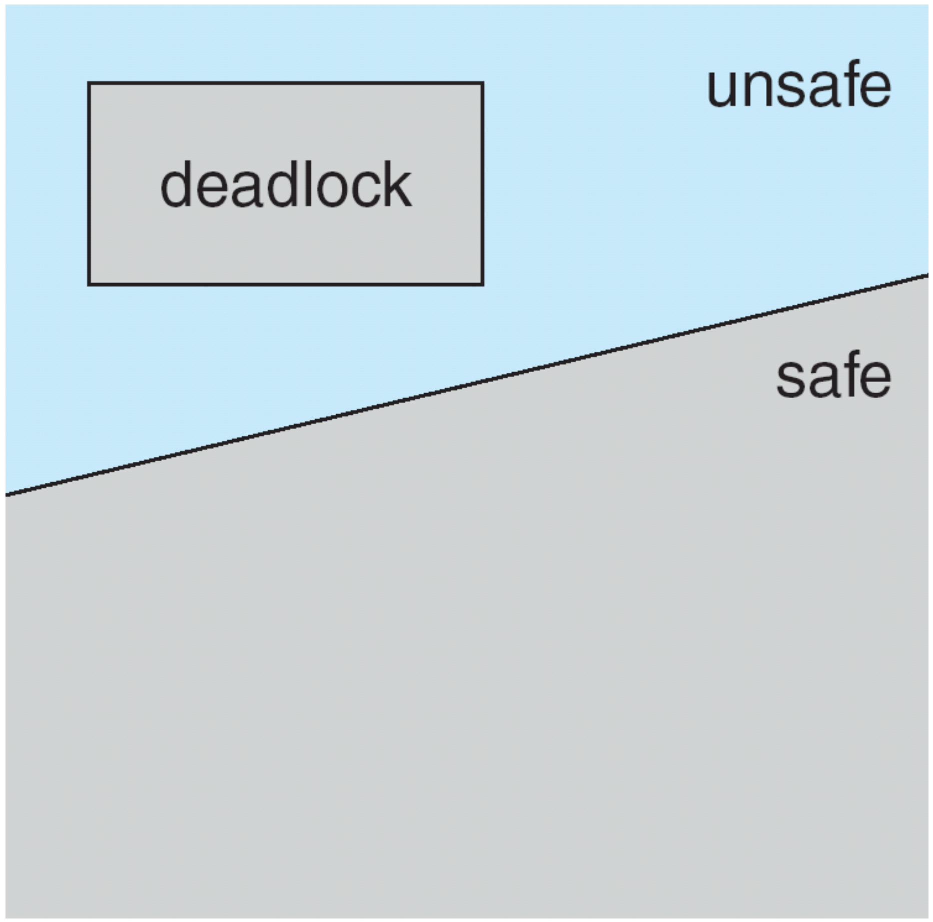
Safe State

- When a process requests an available resource, system must decide if granting the request will leave the system in a safe state.
- System is in **safe state** if there exists a sequence <P_1, P_2, …, P_n> of all the processes in the system such that for each P_i, the resources that P_i will ever need can be satisfied by the currently available resources *plus* the resources held by all the preceding P_j, with $j < i$.
- Rationale:
  - If P_i's resource needs are not immediately available, then P_i can wait until all the preceding P_j processes have finished.
  - When the preceding processes all finished, P_i can obtain its needed resources, do its job, *return its allocated resources*, and finish.
  - When P_i finishes, P_{i+1} can obtain its needed resources and finish, and so on.
  - I.e., <P_1, P_2, …, P_n> defines a feasible order for all the processes to finish.

Basic Facts

- If system is in safe state $\Rightarrow$ no deadlocks
- If system is in unsafe state $\Rightarrow$ *possibility* of deadlock
- Avoidance $\Rightarrow$ ensure that system will never enter an unsafe state.
  - So we'll grant a resource request only if *after* granting the request, the system will still be in a safe state.
  - We need an algorithm to decide if a system is in a safe state (Slide 7.22)

Safe, Unsafe, Deadlock States



Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
  - We won’t cover this algorithm (limited applicability)
- Multiple instances of a resource type
  - Use the Banker’s algorithm (your Lab #3)
  - Subsumes the single-instance problem above

Banker’s Algorithm

- Multiple instances of each resource
- Each process P must *a priori* declare its maximum needs (i.e., maximum number of instances of each resource type that P will ever need)
- When a process requests a resource, it may have to wait
- After a process P got all its resources, it must return them within a finite amount of time (when P finishes its task)

Data Structures for Banker's Algorithm

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

- **Available**: Vector (i.e., 1D array) of length m. If $Available[j] = k$, there are k instances of resource type R_j available

- **Max**: n x m matrix (i.e., 2D array). 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   *// NB: Work, Available are both arrays*

   Finish[*i*] = *false* for *i* = 0, 1, …, *n* - 1

2. Find an *i* such that:

   (a) *Finish*[*i*] = *false*

   (b) *Need*[*i*] ≤ *Work*

   If no such *i* exists, go to Step 4

3. *Work* = *Work* + *Allocation*[*i*]   *// Allocation*[*i*] *is i-th row of Allocation matrix*

   Finish[*i*] = *true*

   go to Step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state; otherwise it's unsafe

Algorithm for Granting Request by Process P_i

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

1. If *Request*[*i*] ≤ *Need*[*i*] go to Step 2. Otherwise, raise error, since process has exceeded its maximum claim.

2. If *Request*[*i*] ≤ *Available*, go to Step 3. Otherwise P_i must wait, since the resources are not immediately available.

3. Try to allocate the requested resources to P_i by updating the resource allocation state as follows (i.e., assume we grant the new request):

   Available = *Available* - *Request*[*i*];

   $Allocation_i$ = *Allocation*[*i*] + *Request*[*i*];

   Need[*i*] = *Need*[*i*] - *Request*[*i*];

   - If *new* state is *safe* ⇒ the resources are allocated to P_i
   - If *new* state is *unsafe* ⇒ P_i must wait, and the old resource-allocation state is restored (i.e., new request not granted after all)

Activity 5.1: 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	

- Is the system safe? If so, give an execution sequence of the processes that demonstrates the safety. If not, why not?

Activity 5.1 (cont'd)

- Given the original system state in the previous slide, P_1 now requests (1,0,2)
- First, check that Request $\leq$ Available, i.e., $(1,0,2) \leq (3,3,2) \Rightarrow$ true
- Second, now assume we grant this request, which will update the data structures as follows:

	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 the safety condition (i.e., system will remain safe after granting (1,0,2) to P_1)
- Can we now *further* grant (i) request (3,3,0) by $P_{4;}$ *or* (ii) request (0,2,0) by P_0?

Deadlock Detection

- Allow system to enter deadlock state
- Detect occurrence of deadlock by *detection algorithm*
- Recover from the detected deadlock

Deadlock Detection: Multiple Instances

- **Available**: A vector (i.e., 1D array) of length *m* indicates the number of available resources of each type.
- **Allocation**: An *n* x *m* matrix (i.e., 2D array) 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.

NB: The deadlock *detection* algorithm (next slide) detects if the system is *already* deadlocked. It's similar to the safety algorithm (Slide 7.22) used by the Banker's algorithm for deadlock avoidance, but the safety algorithm detects if the system *can* become deadlocked *in the future*.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

   For *i* = 1,2, …, *n*, if *Allocation*[i] ≠ 0, then
   Finish[i] = false; else *Finish*[i] = *true*

2. Find an index *i* such that both:

   (a) *Finish*[*i*] == *false*

   (b)*Request*[*i*] ≤ *Work*. *// What i requests now is available if all its preceding processes*
   // finish; i isn't deadlocked already

   If no such *i* exists, go to Step 4

3. *Work* = *Work* + *Allocation*[*i*]. *// Assume i will finish and return the resources it holds now*
   Finish[*i*] = *true*
   go to Step 2

4. If *Finish*[*i*] == *false* for some *i*, then the system is (already) in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is (already) deadlocked.

What is the complexity (in terms of *m* and *n*) of the algorithm?

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*

Detection Example (cont'd)

- Given the original system state in the previous slide, P_2 now requests an additional instance of type *C*, so that we have these requests:

	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

- What's the new state of the system?
  - Can assume that P_0 will return the resources it holds, but there'll still be insufficient resources to satisfy any other processes' requests
  - Deadlock already exists, consisting of processes P_1, P_2, P_3, and P_4

Deadlock Recovery: Abort Processes

- Abort all deadlocked processes (resources held are preempted)
- Abort one process at a time until the deadlock cycle is eliminated
- Which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer until completion?
  - Resources the process has used,
  - Resources the process needs to complete.
  - How many processes will system need to abort?
  - Is process interactive or batch?
  - etc …

Deadlock Prevention

Constrain the ways requests can be made, in order to disallow “hold-and-wait,” *or* “no preemption,” *or* ”circular wait” (in prevention, valid requests won’t cause deadlock *by design*, no need to check for safety of runtime system state)

- **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 (i.e., previously held resources must be released before new resources are requested)
  - OS must support a new system call for acquiring multiple resources *at the same time*
    - E.g., for semaphores, define a new “acquireAll([list of semaphores])” system call for acquiring all the semaphores in the argument list, in addition to normal acquire() system call
  - Disadvantages
    - Low resource utilization if a process has to acquire in the beginning all the resources it’ll ever need – why?
    - Starvation becomes more likely – Process P needs two resources R1 and R2; R1 is available from time to time, similarly R2; but R1 and R2 are never available *at the same time*

Deadlock Prevention (cont'd)

- **No Preemption** – if a process P (holding some resources already) requests another resource that cannot be immediately granted, then OS will force P to release all the resources it already holds
  - Preempted resources are added to the list of resources for which the process needs to wait for again
  - Process will be restarted only when it can obtain all the resources it needs (both the old ones preempted and the new one requested)
  - Disadvantages: preemption costs and starvation
- **Circular Wait** – impose a total ordering of all the resource types, and require that each process requests resources according to that order
  - Say: give an id for each resource type (e.g., 1 for disk, 2 for printer, etc); process can only request resource types in *increasing* id number (i.e., can request printer while holding disk, but *not* vice versa)
  - Disadvantage: Burden on programmer to ensure the order by design, without unnecessarily sacrificing utilization