CSE 421/521 - Operating Systems
Spring 2018

Lecture - XI
Deadlocks - II

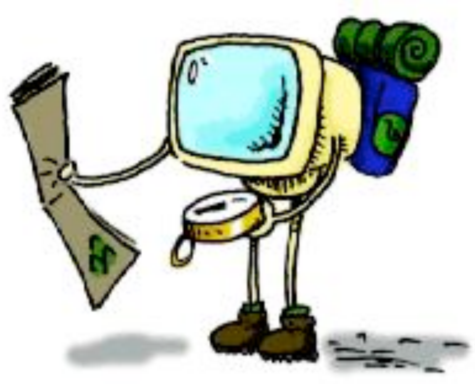
Tevfik Koşar

University at Buffalo

March 6th, 2018

Roadmap

- Deadlocks
  - Deadlock Prevention
  - Deadlock Detection
  - Deadlock Recovery
  - Deadlock Avoidance



Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
  - ➔deadlock prevention or avoidance
- Allow the system to enter a deadlock state and then recover.
  - ➔deadlock detection and recovery
- Ignore the problem and pretend that deadlocks never occur in the system
  - ➔ Programmers should handle deadlocks (UNIX, Windows)

Remember

In the code below, three processes are competing for six resources labeled A to F.

a. Using a resource allocation graph (Silberschatz pp.249-251) show the possiblity of a deadlock in this implementation.

b. Modify the order of some of the `get` requests to prevent the possiblity of any deadlock. You cannot move requests across procedures, only change the order inside each procedure. Use a resource allocation graph to justify your answer.

```
void P0()
{
  while (true) {
    get(A);
    get(B);
    get(C);
    // critical region:
    // use A, B, C
    release(A);
    release(B);
    release(C);
  }
}
```

```
void P1()
{
  while (true) {
    get(D);
    get(E);
    get(B);
    // critical region:
    // use D, E, B
    release(D);
    release(E);
    release(B);
  }
}
```

```
void P2()
{
  while (true) {
    get(C);
    get(F);
    get(D);
    // critical region:
    // use C, F, D
    release(C);
    release(F);
    release(D);
  }
}
```

Deadlock Prevention

➔ Ensure one of the deadlock conditions cannot hold

➔ Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
  - Eg. read-only files

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  1. Require process to request and be allocated all its resources before it begins execution
  2. or allow process to request resources only when the process has none.

  Example: Read from DVD to memory, then print.
  1. holds printer unnecessarily for the entire execution
     - Low resource utilization
  2. may never get the printer later
     - 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.

Exercise - (cont.)

In the code below, three processes are competing for six resources labeled A to F.

a. Using a resource allocation graph (Silberschatz pp.249-251) show the possiblity of a deadlock in this implementation.

b. Modify the order of some of the `get` requests to prevent the possiblity of any deadlock. You cannot move requests across procedures, only change the order inside each procedure. Use a resource allocation graph to justify your answer.

```
void P0()
{
  while (true) {
    get(A);
    get(B);
    get(C);
    // critical region:
    // use A, B, C
    release(A);
    release(B);
    release(C);
  }
}
```

```
void P1()
{
  while (true) {
    get(D);
    get(E);
    get(B);
    // critical region:
    // use D, E, B
    release(D);
    release(E);
    release(B);
  }
}
```

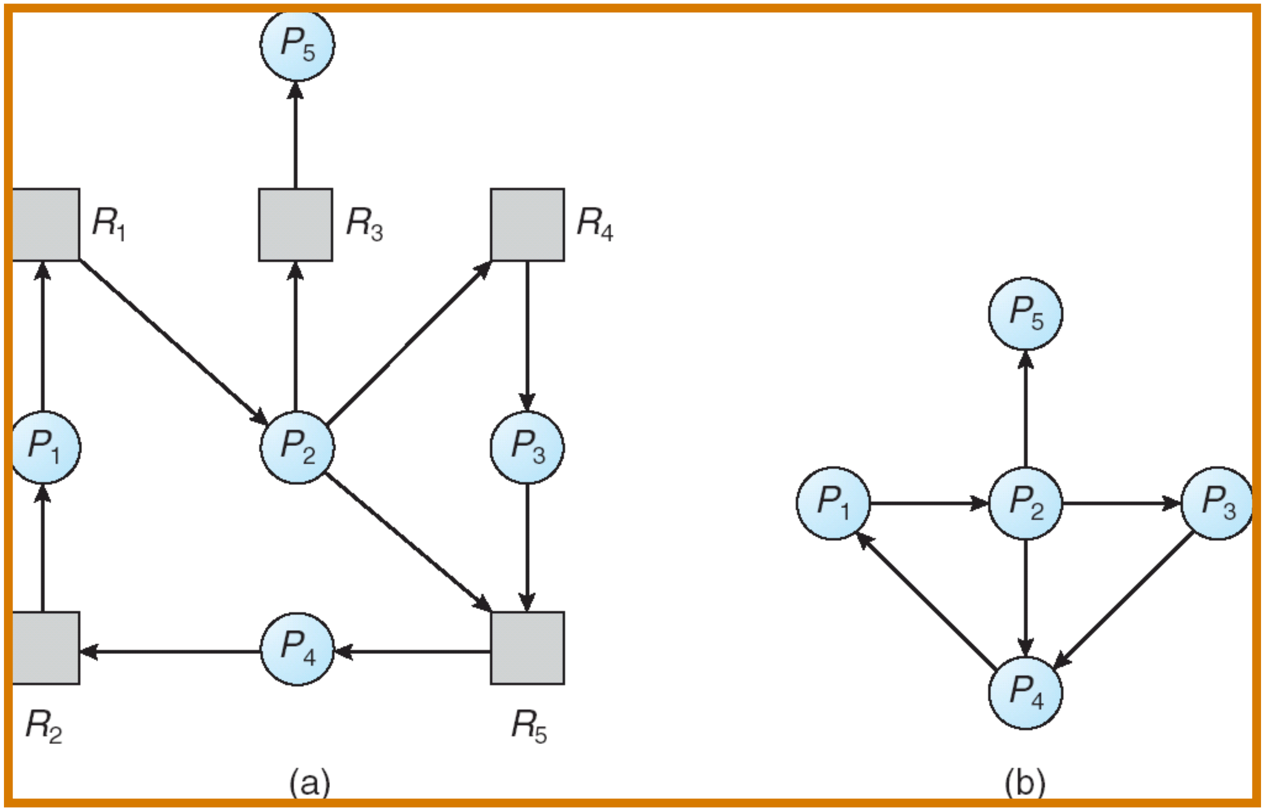
```
void P2()
{
  while (true) {
    get(C);
    get(F);
    get(D);
    // critical region:
    // use C, F, D
    release(C);
    release(F);
    release(D);
  }
}
```

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.



Resource-Allocation Graph   Corresponding wait-for graph

Single Instance of Each Resource Type

- 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.
- Only good for single-instance resource allocation systems.

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* = 0,1, 2, …, *n-1*, *Finish*[i] = false.

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, $0 \leq i \leq n-1$, 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*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	0 0 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.

	Allocation	*Request*	*Available*	*Work*
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$
P_0	0 1 0	0 0 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2		
P_2	3 0 3	0 0 **1**		
P_3	2 1 1	1 0 0		
P_4	0 0 2	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.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes. --> expensive

- Abort one process at a time until the deadlock cycle is eliminated. --> overhead of deadlock detection alg.

- 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.

Deadlock Avoidance

Deadlock Prevention: prevent deadlocks by restraining resources and making sure one of 4 necessary conditions for a deadlock does not hold. (system design)

--> possible side effect: low device utilization and reduced system throughput

Deadlock Avoidance: Requires that the system has some additional *a priori* information available. (dynamic request check)

i.e. request disk and then printer..

or request at most n resources

--> allows more concurrency

- **Similar to the difference between a traffic light and a police officer directing the traffic!**

Deadlock Avoidance

- 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

- A state is **safe** if the system can allocate resources to each process (upto its maximum) in some order and can still avoid a deadlock.
- 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.

Safe State

- Sequence <P_1, P_2, …, 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 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.
- If no such sequence exists, the state is **unsafe!**

Example of Safe State

- 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:

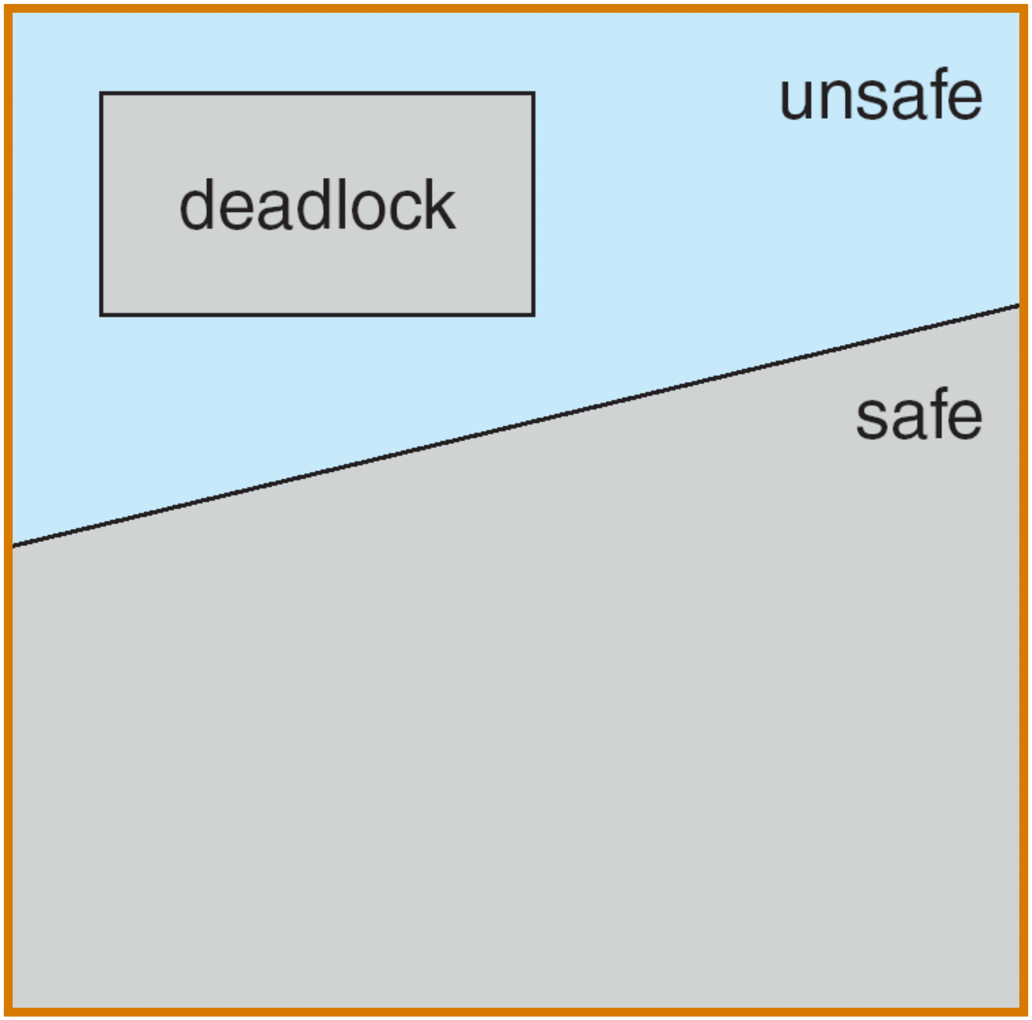
	Allocation	*Request*	*Available*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2		
P_2	3 0 3	0 0 0		
P_3	2 1 1	1 0 0		
P_4	0 0 2	0 0 2		

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ represents 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, Unsafe , Deadlock State



Example

Consider a system with 3 processes and 12 disks.

At t = t0;

	Maximum Needs	Current Allocation
P1	10	5
P2	4	2
P3	9	2

Example *(cont.)*

Consider a system with 3 processes and 12 disks.

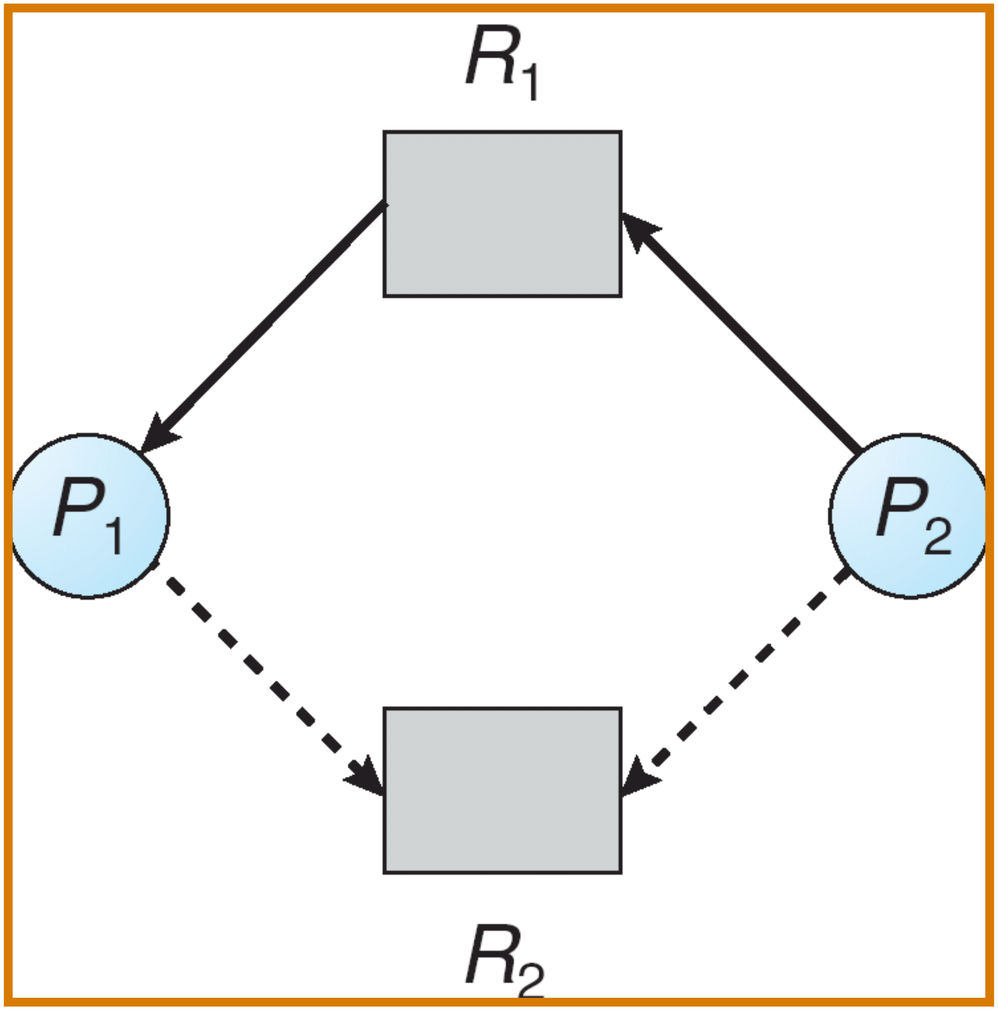
At t = t1;

	Maximum Needs	Current Allocation
P1	10	5
P2	4	2
P3	9	3

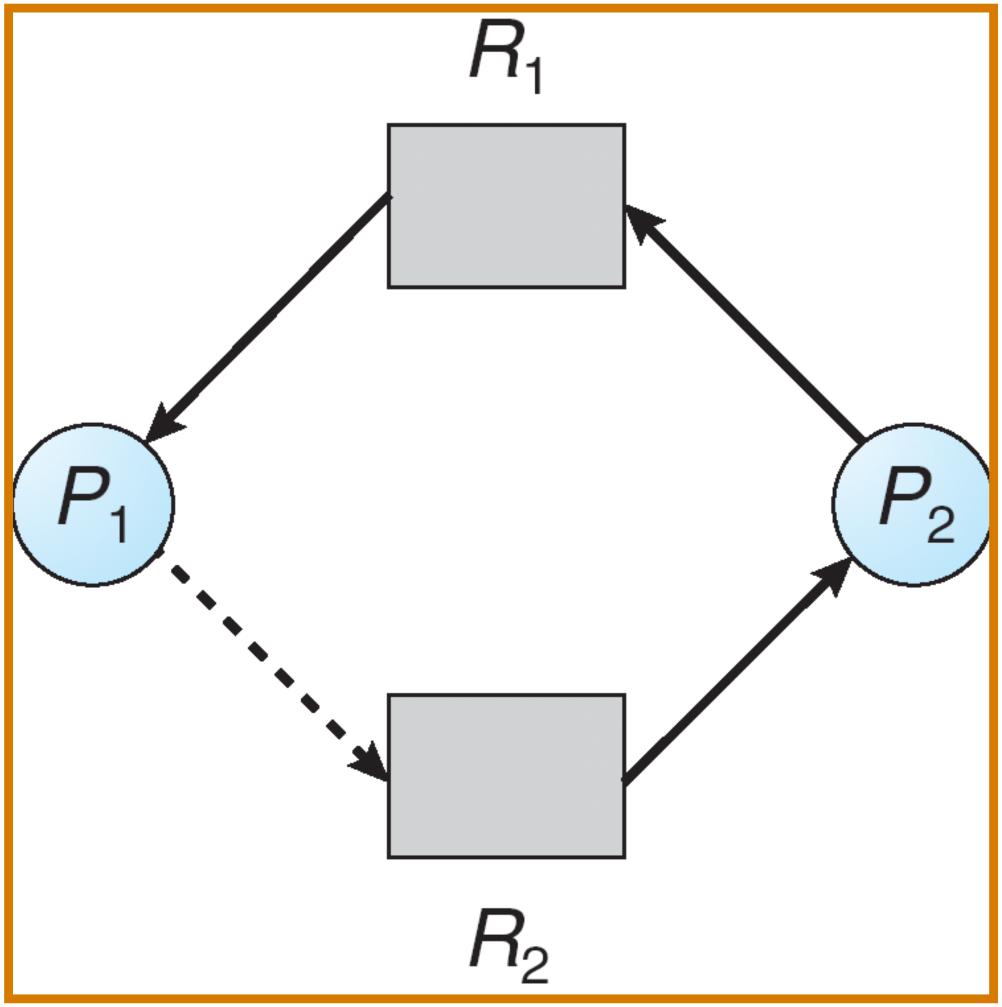
Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by 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.
- If a new allocation results in a cycle in the graph (unsafe state), we cannot allow that allocation.
- Resources must be claimed *a priori* in the system.

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



Banker’s Algorithm

- Works for multiple resource instances.

- Each process declares maximum # of resources it may need.

- When a process requests a resource, it may have to wait if this leads to an unsafe state.

- 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.
- *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* = 1,2, …, *n*.

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

Let Request$_i$ be the 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:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- *If safe* $\Rightarrow$ *the resources are allocated to Pi.*
- *If unsafe* $\Rightarrow$ *Pi 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	

Example of Banker's Algorithm

- 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

Example of Banker's Algorithm

- Snapshot at time T_0:

	Allocation	Max	Available	*Need*
	A B C	A B C	A B C	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Example of Banker's Algorithm

- Snapshot at time T_0:

	Allocation	Max	Available	*Need*
	A B C	A B C	A B C	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	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)

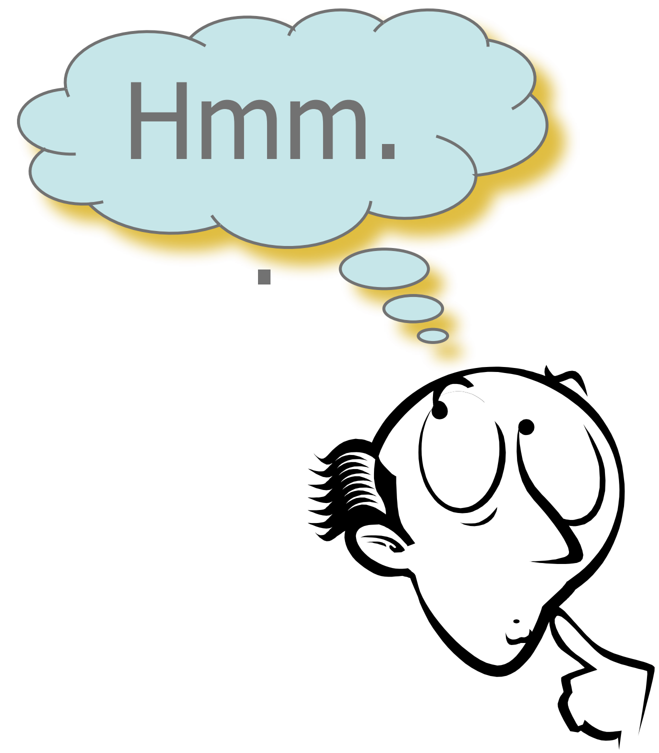
- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.

	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?

Summary

- Deadlocks
  - Deadlock Prevention
  - Deadlock Detection
  - Deadlock Recovery
  - Deadlock Avoidance



Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings
- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from UNR