Part 5: Deadlocks and Starvation

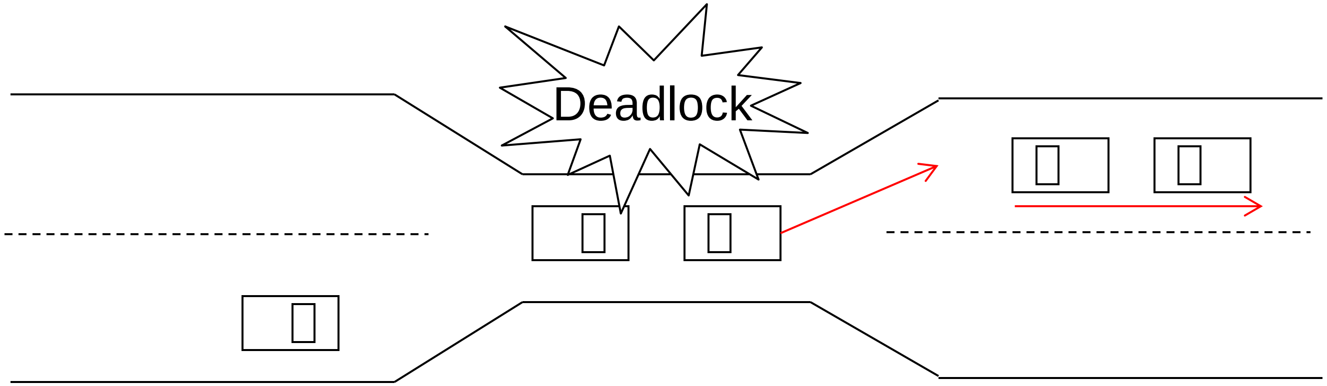
- Deadlock Problem
- System Model
- Deadlock Conditions
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection

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 different type of tape drives; both P1 and P2 need the two tapes to finish execution.
  - P_1 and P_2 each hold one tape 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



- Allow only one car on the bridge.
- The bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (release resource, i.e. the bridge, and rollback).
- Several cars may have to be backed up if a deadlock occurs.

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  e.g. memory space, I/O devices

- Each resource type R_i has W_i identical instances.

- Each process utilizes a resource as follows:
  - request
  - use
  - release

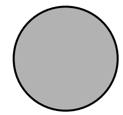
Resource-Allocation Graph

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

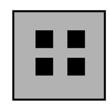
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge: directed edge $P_\mathrm{i} \rightarrow R_j$
  - *When the request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

Resource-Allocation Graph (Cont.)

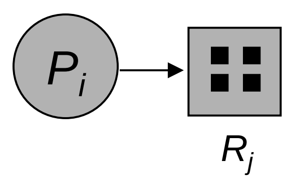
- Process



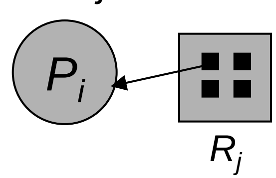
- Resource Type with 4 instances



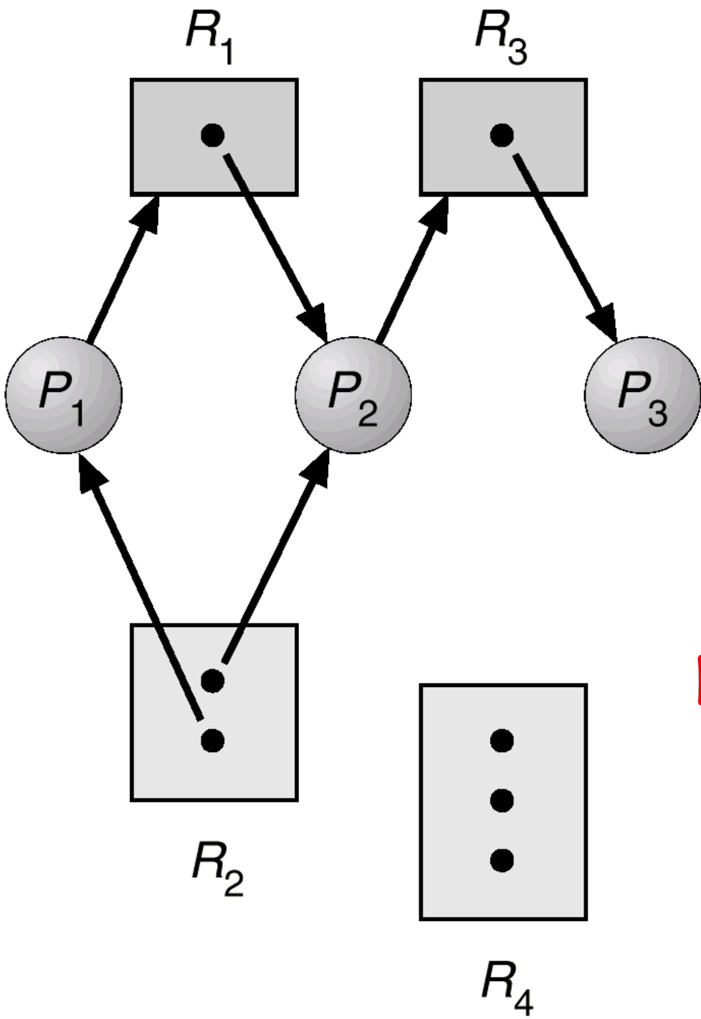
- P_i requests instance of R_j



- P_i is holding an instance of R_j



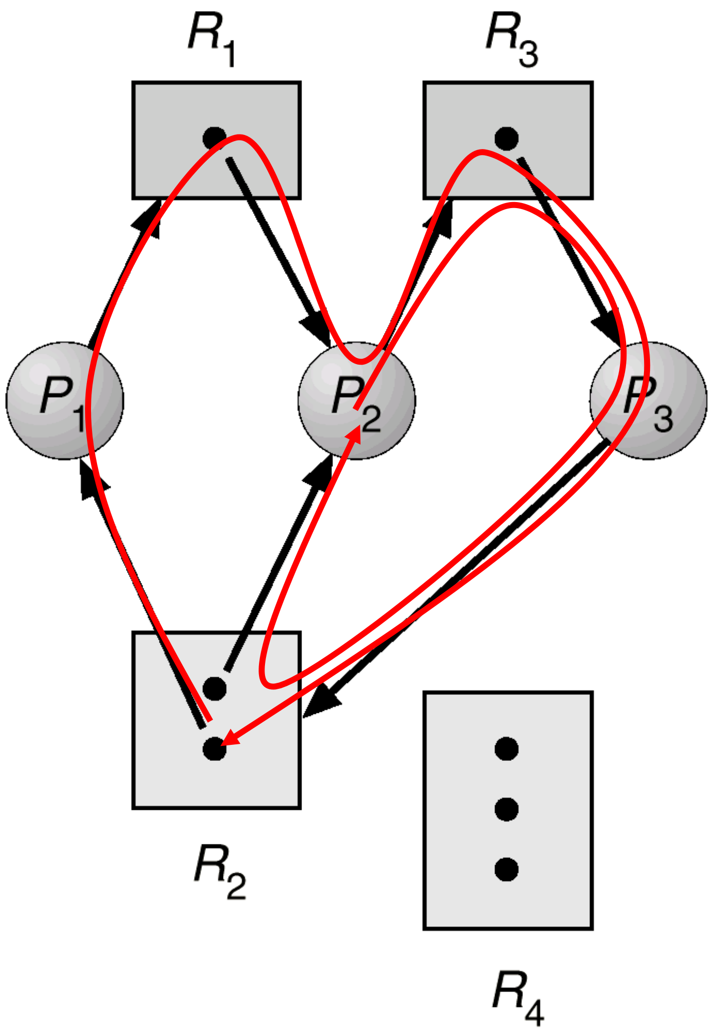
Example of a Resource Allocation Graph



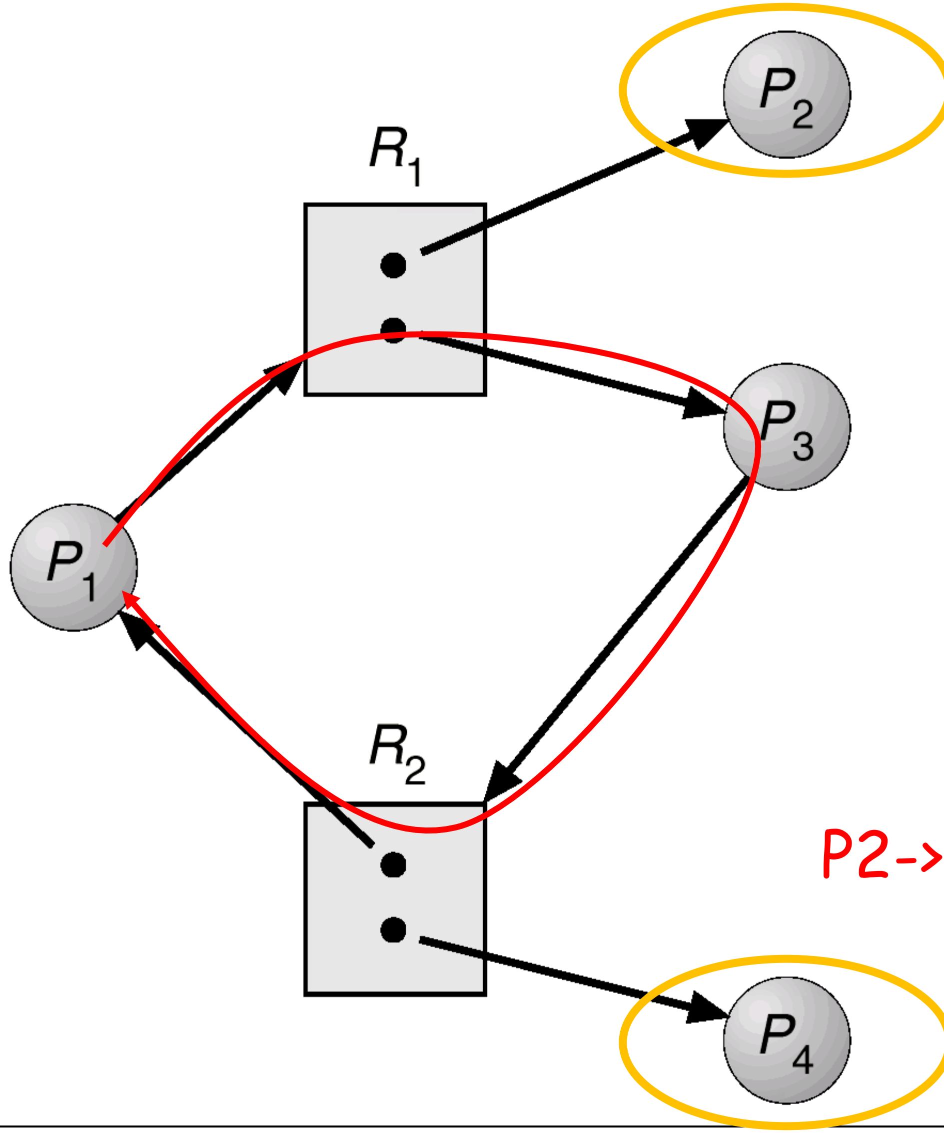
No deadlock.

P3->P2->P1

Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



P2->P4->P3->P1

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 Conditions

Deadlock **can** arise if 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.

Deadlock Conditions (Cont.)

- **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 $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for P_2, …, P_{n-1} is waiting for P_n, and P_n is waiting for P_0.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

Prevent at least one of the deadlock conditions to happen.

- Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources.
  - Sharable resources: code section, read-only data areas.
  - Nonshareble resources: printers, data areas to be written, <u>chopstick in Dining-Philosophers.</u>
- Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - In Dining-Philosophers, allow a philosopher to pick up his chopsticks only if both chopsticks are available.

Deadlock Prevention

Prevent at least one of the deadlock conditions to happen.

- 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. The process restarts later.
  - In Dining-Philosophers, if a philosopher cannot get another chopstick for a long time, preempt.
- Circular Wait –
  - In Dining-Philosophers, allow at most **four** philosophers to be hungry simultaneously.
  - Impose a total ordering of all resource types (see backup slides).

Deadlock Avoidance

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that the system never goes into **unsafe** state.
  - When a process requests an available resource, system must decide if immediate allocation leaves the system in a **safe** state.
  - If safe, the request is granted. Otherwise, the process must wait.
- System is in safe state if there exists a *safe sequence* of all processes.

Safe State (Cont.)

- Sequence <P_1, P_2, …, P_n> is **safe** if for each P_i, the resources that P_i requests 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. <u>→All processes in the sequence can finish.</u>

Example of Safe Sequence

Available: 1

Processes	Hold	Request
P1	1	1
P2	1	2
P3	1	3

Q1: <P1, P2, P3> safe?

Yes.

P1.request<=Available

P2.request<=P1.Hold+Available

P3.request<=P1.Hold+P2.Hold+Available

Q2: <P3, P2, P1> safe?

No.

P3.request>Available.

Q3: Is the system safe?

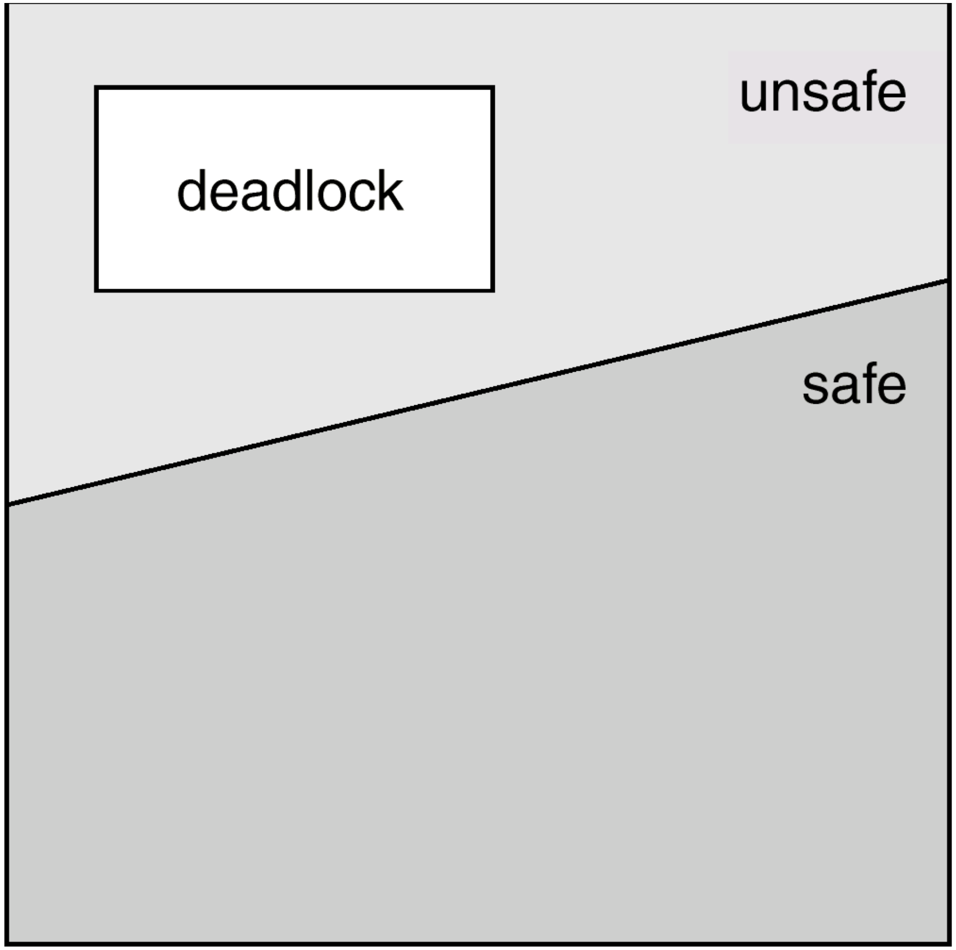
Yes.

<P1, P2, P3> is the safe seq.

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
  - Consider the process can release resources before termination.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Safe, unsafe, deadlock state spaces



Banker's Algorithm

- An algorithm decides whether to allocate the resource and to avoid the unsafe state.
- For multiple instances of each resource type.
- Each process must declare max. # of instances of each resource type that it needs.
- When a process requests a resource it may have to wait (<u>if the allocation leads to unsafe state</u>).
- 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:

   What if multiple answers?

   (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.

   Work= Available + resources held by all the Pj (Finish[j]=true).

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state. Otherwise, the system is unsafe.

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 (Cont.)

- Then the matrix Need is defined to be Max – Allocation.

	Allocation	*Max*	***Need***	*Available*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
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 (Cont.)

- The working is as follows:

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

	Work A	B	C	Finished Process
	3	3	2	
+	2	0	0	P_1
	5	3	2	
+	2	1	1	P_3
	7	4	3	
+	0	0	2	P_4
	7	4	5	
+	3	0	2	P_2
	10	4	7	
+	0	1	0	P_0
	10	5	7	

Example of Banker's Algorithm (Cont.)

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.
- There are multiple safe sequences for that system state.

Resource-Request Algorithm for Process P_i

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

V1<=V2 <=> V1[j]<=V2[j], for all j

The i^{th} row of the matrix.

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 (Cont.)

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

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i.$$

4. Then run the safety algorithm, if the result is:
   - safe $\Rightarrow$ the resources are allocated to P_i.
   - unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm (Cont.)

- Suppose P_1 requests (1, 0, 2) resources.
- Step1: Check $\text{Request}_1 \leq \text{Need}_1$, $(1,0,2) \leq (1,2,2) \Rightarrow$ *true*
- Step2: Check that $\text{Request}_1 \leq$ Available, i.e. $(1,0,2) \leq (3,3,2) \Rightarrow$ *true.*
- Steps 3 & 4:

	Allocation	*Max*	*Need*	*Available*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	~~3 3 2~~ **2 3 0**
P_1	~~2 0 0~~ **3 0 2**	3 2 2	~~1 2 2~~ **0 2 0**	
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	

Pretend to allocate requested resources

Example of Banker's Algorithm (Cont.)

Executing Safety Algorithm

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	~~3 3 2~~ **2 3 0**
P_1	~~2 0 0~~ **3 0 2**	~~1 2 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.

Example of Banker's Algorithm (Cont.)

- Can further request for (1,0,2) by P_1 be granted?
  - Check if $Request_1 \le Need_1$, $(1,0,2) \not\le (0,2,0) \Rightarrow$ *false* $\Rightarrow$ error
- Can further request for (0,2,0) by P_0 be granted?
  - Check if $Request_0 \le Need_0$, $(0,2,0) \le (7,4,3) \Rightarrow$ *true*
  - Check that $Request_0 \le Available$, i.e. $(0,2,0) \le (2,3,0) \Rightarrow$ *true.*

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	~~0 1 0~~ 0 3 0	~~7 4 3~~ 7 2 3	~~2 3 0~~ 2 1 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	

☹We cannot find Needi<Available
→ Restore the state

- How about a request for (2,3,0) by P_4 ?

Comments are Welcome

- Online feedback sys
  your feedbacks (Du
- Your comments are
  - To have a self-a
  - To improve the future teaching
- Speak out:
  - If you love my le
    treasure the goo
  - If you think of an
    lecture…

Deadlock Detection

- Allow system to enter deadlock state
- Then invoke detection algorithms. There are two:
  - For single instance of each resource type (refer to textbook)
  - For multiple instances of each resource type
- Then invoke recovery algorithm (refer to textbook)

<u>This slide and the later slides in this chapter are not examinable.</u>

Multiple Instances of Each Resource Type

- **Available**: A vector of length *m* indicates the number of available resource types.
- **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.

Detection Algorithm (Cont.)

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 $m \times n^2$ 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 of Detection Algorithm(Cont.)

- Suppose P_2 requests an additional instance of type C.

	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 1	☹	
P_3	2 1 1	1 0 0	☹	
P_4	0 0 2	0 0 2	☹	

Work	Finished
A B C	
0 0 0	P_0
+ 0 1 0	
0 1 0	

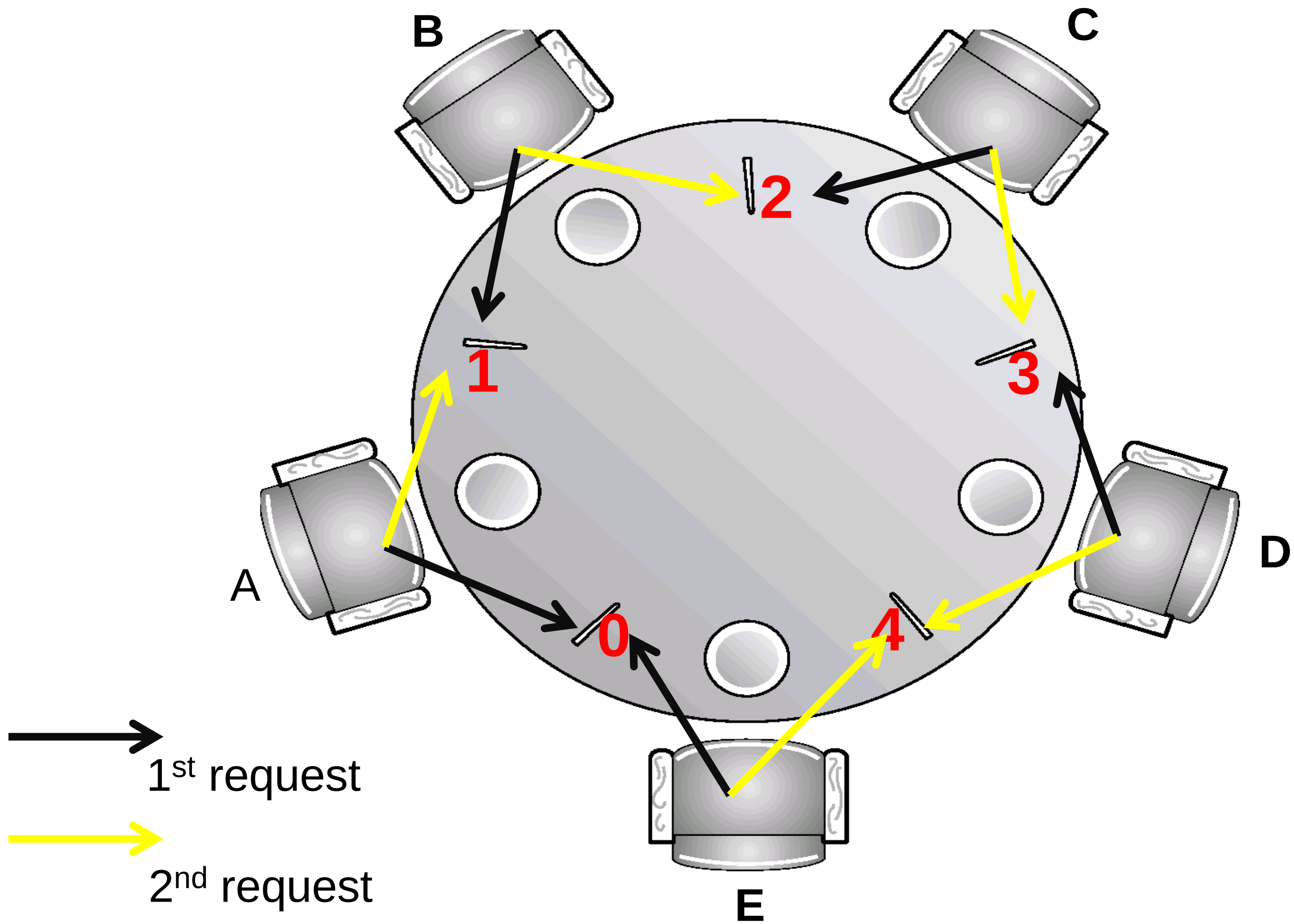
Example of Detection Algorithm(Cont.)

- 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.

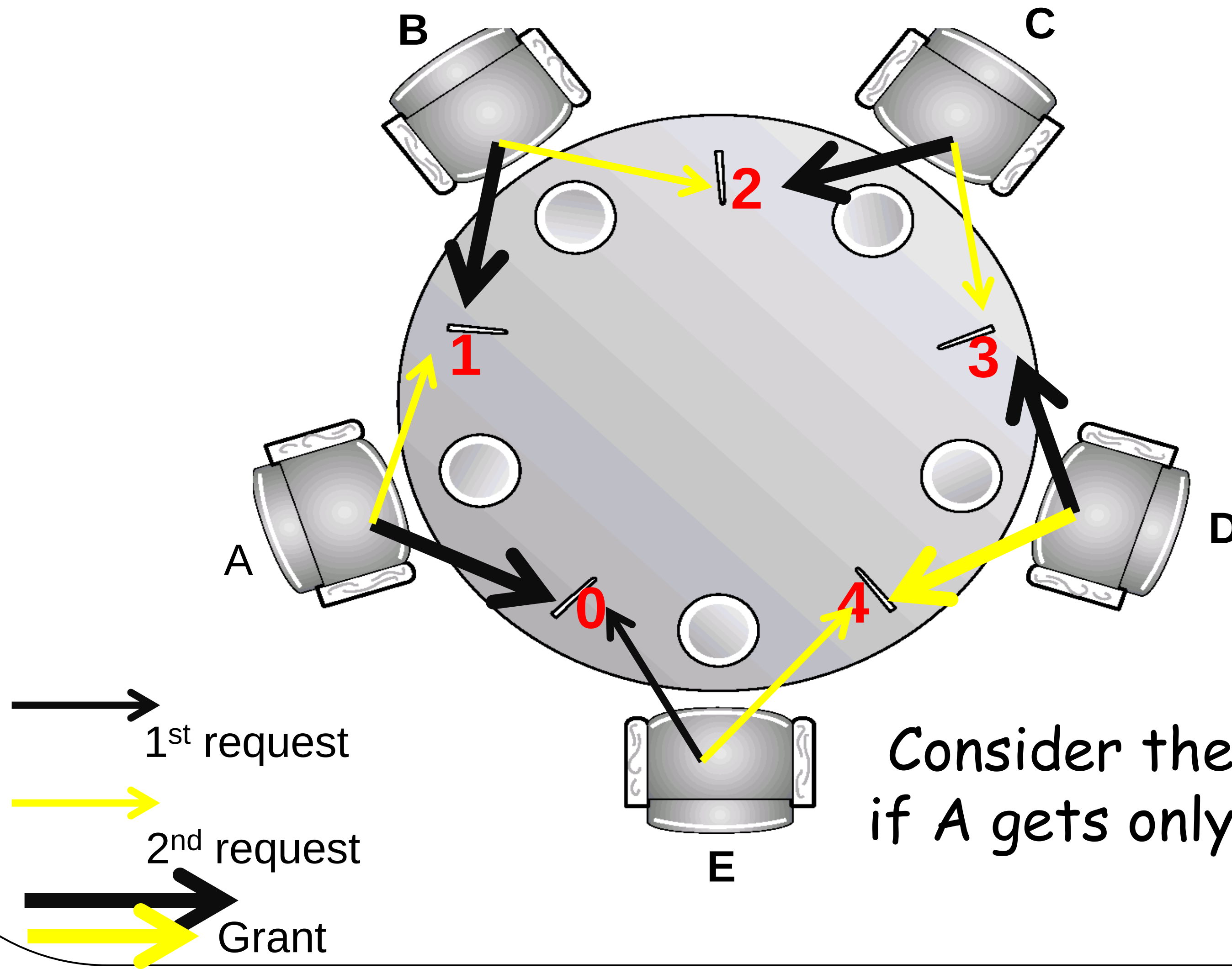
Advanced Readings

- “The Deadlock Problem: An Overview”, By Sreekaanth S. Isloor, T. Anthony Marsland. (pdf in EdveNTUre)
- Other readings
  - Java Concurrency, http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
  - Deadlock, http://en.wikipedia.org/wiki/Deadlock

An Example of Preventing Circular Wait



An Example of Preventing Circular Wait (Cont')



Consider the worst case if A gets only chopstick 0.