

Comments are Welcome

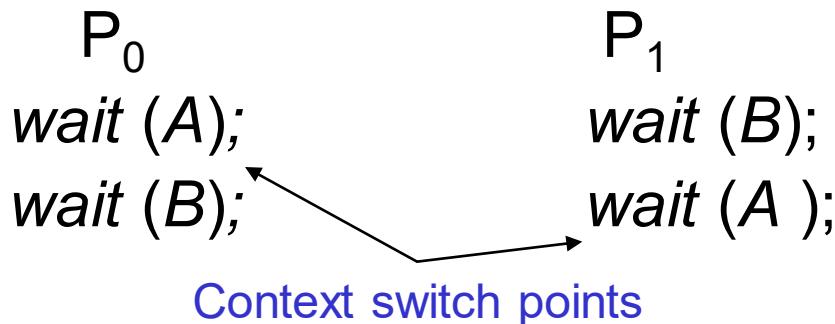
- Online feedback systems are open (Due soon)
 - <https://venus2.wis.ntu.edu.sg/SFT/Login.aspx>
- Your comments are **VERY important**:
 - To have a self-assessment process
 - To improve teaching and course content/structure
- Speak out:
 - If you like my lecture (Score 5), I will treasure the good teaching experience
 - If you think of any improvements that you can suggest, that would be great feedback as well

Part 5: Deadlocks and Starvation

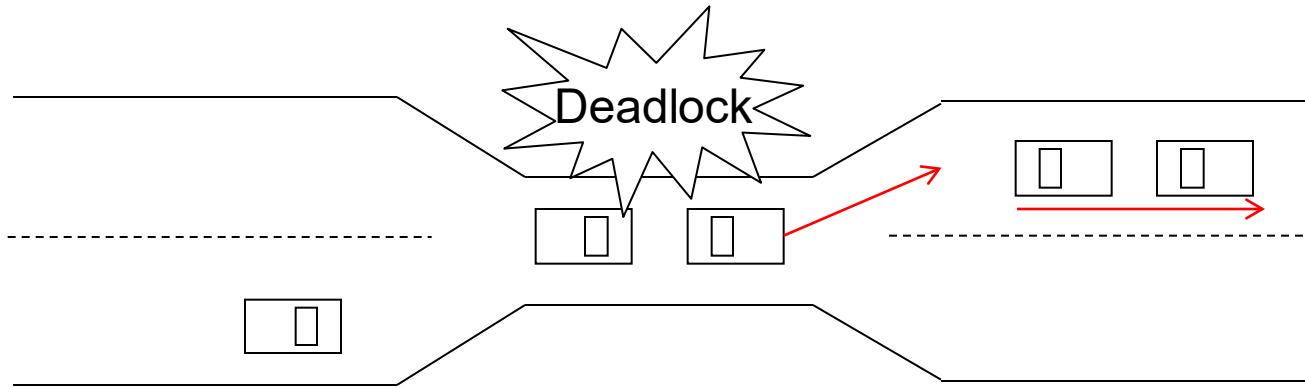
- **Deadlock Problem**
- System Model
- Deadlock Conditions
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection – **not examinable**

The Deadlock Problem

- **Deadlock:** A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- **Example 1:**
 - System has two different types of tape drives; both P_1 and P_2 need the two tapes to finish execution
 - P_1 and P_2 each hold one tape drive and need the other
- **Example 2:** Semaphores A and B , initialized to 1



Bridge Crossing Example



- Only one car is allowed on the bridge
- The bridge can be viewed as a shared 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

Part 5: Deadlocks and Starvation

- Deadlock Problem
- **System Model**
- Deadlock Conditions
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection

System Model

- Resource types R_1, R_2, \dots, R_m
 - Memory space, I/O devices, semaphores, etc.
- Each resource type R_i has W_i identical instances
- Each process utilizes a resource as follows:
 1. Request for a specific number of instances
 2. Use
 3. Release all the instances

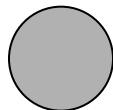
Resource-Allocation Graph

Graph → A set of vertices V and a set of edges E

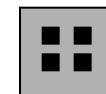
- V is partitioned into two types
 1. $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 2. $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- E is partitioned into two types
 - Request Edge: Directed edge $P_i \rightarrow R_j$
 - When the request is granted, this edge is removed
 - Assignment Edge: Directed edge $R_j \rightarrow P_i$
 - When the resource is released, this edge is removed

Resource-Allocation Graph (Cont.)

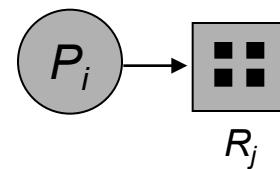
- Process



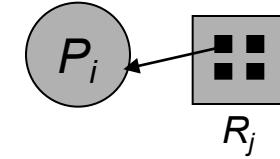
- Resource Type with 4 instances



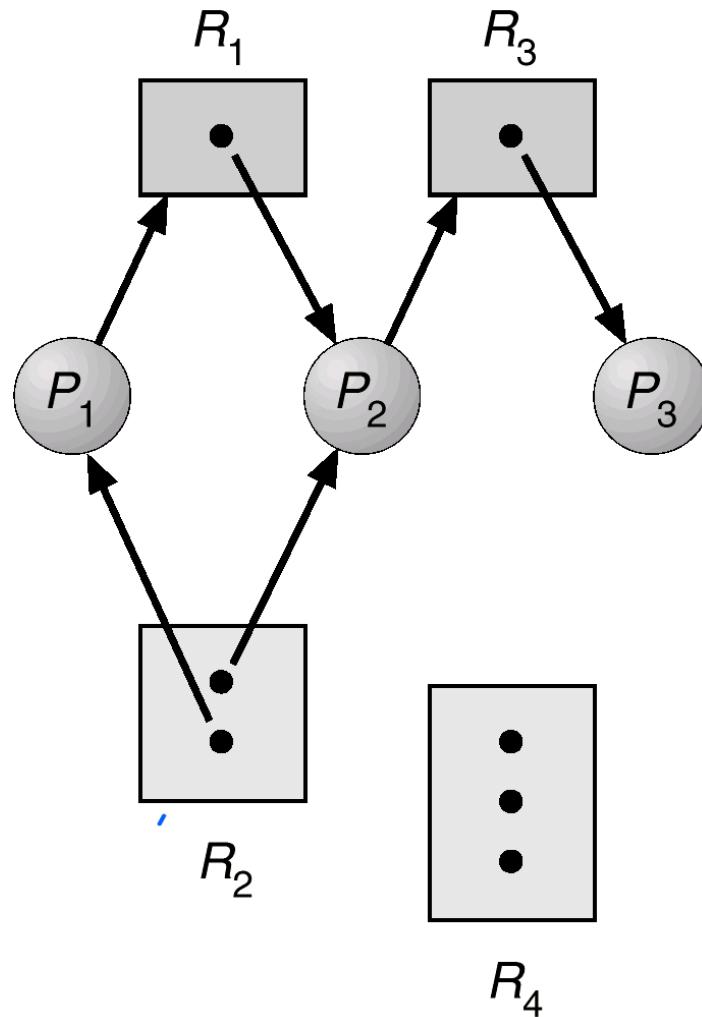
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

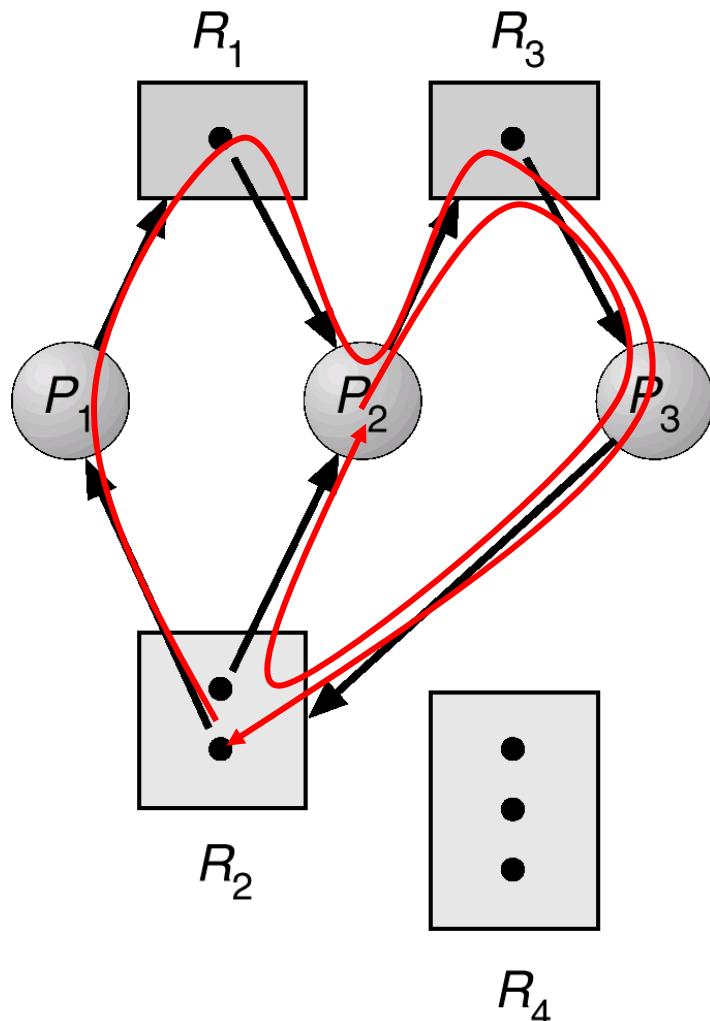


Is there a deadlock?
No deadlock

Can you identify a sequence
of process executions so
that all requests are met?

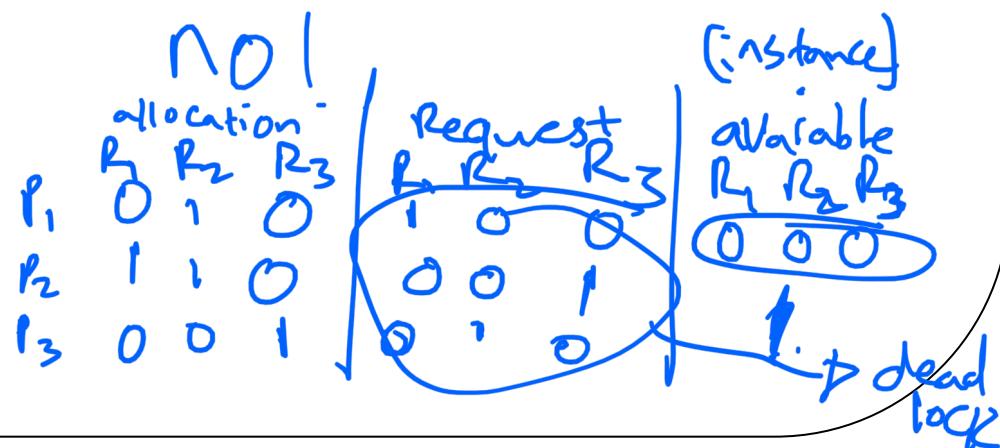
P3->P2->P1

Resource Allocation Graph With A Deadlock

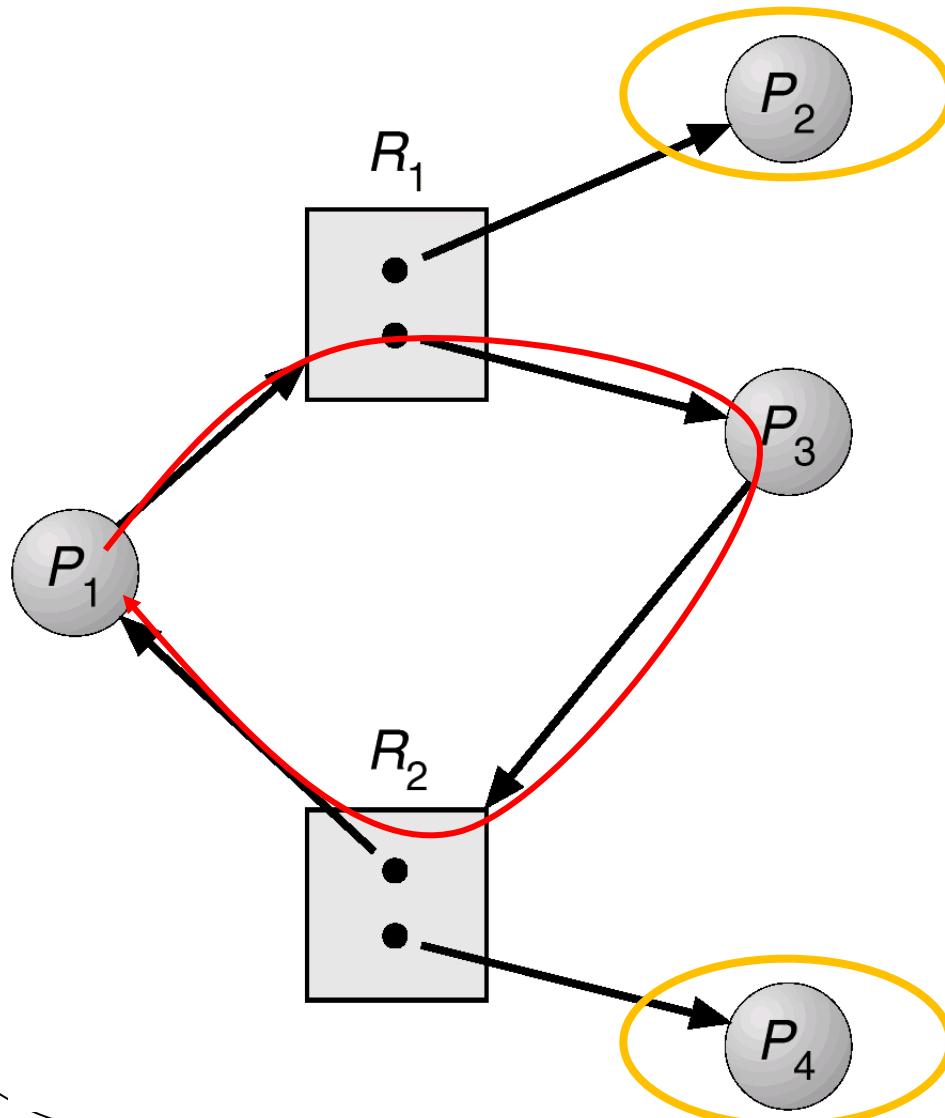


Use DFS or BFS
to detect cycle in
this graph

Does this mean a cycle in
the graph always indicates
a deadlock?



Resource Allocation Graph With A Cycle But No Deadlock



	allocation	request	available	
	R_1	R_2	R_1	R_2
P_1	0	1	1	0
P_2	1	0	0	0
P_3	1	0	0	1
P_4	0	1	0	0

Process execution order
that avoids a deadlock
 $P_2 \rightarrow P_4 \rightarrow P_3 \rightarrow P_1$

Basic Facts

- If graph contains no cycles \Rightarrow 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

Part 5: Deadlocks and Starvation

- Deadlock Problem
- System Model
- **Deadlock Conditions**
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection

Deadlock Conditions

Deadlock **may** arise if the following four conditions hold **simultaneously**
(necessary, but not sufficient)

1. **Mutual exclusion:** Only one process at a time can use a resource instance
2. **Hold and wait:** A process holding at least one resource is waiting to acquire additional resources held by other processes

Deadlock Conditions (Cont.)

3. **No preemption:** A resource can be released only voluntarily by the process holding it, after that process has completed its task
4. **Circular wait:** There exists a set $\{P_0, P_1, \dots, P_n\}$ 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, \dots, P_{n-1} is waiting for P_n , and P_n is waiting for P_0

Part 5: Deadlocks and Starvation

- Deadlock Problem
- System Model
- Deadlock Conditions
- **Deadlock Prevention**
- Deadlock Avoidance
- Deadlock Detection

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

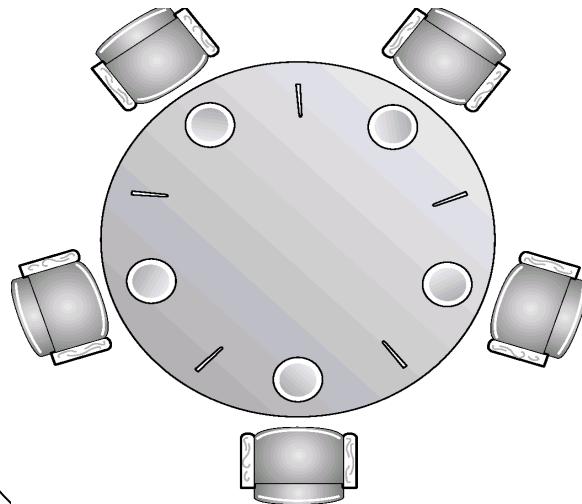


Gathings (c) 2007

Deadlock Prevention

Prevent at least one of the four deadlock conditions
We will illustrate using Dining-Philosophers Problem

- Recall dining-philosopher solution using semaphores
 - Each chopstick is a shared resource protected by a binary semaphore (**chopstick [5];**)
 - Initially, for all i, **chopstick [i] = 1;**
 - Code (Process Philosopher i)



```
while (1) {  
    wait(chopstick [ i ]);  
    wait(chopstick [ (i+1)%5 ]);  
    eat  
    signal(chopstick [ i ]);  
    signal(chopstick [ (i+1)%5 ]);  
    think  
}
```

Deadlock Prevention (Cont.)

Prevent at least one of the four deadlock conditions

We will illustrate using Dining-Philosophers Problem

1. Mutual Exclusion

- Chopsticks are not shareable (simultaneously), hence this condition cannot be eliminated

2. Hold and Wait: Must guarantee that whenever a process requests a resource, it does not hold any other resource

- Allow a philosopher to pick up chopsticks only if both the required chopsticks are available

Deadlock Prevention (Cont.)

Prevent at least one of the four deadlock conditions

We will illustrate using Dining-Philosophers Problem

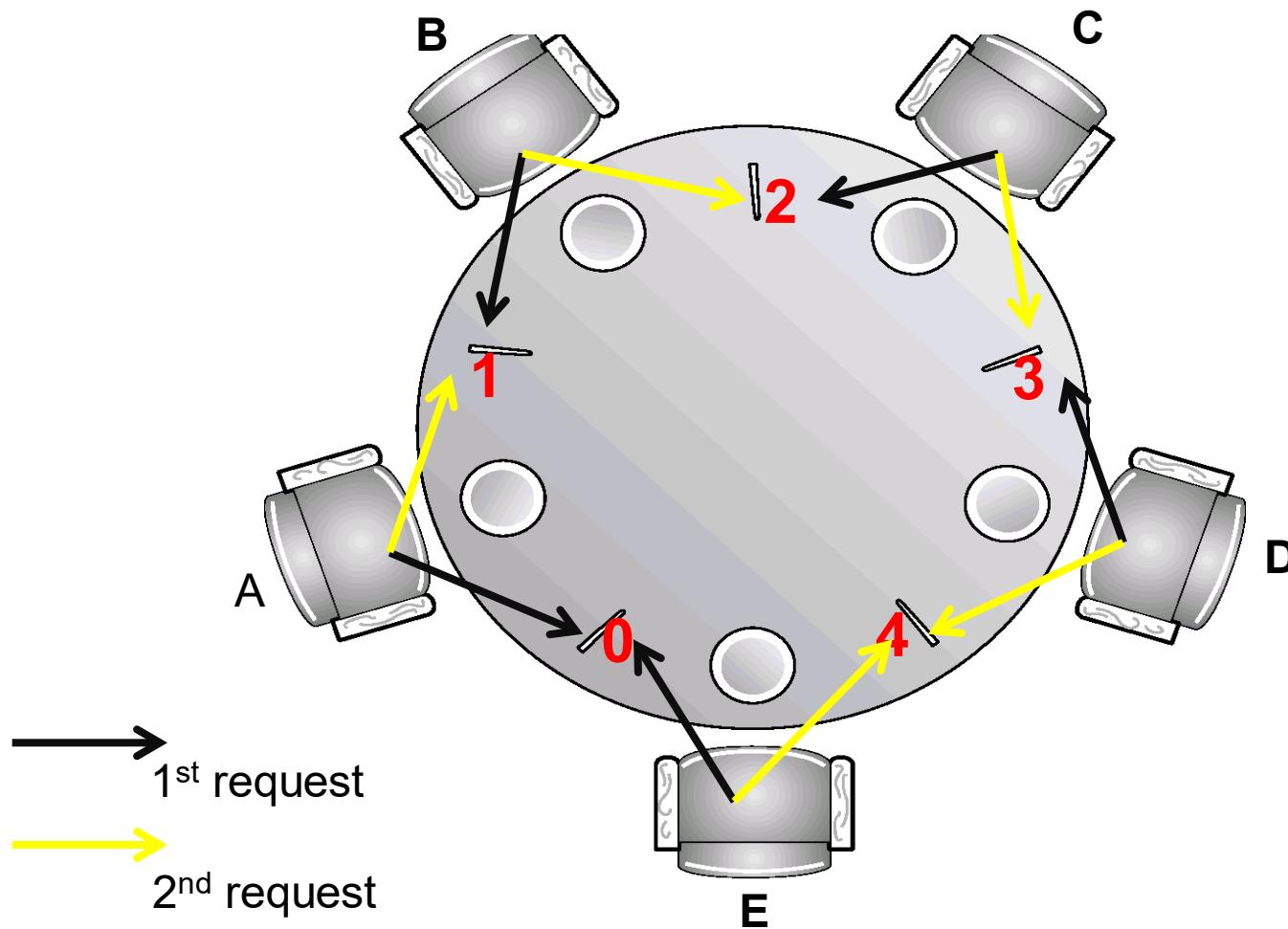
3. No Preemption

- If a philosopher cannot get the second chopstick, then preempt and release the chopstick that is being held

4. Circular Wait

1. Allow at most **four** philosophers to be hungry simultaneously
2. Odd-Even solution (see previous lecture)
3. Chopsticks can only be acquired in order (next slide)

Chopstick Ordering to Prevent Circular Wait



Part 5: Deadlocks and Starvation

- Deadlock Problem
- System Model
- Deadlock Conditions
- Deadlock Prevention
- **Deadlock Avoidance**
- Deadlock Detection

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, and otherwise, the process must wait
- System is in safe state if **there exists a *safe completion sequence* of all processes without deadlock**

Safe State

- A process completion sequence $\langle P_1, P_2, \dots, P_n \rangle$ 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 $P_j, j < i$
 - If P_i 's needs cannot be immediately met, then P_i can wait until all $P_j, j < i$ have finished
 - When all P_j ($j < i$) are finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
 - When P_i terminates, P_{i+1} can obtain its resources, and so on
 - All processes in the sequence can finish

Example of Safe Process Sequence

Available: 1

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

Q1: Is $\langle P1, P2, P3 \rangle$ safe?

Yes

$P1.request \leq Available$

$P2.request \leq P1.Hold + Available$

$P3.request \leq P1.Hold + P2.Hold + Available$

Q2: $\langle P3, P2, P1 \rangle$ safe?

No

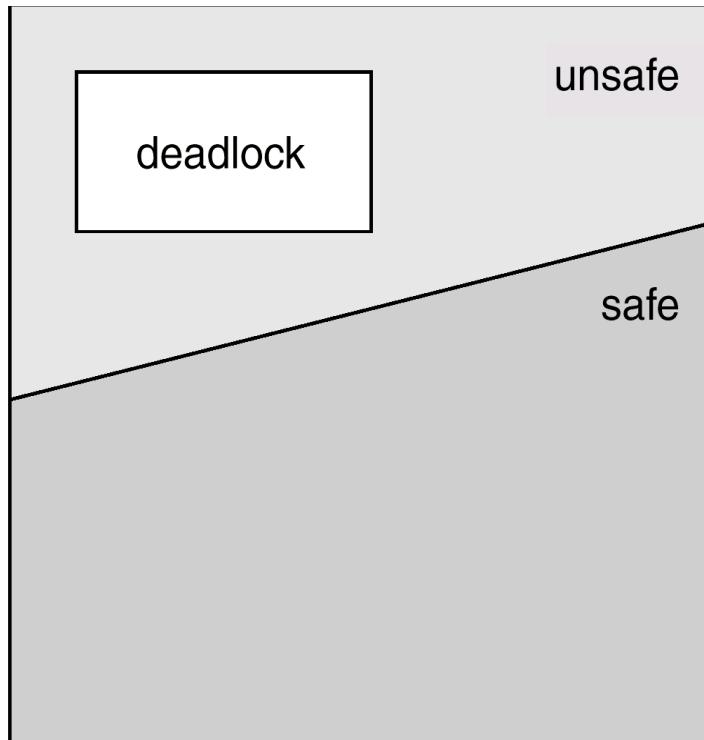
$P3.request > Available$

Q3: Is the system safe?

Yes

Exists one safe sequence
 $\langle P1, P2, P3 \rangle$

Safe, Unsafe and Deadlock States



- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
 - If a process releases resources much before its completion, then deadlock may not occur
- **Deadlock Avoidance:** Ensure that a system will never enter an unsafe state

Banker's Algorithm

- An algorithm to check whether satisfaction of a resource request would lead to an unsafe state
 - For multiple instances of each resource type
- When a process requests a resource it may have to wait (if the allocation leads to unsafe state)
- Assumptions
 - Each process must declare the maximum instances of each resource type that it needs
 - When a process gets all its resources it must return them in a finite amount of time

Data Structures for Banker's Algorithm

n = number of processes; m = number of resource types

- **Available** (vector of length m): If $\text{Available}[j] == k$, there are k instances of resource type R_j available
- **Max** ($n \times m$ matrix): If $\text{Max}[i, j] == k$, then process P_i may request at most k instances of resource type R_j
- **Allocation** ($n \times m$ matrix): If $\text{Allocation}[i, j] == k$, then P_i is currently allocated k instances of R_j
- **Need** ($n \times m$ matrix): If $\text{Need}[i, j] == k$, then P_i may need k more instances of R_j to complete its task

$$\text{Need}[i, j] = \text{Max}[i, j] - \text{Allocation}[i, j]$$

Banker's Algorithm Part 1: Safety Algorithm

1. Let **Work** and **Finish** be vectors of length **m** and **n**, respectively; initially
 - **Work = Available**
 - **Finish [i] = false for $i = 1, 2, \dots, n$**
2. Find an **i** such that (a) **Finish [i] == false** and (b) **Need [i, *] \leq Work**
 - If no such **i** exists, go to step 4

What if multiple answers?
3. **Work = Work + Allocation [i, *]**; **Finish[i] = true**
 - Go to step 2

*Work = Available + resources held by all P_j (**Finish [j] = true**)*
4. If **Finish [i] == true** for all **i**, then the system is in a safe state
 - Otherwise the system is in a unsafe state

time complexity
 $O(n^2m)$

Example of Safety 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 Safety Algorithm (Cont.)

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

	<u>Allocation</u>	<u>Max</u>	<u>Need</u>	<u>Available</u>
	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 Safety Algorithm (Cont.)

- The working is as follows:

	<u>Need</u>	<u>Allocation</u>	Work			Finished Process
	A	B	C			
P ₀	7 4 3 ✓ ↳	0 1 0	3 + 2	3 0	2 0	P ₁
P ₁	1 2 2 ✓ ↲	2 0 0	5 + 2	3 1	2 1	P ₃
P ₂	6 0 0 ✓ ↴	3 0 2	7	4	3	
P ₃	0 1 1 ✓ ↳ ↲	2 1 1	+ 0	0 0	2 2	P ₄
P ₄	4 3 1 ✓ ↳ ↴	0 0 2	7 + 3	4 0	5 2	P ₂
The system is safe (sequence <P ₁ , P ₃ , P ₄ , P ₂ , P ₀ >)			10 + 0	4 1	7 0	P ₀
				10	5	7

Banker's Part 2: Resource-Request 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**

$V1 \leq V2 \equiv V1[j] \leq V2[j], \text{ for all } j$

The i^{th} row of the matrix

1. If **Request_i ≤ Need_i**, go to step 2
 - Otherwise, raise error condition, since process has exceeded its maximum claim
2. If **Request_i ≤ Available**, go to step 3
 - Otherwise P_i must wait, since resources are not available

Resource-Request Algorithm (Cont.)

3. Pretend to allocate requested resources to P_i by modifying the state as follows:
 - $\text{Available} = \text{Available} - \text{Request}_i;$
 - $\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$
 - $\text{Need}_i = \text{Need}_i - \text{Request}_i;$
4. Run the safety algorithm
 - 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 Resource-Request Algorithm

- Suppose P_1 requests $(1, 0, 2)$ resources
- Step1: Check $\text{Request}_1 \leq \text{Need}_1$, $(1,0,2) \leq (1,2,2) \Rightarrow \text{true}$
- Step2: Check that $\text{Request}_1 \leq \text{Available}$, i.e. $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$
- 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
P_1	2	0	0	3	2	2	1	2	2	2	3	0
P_2	3	0	2	9	0	2	6	0	0	3	3	2
P_3	2	1	1	2	2	2	0	1	1	3	3	2
P_4	0	0	2	4	3	3	4	3	1	3	3	2

Step 3: Pretend to allocate requested resources

Step 4: Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ is safe

Example of Resource-Request Algorithm

- Can a further request for $(1,0,2)$ by P_1 be granted?
 - Step 1: Check if $\text{Request}_1 \leq \text{Need}_1$, $(1,0,2) \not\leq (0,2,0) \Rightarrow \text{false}$
- Can a further request for $(0,2,0)$ by P_0 be granted?
 - Step 1: Check if $\text{Request}_0 \leq \text{Need}_0$, $(0,2,0) \leq (7,4,3) \Rightarrow \text{true}$
 - Step 2: Check if $\text{Request}_0 \leq \text{Available}$, $(0,2,0) \leq (2,3,0) \Rightarrow \text{true}$

Allocation Need Available

A B C A B C A B C

	P ₀	P ₁	P ₂	P ₃	P ₄
	0 1 0	7 4 3	2 3 0		
	0 3 0	7 2 3	2 1 0		
	3 0 2	0 2 0	6 0 0	0 1 1	4 3 1

😦 We cannot find $\text{Need}_i < \text{Available}$
→ Restore the state

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

Part 5: Deadlocks and Starvation

- Deadlock Problem
- System Model
- Deadlock Conditions
- Deadlock Prevention
- Deadlock Avoidance
- **Deadlock Detection**

Deadlock Detection

- Allow system to enter deadlock state
- Then invoke detection algorithms
 - For single instance of each resource type (in textbook based on identifying cycle in resource-allocation graph)
 - For multiple instances of each resource type (next few slides)
- Then invoke recovery algorithm (in textbook)

This slide and the later slides in this chapter are not examinable

Multiple Instances of Each Resource Type

- **Available:** A vector of length m indicates the number of available resource types
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process
- **Request:** An $n \times m$ matrix indicates the current request of each process. If $\text{Request} [i, j] = k$, 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; initially
 - **Work = Available**
 - For $i = 1, 2, \dots, n$, if **Allocation_i != 0**, then **Finish[i] = false**, else **Finish[i] = true**
2. Find an **i** such that (a)**Finish [i] == false** and (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 system (process **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_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 $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ results in $\text{Finish}[i] = \text{true}$ for all i

Example of Detection Algorithm (Cont.)

Suppose P_2 requests an additional instance of type C

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>		Work	Finished
	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_0
P_1	2 0 0	2 0 2		0 0 0	+ 0 1 0	
P_2	3 0 3	0 0 1		0 0 0	0 1 0	
P_3	2 1 1	1 0 0		0 0 0		
P_4	0 0 2	0 0 2		0 0 0		

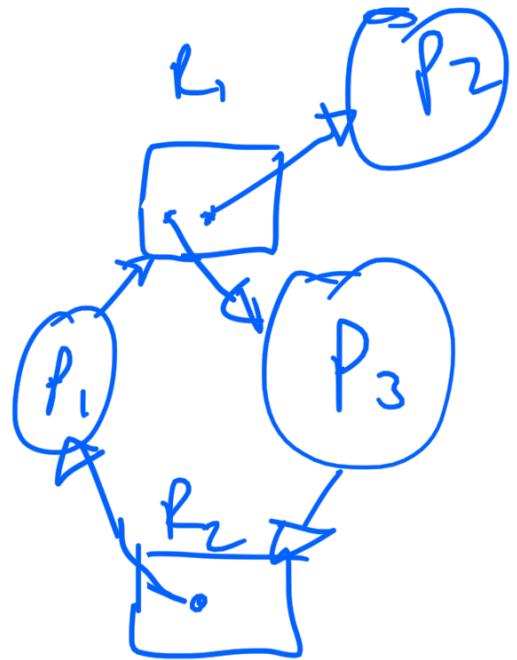
The diagram illustrates a state transition in a deadlock detection algorithm. The initial state shows processes P_0, P_1, P_2, P_3, P_4 with their current allocation and available resources. Process P_2 has requested an additional unit of resource C, indicated by a red checkmark above its row. The available resources are shown with a red sad face icon below each column. The final state shows the work and finished states after the request is processed. A vertical line separates the available resources from the work and finished states.

Example of Detection Algorithm (Cont.)

- State of system?
 - Can reclaim resources held by process P_0 , but insufficient resources to fulfill other process 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 NTULearn)
- Other readings
 - Java Concurrency,
<http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
 - Deadlock,
<http://en.wikipedia.org/wiki/Deadlock>



	Allocation		Request		Available	
	R ₁	R ₂	R ₁	R ₂	R ₁	R ₂
P ₁	0	1	1	0	0	0
P ₂	1	0	0	0	1	0
P ₃	1	0	0	1	1	0
					0	1
					1	1
					2 1	