

Deadlocks



Operating Systems
Wenbo Shen

Contents

- Deadlock problem
- System model
- Handling deadlocks
 - deadlock prevention
 - deadlock avoidance
 - deadlock detection
- Deadlock recovery

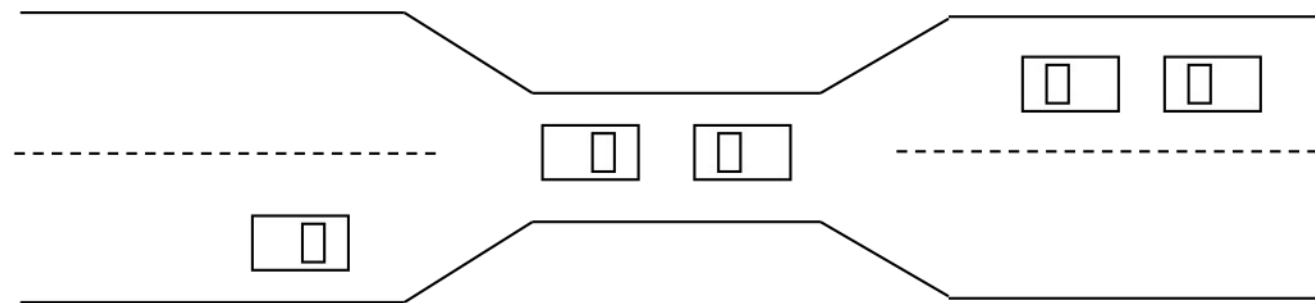
The Deadlock Problem

- **Deadlock:** a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Examples:
 - a system has 2 disk drives, P_1 and P_2 each hold one disk drive and each needs another one
 - semaphores A and B, initialized to 1

P_1	P_2
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example

- Traffic only in one direction, each section 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
 - starvation is possible
- Note: most OSes do not prevent or deal with deadlocks



System Model

- Resources: R_1, R_2, \dots, R_m
 - each represents a different **resource type**
 - e.g., CPU cycles, memory space, I/O devices
 - each resource type R_i has W_i **instances**.
- Each process utilizes a resource in the following pattern
 - request
 - use
 - release

Deadlock in program

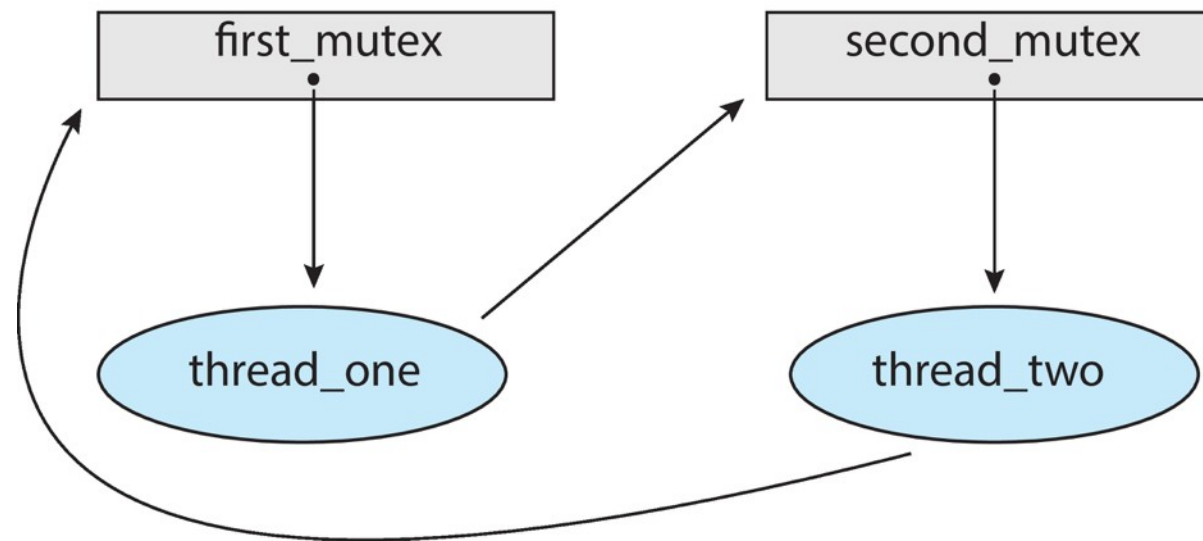
- Two mutex locks are created and initialized:

```
pthread_mutex_t first_mutex;  
pthread_mutex_t second_mutex;  
  
pthread_mutex_init(&first_mutex, NULL);  
pthread_mutex_init(&second_mutex, NULL);
```

```
/* thread_one runs in this function */  
void *do_work_one(void *param)  
{  
    pthread_mutex_lock(&first_mutex);  
    pthread_mutex_lock(&second_mutex);  
    /**  
     * Do some work  
     */  
    pthread_mutex_unlock(&second_mutex);  
    pthread_mutex_unlock(&first_mutex);  
  
    pthread_exit(0);  
}  
  
/* thread_two runs in this function */  
void *do_work_two(void *param)  
{  
    pthread_mutex_lock(&second_mutex);  
    pthread_mutex_lock(&first_mutex);  
    /**  
     * Do some work  
     */  
    pthread_mutex_unlock(&first_mutex);  
    pthread_mutex_unlock(&second_mutex);  
  
    pthread_exit(0);  
}
```

Deadlock in program

- Deadlock is possible if thread 1 acquires **first_mutex** and thread 2 acquires **second_mutex**. Thread 1 then waits for **second_mutex** and thread 2 waits for **first_mutex**.
- Can be illustrated with a **resource allocation graph**:



Review

- Problems of synchronization
- System model of deadlock

Four Conditions of Deadlock

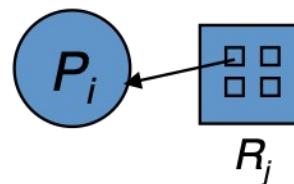
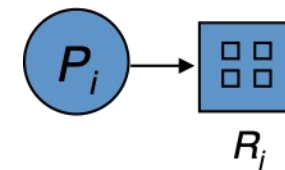
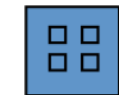
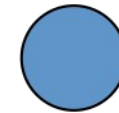
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after it has completed its task
- **Circular wait:** there exists a set of waiting processes $\{P_0, P_1, \dots, P_n\}$
 - P_0 is waiting for a resource that is held by P_1
 - P_1 is waiting for a resource that is held by $P_2 \dots$
 - P_{n-1} is waiting for a resource that is held by P_n
 - P_n is waiting for a resource that is held by P_0

Resource-Allocation Graph

- Two types of nodes:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set of all the **processes** in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set of all **resource** types in the system
- Two types of edges:
 - **request edge**: directed edge $P_i \rightarrow R_j$
 - **assignment edge**: directed edge $R_j \rightarrow P_i$

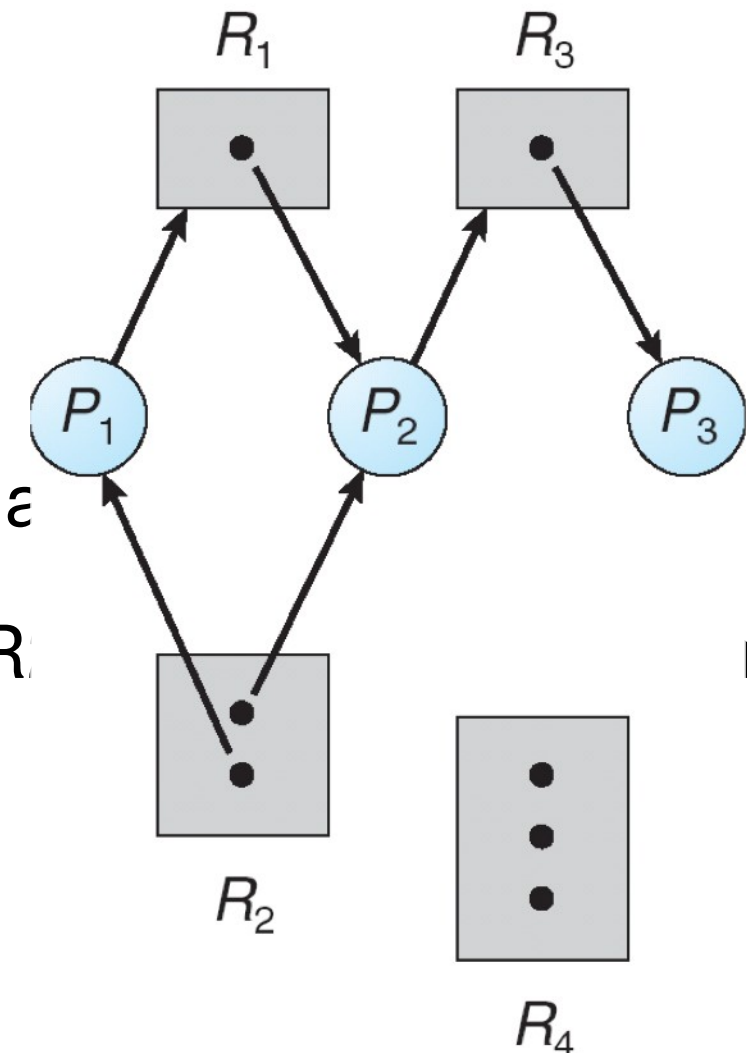
Resource-Allocation Graph

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



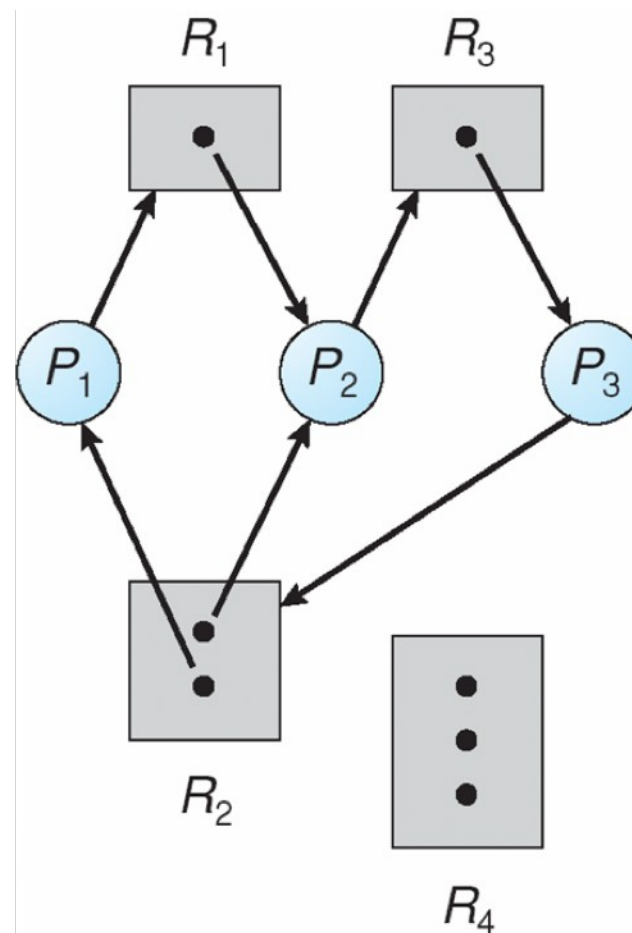
Resource Allocation Graph

- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- P_1 holds one instance of R_2 and is waiting for ε
- P_2 holds one instance of R_1 , one instance of R_2 , and one instance of R_3
- P_3 is holds one instance of R_3



Resource Allocation Graph

- Is there a deadlock?

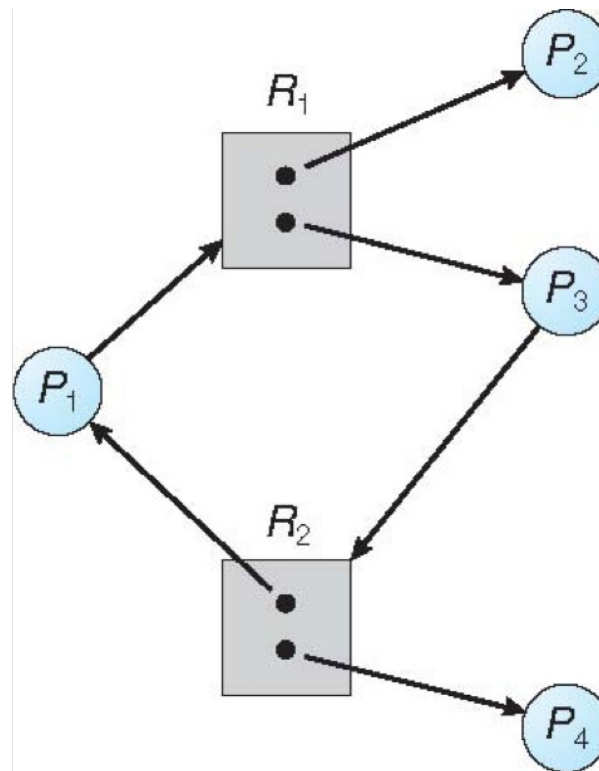


$p1 \rightarrow r1 \rightarrow p2 \rightarrow r3 \rightarrow p3 \rightarrow r2 \rightarrow p1$

$p2 \rightarrow r3 \rightarrow p3 \rightarrow r2 \rightarrow p2$

Resource Allocation Graph

- Is there a deadlock?
- **circular wait does not necessarily lead to deadlock**



$p1 \rightarrow r1 \rightarrow p3 \rightarrow r2 \rightarrow p1$

P4 releases first

Basic Facts

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

How to Handle Deadlocks

- Ensure that the system will never enter a deadlock state
 - **Prevention**
 - **Avoidance**
- Allow the system to enter a deadlock state and then recover - database
 - **Deadlock detection and recovery:**
- **Ignore the problem** and pretend deadlocks never occur in the system



Deadlock Prevention

- How to prevent **mutual exclusion**
 - not required for sharable resources
 - must hold for non-sharable resources
- How to prevent **hold and wait**
 - whenever a process requests a resource, it doesn't hold any other resources
 - require process to request ***all*** its resources before it begins execution
 - allow process to request resources only when the process has none
 - low resource utilization; starvation possible

Deadlock Prevention

- How to handle **no preemption**
 - if a process requests a resource not available
 - release all resources currently being held
 - preempted resources are added to the list of resources it waits for
 - process will be restarted only when it can get all waiting resources
- How to handle **circular wait**
 - impose a total ordering of all resource types
 - require that each process requests resources in an increasing order
 - Many operating systems adopt this strategy for some locks.

Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e. mutex locks) a unique number.
- Resources must be acquired in order.

first_mutex = 1
second_mutex = 5

code for thread_two could not be written as follows:



```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

For dynamic acquired lock

```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);

    acquire(lock1);
    acquire(lock2);

    withdraw(from, amount);
    deposit(to, amount);

    release(lock2);
    release(lock1);
}
```

```
transaction(checking_account, savings_account, 25.0)
transaction(savings_account, checking_account, 50.0)
```

Deadlock Avoidance

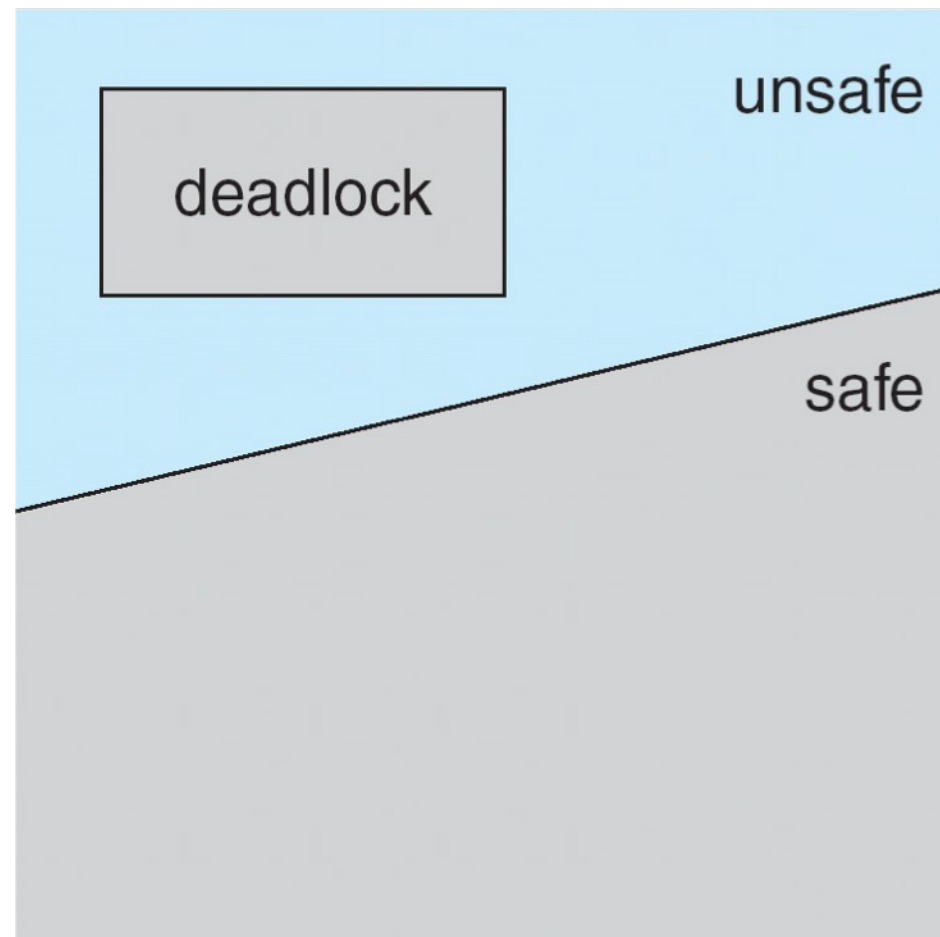
- Dead avoidance: require **extra information** about how resources are to be requested
 - **Is this requirement practical?**
- Each process declares a **max** number of resources it may need
- Deadlock-avoidance algorithm ensure there can **never** be a **circular-wait** condition
- Resource-allocation state:
 - the number of **available** and **allocated** resources
 - 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**:
 - there exists a **sequence** $\langle P_1, P_2, \dots, P_n \rangle$ of all processes in the system
 - for each P_i , resources that P_i can still request can be satisfied by currently **available resources** + **resources held by all the P_j , with $j < i$**
- **Safe state can guarantee no deadlock**
 - if P_i 's resource needs are not immediately available:
 - wait until all P_j have finished ($j < i$)
 - when P_j ($j < i$) has finished, P_i can obtain needed resources,
 - when P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in **safe state** \Rightarrow **no deadlocks**
- If a system is in **unsafe state** \Rightarrow **possibility of deadlock**
- **Deadlock avoidance** \Rightarrow ensure a system **never enters an unsafe state**



Example

- Resources: 12
 - Available is 3

	Max need	Current have	Extra need
P0	10	5	5
P1	4	2	2
P2	9	2	7

- Safe sequences: P1 P0 P2
 - P1 gets and return (5 in total)
 - and then P0 gets all and returns (10 in total)
 - and then P2

Example

- Safe sequences: P1 P0 P2
 - Available is 3

	Max need	Current have	Extra need
P0	10	5	5
P1	4	2	2
P2	9	2	7

Example

- Safe sequences: P1 P0 P2
 - Available is 3 -> give 2 to P1

	Max need	Current have	Extra need
P0	10	5	5
P1	4	2	2
P2	9	2	7

Example

- Safe sequences: P1 P0 P2
 - Available is **1** -> give 2 to P1

	Max need	Current have	Extra need
P0	10	5	5
P1	4	4	0
P2	9	2	7

- P1 gets and return -> Available is **5**

	Max need	Current have	Extra need
P0	10	5	5
P1	0	0	0
P2	9	2	7

Example

- Safe sequences: P1 P0 P2
 - Available is 3

	Max need	Current have	Extra need
P0	10	5	5
P1	4	2	2
P2	9	2	7

- P1 gets and return (5 in total) -> Available is 5

	Max need	Current have	Extra need
P0	10	5	5
P1	0	0	0
P2	9	2	7

Example

- Safe sequences: P1 P0 P2
 - P1 gets and return (available = 5),
 - and then P0 gets all needs and returns (available = 10),
 - and then P2
- What if we allocate 1 more for T2?
 - Available is 2

	Max need	Current have	Extra need
P0	10	5	5
P1	4	2	2
P2	9	3	6

Example

- Resources: 12
 - Available is 2

	Max need	Current have	Extra need
P0	10	5	5
P1	4	2	2
P2	9	3	6

- P1 gets and returns, available = 4, cannot fulfil the needs of P0 or P2

Deadlock Avoidance Algorithms

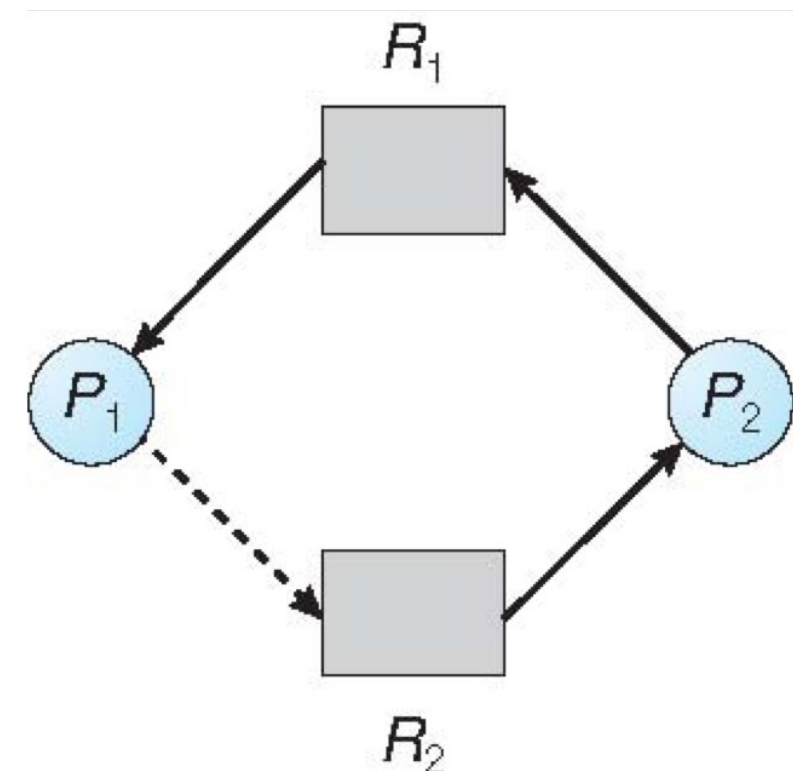
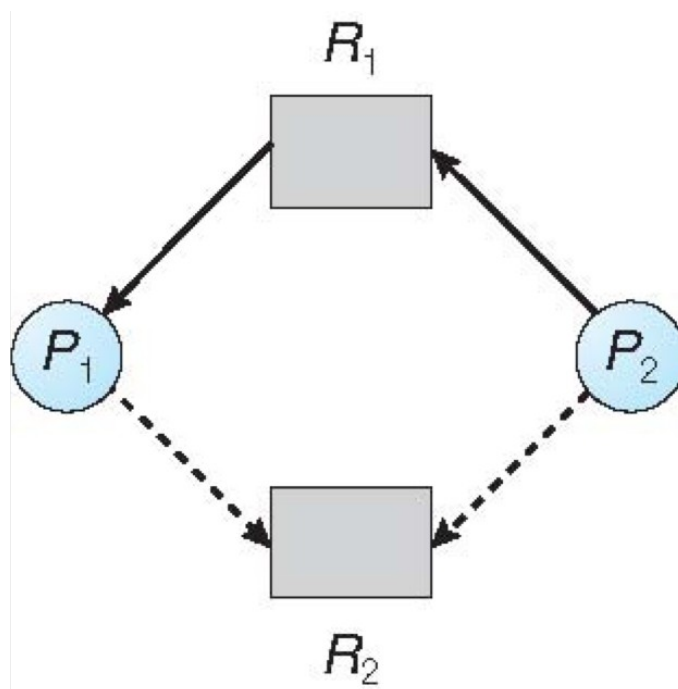
- Single instance of each resource type \Rightarrow use **resource-allocation graph**
- Multiple instances of a resource type \Rightarrow use the **banker's algorithm**

Single-instance Deadlock Avoidance

- Resource-allocation graph can be used for **single instance resource** deadlock avoidance
 - one new type of edge: **claim edge**
 - claim edge $P_i \rightarrow R_j$ indicates that process P_i **may** request resource R_j
 - claim edge is represented by a **dashed** line
 - **resources must be claimed a priori in the system**
- Transitions in between edges
 - ***claim edge*** converts to ***request edge*** when a process requests a resource
 - ***request edge*** converts to an ***assignment edge*** when the resource is allocated to the process
 - ***assignment edge*** reconverts to a ***claim edge*** when a resource is released by a process

Single-instance Deadlock Avoidance

- 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**
 - **no cycle** \Rightarrow **safe state**



Banker's Algorithm

- Banker's algorithm is for **multiple-instance resource deadlock avoidance**
 - each process must claim **maximum** use of each resource type **in advance**
 - when a process requests a resource it may have to wait
 - when a process gets all its resources it must release them in a finite amount of time

Data Structures for the Banker's Algorithm

- **n** processes, **m** types of resources
 - **available**: an array of length **m**, instances of available resource
 - $\text{available}[j] = k$: k instances of resource type R_j available
 - **max**: a **$n \times m$** matrix
 - $\text{max}[i,j] = k$: process P_i may request at most k instances of resource R_j
 - **allocation**: **$n \times m$** matrix
 - $\text{allocation}[i,j] = k$: P_i is currently allocated k instances of R_j
 - **need**: **$n \times m$** matrix
 - $\text{need}[i,j] = k$: 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: Example

- System state:
 - **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	

Banker's Algorithm: Safe State

- Data structure to compute whether the system is in a safe state
 - use **work** (a vector of length m) to track **allocatable resources**
 - **current available resources**
 - **unallocated + released by finished processes**
 - use **finish** (a vector of length n) to track whether process has finished
 - initialize: **work** = **available**, **finish[i] = false** for $i = 0, 1, \dots, n-1$
- Algorithm:
 - find an i such that **finish[i] = false && need[i] \leq work** if no such i exists, go to step 3
 - **work = work + allocation[i], finish[i] = true**, go to step 1
 - if **finish[i] == true** for all i , then the system is in a safe state

Bank's Algorithm: Resource Allocation

- Data structure: request vector for process P_i
 - **request**[j] = k then process P_i wants k instances of resource type R_j
- Algorithm:
 1. if $\text{request}[i] \leq \text{need}[i]$ go to step 2; otherwise, raise error condition (the process has exceeded its maximum claim)
 2. if $\text{request}[i] \leq \text{available}$, go to step 3; otherwise P_i must wait (not all resources are not available)
 3. pretend to allocate requested resources to P_i by modifying the state:
$$\text{available} = \text{available} - \text{request}[i]$$
$$\text{allocation}[i] = \text{allocation}[i] + \text{request}[i]$$
$$\text{need}[i] = \text{need}[i] - \text{request}[i]$$
 4. use previous algorithm to test if it is a safe state, if so \Rightarrow allocate the resources to P_i
 5. if unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored

Banker's Algorithm: Example

- System state:
 - **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		
	A	B	C	A	B	C
P_0	0	1	0	7	5	3
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

available		
A	B	C
3	3	2

Banker's Algorithm: Example

- System state:
 - **5 processes** P_0 through P_4
 - **3 resource types**: A (10 instances), B (5 instances), and C (7 instances)
- $\text{need} = \text{max} - \text{allocation}$

	allocation	max	need
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	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

available
A B C
3 3 2

Banker's Algorithm: Example

- System state:
 - **5 processes** P_0 through P_4
 - **3 resource types**: A (10 instances), B (5 instances), and C (7 instances)
- $\text{need} = \text{max} - \text{allocation}$

	allocation	max	need
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	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

available
A B C
3 3 2

- First one can be either P_1 or P_3

Banker's Algorithm: Example

- What's the safe sequence

	allocation	max	need
	A B C	A B C	A B C
P ₀	0 1 0	7 5 3	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

available
A B C
3 3 2

- finish[1] = true, needed[1] < work -> work = work + allocation = [5 3 2]
- finish[3] = true, needed[3] < work -> work = work + allocation = [7 4 3]
- finish[4] = true, needed[4] < work -> work = work + allocation = [7 4 5]
- finish[2] = true, needed[2] < work -> work = work + allocation = [10 4 7]
- finish[0] = true, needed[0] < work -> work = work + allocation = [10 5 7]

Banker's Algorithm: Example

- P1 allocates 1 0 2 more

	allocation	max	need
	A B C	A B C	A B C
P ₀	0 1 0	7 5 3	7 4 3
P ₁	3 0 2	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

available
A B C
2 3 0

- Check whether it is in safe state?
 - We cannot find a process that the $\text{need}[i] < \text{work}[i]$

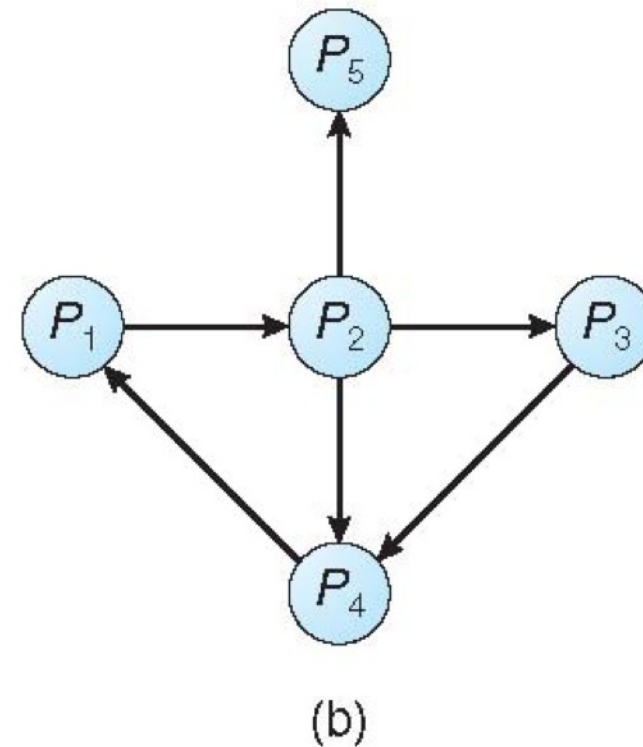
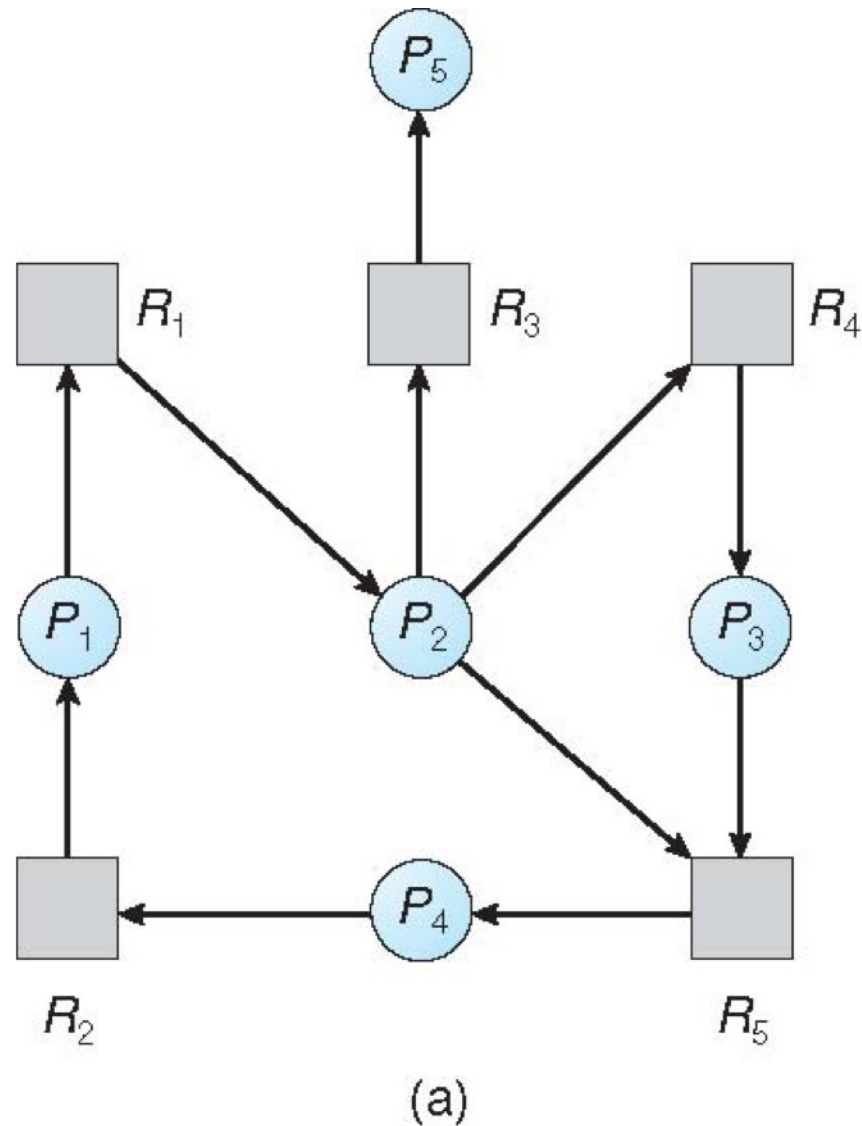
Deadlock Detection

- Allow system to enter deadlock state, but detect and recover from it
- Detection algorithm and recovery scheme

Deadlock Detection: Single Instance Resources

- Maintain a 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

Wait-for Graph Example



Resource-allocation Graph

wait-for graph

Deadlock Detection: Multi-instance Resources

- Detection algorithm similar to Banker's algorithm's safety condition
 - to prove it is **not possible** to enter a **safe state**
- Data structure
 - **available**: a vector of length m , number of available resources of each type
 - **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
 - request $[i, j] = k$: process P_i is requesting k more instances of resource R_j
 - **work**: a vector of m , the allocatable instances of resources
 - **finish**: a vector of m , whether the process has finished
 - if $\text{allocation}[i] \neq 0 \Rightarrow \text{finish}[i] = \text{false}$; otherwise, $\text{finish}[i] = \text{true}$

Deadlock Detection: Multi-instance

- Find an process i such that **$\text{finish}[i] == \text{false} \ \&\& \ \text{request}[i] \leq \text{work}$**
 - if no such i exists, go to step 3
- **$\text{work} = \text{work} + \text{allocation}[i]$; $\text{finish}[i] = \text{true}$** , go to step 1
- If $\text{finish}[i] == \text{false}$, for some i the system is in deadlock state
 - if $\text{finish}[i] == \text{false}$, then P_i is deadlocked

Example of Detection Algorithm

- System states:
 - 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	

P_0 : [0 0 0] -> [0 1 0]
 P_2 : [0 1 0] -> [3 1 3]
 P_3 : [3 1 3] -> [5 2 4]
 P_1 : [5 2 4] -> [7 2 4]
 P_4 : [7 2 4] -> [7 2 6]

- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $\text{finish}[i] = \text{true}$ for all i

Example (Cont.)

- P2 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	

- State of system?
 - can reclaim resources held by process P₀, but insufficient resources to fulfill other processes; requests
 - P1: [0 0 0] -> [0 1 0]
 - deadlock exists, consisting of processes P₁, P₂, P₃, and P₄

Deadlock Recovery: Option I

- Terminate deadlocked processes. options:
 - 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?

Deadlock Recovery: Option II

- Resource preemption
 - Select a victim
 - Rollback
 - Starvation
 - How could you ensure that the resources do not preempt from the same process?

Takeaway

- Deadlock occurs in which condition?
- Four conditions for deadlock
- Deadlock can be modeled via resource-allocation graph
- Deadlock can be prevented by breaking one of the four conditions
- Deadlock can be avoided by using the banker's algorithm
- A deadlock detection algorithm
- Deadlock recovery

- Class on Oct 31 cancelled
 - No Class on Nov 4 and Nov 7
 - Make up on Nov 11 and Nov 14 (6, 7, 8)
-
- Regular Lab 1 due on Oct 29th
 - Advanced Lab 2 due on Nov 7th
 - Homework 2 due on Nov 7th