



# Processes Synchronization - Part II

Amir H. Payberah  
payberah@kth.se  
2022



# Deadlocks



## Motivation

- ▶ **Multiprogramming** environment: **several processes** compete for a **finite number of resources**.



# Motivation

- ▶ **Multiprogramming** environment: **several processes** compete for a **finite number of resources**.
- ▶ A process requests resources: if the resources are **not available** at that time, the process enters a **waiting state**.



# Motivation

- ▶ **Multiprogramming** environment: **several processes** compete for a **finite number of resources**.
- ▶ A process requests resources: if the resources are **not available** at that time, the process enters a **waiting state**.
- ▶ What if the requests resources are **held by other waiting processes**?



# Motivation

- ▶ **Multiprogramming** environment: **several processes** compete for a **finite number of resources**.
- ▶ A process requests resources: if the resources are **not available** at that time, the process enters a **waiting state**.
- ▶ What if the requests resources are **held by other waiting processes**?
- ▶ This situation is called a **deadlock**.



## Deadlock System Model

- ▶ System consists of  $m$  resources:  $R_1, R_2, \dots, R_m$



## Deadlock System Model

- ▶ System consists of  $m$  resources:  $R_1, R_2, \dots, R_m$
- ▶ Resource types: CPU cycles, memory space, I/O devices





## Deadlock System Model

- ▶ System consists of  $m$  resources:  $R_1, R_2, \dots, R_m$
- ▶ Resource types: CPU cycles, memory space, I/O devices
- ▶ Each resource type  $R_i$  has  $W_i$  instances.



# Deadlock System Model

- ▶ System consists of  $m$  resources:  $R_1, R_2, \dots, R_m$
- ▶ Resource types: CPU cycles, memory space, I/O devices
- ▶ Each resource type  $R_i$  has  $W_i$  instances.
- ▶ Each process utilizes a resource as follows:
  - Request
  - Use
  - Release



## Deadlock Characterization (1/3)

- ▶ Deadlock can arise if **four conditions** hold **simultaneously**:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait



## Deadlock Characterization (2/3)

▶ **Mutual exclusion**

- Only one process at a time can use a resource.



## Deadlock Characterization (2/3)

### ▶ 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 Characterization (3/3)

▶ **No preemption**

- A resource can be released only **voluntarily** by the process holding it, after that process has **completed its task**.



## Deadlock Characterization (3/3)

### ▶ No preemption

- A resource can be released only **voluntarily** by the process holding it, after that process has **completed its task**.

### ▶ Circular wait

- A set 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  $P_2$
- ...
- $P_n$  is **waiting** for a resource that is held by  $P_0$



## Deadlock Example (1/2)

```
/* Create and initialize the mutex locks */  
pthread_mutex_t first_mutex;  
pthread_mutex_t second_mutex;  
  
pthread_mutex_init(&first_mutex, NULL);  
pthread_mutex_init(&second_mutex, NULL);
```





## Deadlock Example (2/2)

```
void *thread_one(void *args) {  
    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);  
}
```



## Deadlock Example (2/2)

```
void *thread_one(void *args) {  
    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);  
}
```

```
void *thread_two(void *args) {  
    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);  
}
```



# Resource-Allocation Graph



## Resource-Allocation Graph (1/2)

- ▶ A set of **vertices**  $V$  and a set of **edges**  $E$ .



## Resource-Allocation Graph (1/2)

- ▶ A set of **vertices**  $V$  and a set of **edges**  $E$ .
- ▶ **Vertices**
  - All the **processes** in the system:  $P = P_1, P_2, \dots, P_n$
  - All **resource types** in the system:  $R = R_1, R_2, \dots, R_m$



## Resource-Allocation Graph (1/2)

- ▶ A set of **vertices**  $V$  and a set of **edges**  $E$ .
- ▶ **Vertices**
  - All the **processes** in the system:  $P = P_1, P_2, \dots, P_n$
  - All **resource types** in the system:  $R = R_1, R_2, \dots, R_m$
- ▶ **Edges**
  - **Request edge**: directed edge  $P_i \rightarrow R_j$
  - **Assignment edge**: directed edge  $R_j \rightarrow P_i$



## Resource-Allocation Graph (2/2)

- ▶ Process (**vertices**)



## Resource-Allocation Graph (2/2)

▶ Process (**vertices**)



▶ Resource type with 4 instances (**vertices**)





## Resource-Allocation Graph (2/2)

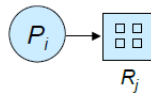
- ▶ Process (vertices)



- ▶ Resource type with 4 instances (vertices)



- ▶  $P_i$  requests instance of  $R_j$  (edge)



## Resource-Allocation Graph (2/2)

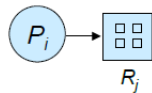
- ▶ Process (**vertices**)



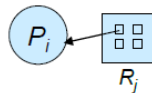
- ▶ Resource type with 4 instances (**vertices**)



- ▶  $P_i$  requests instance of  $R_j$  (**edge**)

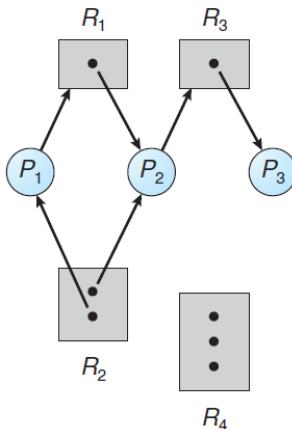


- ▶  $P_i$  is holding an instance of  $R_j$  (**edge**)



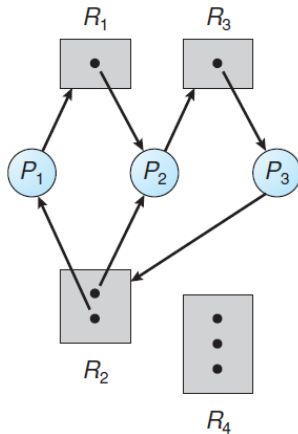
# Resource-Allocation Graph Example (1/3)

- ▶ Example of a resource allocation graph.



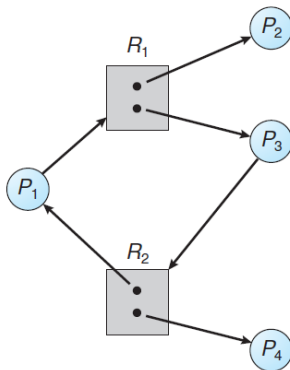
## Resource-Allocation Graph Example (2/3)

- ▶ Resource allocation graph **with a deadlock**.



## Resource-Allocation Graph Example (3/3)

- ▶ Resource allocation graph with a **cycle** but no deadlock.





## Basic Facts

- ▶ If graph contains no cycles
  - No deadlock



## Basic Facts

- ▶ If graph contains **no cycles**
  - **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**.



## Methods for Handling Deadlocks

- ▶ Ensure that the system will **never enter a deadlock state**:





# Methods for Handling Deadlocks

- ▶ Ensure that the system will **never enter a deadlock state**:
  - **Deadlock prevention**
  - **Deadlock avoidance**



# Methods for Handling Deadlocks

- ▶ Ensure that the system will **never enter a deadlock state**:
  - **Deadlock prevention**
  - **Deadlock avoidance**
  
- ▶ **Allow** the system to **enter a deadlock** state and then **recover**.



# Methods for Handling Deadlocks

- ▶ Ensure that the system will **never enter a deadlock state**:
  - **Deadlock prevention**
  - **Deadlock avoidance**
- ▶ **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**.



# Deadlock Prevention



## Deadlock Prevention (1/3)

- ▶ Deadlock can arise if **four conditions** hold **simultaneously**:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait



## Deadlock Prevention (1/3)

- ▶ Deadlock can arise if **four conditions** hold **simultaneously**:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
  
- ▶ **Restrain** the ways requests can be made.



## Deadlock Prevention (2/3)

### ▶ Mutual exclusion

- Not required for **sharable** resources, e.g., **read-only files**.
- Must hold for **non-sharable** resources.



## Deadlock Prevention (2/3)

### ▶ Mutual exclusion

- Not required for **sharable** resources, e.g., **read-only files**.
- Must hold for **non-sharable** resources.

### ▶ Hold and wait

- Must guarantee that whenever a process requests a resource, it **does not hold any other resources**.





## Deadlock Prevention (2/3)

### ▶ Mutual exclusion

- Not required for **sharable** resources, e.g., **read-only files**.
- Must hold for **non-sharable** resources.

### ▶ Hold and wait

- Must guarantee that whenever a process requests a resource, it **does not hold any other resources**.
- **Solution 1**: require a process to request and be **allocated all its resources before it begins execution**.



## Deadlock Prevention (2/3)

### ▶ Mutual exclusion

- Not required for **sharable** resources, e.g., **read-only files**.
- Must hold for **non-sharable** resources.

### ▶ Hold and wait

- Must guarantee that whenever a process requests a resource, it **does not hold any other resources**.
- **Solution 1**: require a process to request and be **allocated all its resources before it begins execution**.
- **Solution 2**: allows a process to request resources only when **it has none**.



## Deadlock Prevention (2/3)

### ▶ Mutual exclusion

- Not required for **sharable** resources, e.g., **read-only files**.
- Must hold for **non-sharable** resources.

### ▶ Hold and wait

- Must guarantee that whenever a process requests a resource, it **does not hold any other resources**.
- **Solution 1**: require a process to request and be **allocated all its resources before it begins execution**.
- **Solution 2**: allows a process to request resources only when it has none.
- **Low resource utilization**
- **Starvation** possible



## Deadlock Prevention (3/3)

- ▶ No preemption



## Deadlock Prevention (3/3)

### ▶ 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**.



## Deadlock Prevention (3/3)

### ▶ 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.



## Deadlock Prevention (3/3)

### ▶ 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.



## Deadlock Prevention (3/3)

### ▶ 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.

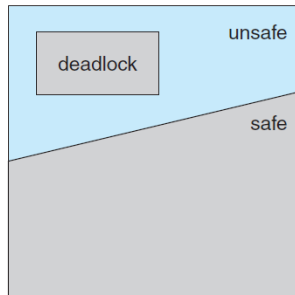




# Deadlock Avoidance

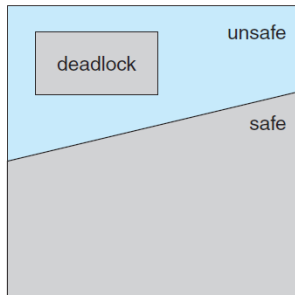
## Basic Facts

- ▶ If a system is in the **safe state**
  - **No deadlock**



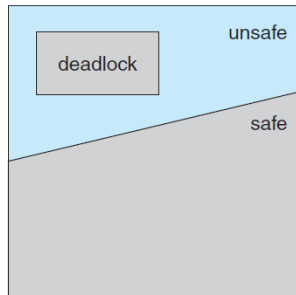
## Basic Facts

- ▶ If a system is in the **safe state**
  - **No deadlock**
- ▶ If a system is in the **unsafe state**
  - **Possibility of deadlock**



## Basic Facts

- ▶ If a system is in the **safe state**
  - **No deadlock**
- ▶ If a system is in the **unsafe state**
  - **Possibility of deadlock**
- ▶ **Avoidance**
  - Ensure that a system will never enter an unsafe state.





## Safe State (1/2)

- ▶ When a process requests an **available resource**, system must decide if immediate allocation leaves the system in a **safe state**.



## Safe State (1/2)

- ▶ When a process requests an **available resource**, system must decide if immediate allocation leaves the system in a **safe state**.
- ▶ **Safe state**: there exists a sequence  $\langle P_1, P_2, \dots, P_n \rangle$  of **all** the processes in the systems such that for each  $P_i$ , the resources that  $P_i$  **can still request** be satisfied by:  
**currently available resources + resources held by all the  $P_j$ , with  $j < i$ .**



## Safe State (2/2)

- ▶ If  $P_i$  resource needs are **not immediately available**, then  $P_i$  can **wait** until all  $P_j$  have finished.



## Safe State (2/2)

- ▶ 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.





## Safe State (2/2)

- ▶ 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.



## Safe Mode Example (1/2)

- ▶ 3 processes:  $P_0$  through  $P_2$
- ▶ 1 resource type:
  - $A$  (12 instances)
- ▶ Snapshot at time  $T_0$

	<u>Maximum Needs</u>	<u>Current Needs</u>
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2



## Safe Mode Example (1/2)

- ▶ 3 processes:  $P_0$  through  $P_2$
- ▶ 1 resource type:
  - $A$  (12 instances)
- ▶ Snapshot at time  $T_0$

	<u>Maximum Needs</u>	<u>Current Needs</u>
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

- ▶ Safe mode sequence?



## Safe Mode Example (1/2)

- ▶ 3 processes:  $P_0$  through  $P_2$
- ▶ 1 resource type:
  - $A$  (12 instances)
- ▶ Snapshot at time  $T_0$

	<u>Maximum Needs</u>	<u>Current Needs</u>
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

- ▶ Safe mode sequence?  $\langle P_1, P_0, P_2 \rangle$



## Safe Mode Example (2/2)

- ▶ 3 processes:  $P_0$  through  $P_2$
- ▶ 1 resource type:
  - $A$  (12 instances)
- ▶ Snapshot at time  $T_0$

	<u>Maximum Needs</u>	<u>Current Needs</u>
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2



## Safe Mode Example (2/2)

- ▶ 3 processes:  $P_0$  through  $P_2$
- ▶ 1 resource type:
  - $A$  (12 instances)
- ▶ Snapshot at time  $T_0$

	<u>Maximum Needs</u>	<u>Current Needs</u>
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

- ▶ Suppose that, at time  $T_1$ , process  $P_2$  requests and is allocated one more resource.



## Safe Mode Example (2/2)

- ▶ 3 processes:  $P_0$  through  $P_2$
- ▶ 1 resource type:
  - $A$  (12 instances)
- ▶ Snapshot at time  $T_0$

	<u>Maximum Needs</u>	<u>Current Needs</u>
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

- ▶ Suppose that, at time  $T_1$ , process  $P_2$  requests and is allocated one more resource.
- ▶ **Safe mode sequence?**



## Safe Mode Example (2/2)

- ▶ 3 processes:  $P_0$  through  $P_2$
- ▶ 1 resource type:
  - $A$  (12 instances)
- ▶ Snapshot at time  $T_0$

	<u>Maximum Needs</u>	<u>Current Needs</u>
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

- ▶ Suppose that, at time  $T_1$ , process  $P_2$  requests and is allocated one more resource.
- ▶ **Safe mode sequence?** Not safe





# Avoidance Algorithms

- ▶ Single instance of a resource type
  - Use a resource-allocation graph



# Avoidance Algorithms

- ▶ Single instance of a resource type
  - Use a resource-allocation graph
  
- ▶ Multiple instances of a resource type
  - Use the banker's algorithm



# Resource-Allocation Graph Algorithm



## Resource-Allocation Graph Scheme

- ▶ **Claim edge**  $P_i \rightarrow R_j$ : indicates that process  $P_i$  may **request** resource  $R_j$ ; represented by a dashed line



## Resource-Allocation Graph Scheme

- ▶ **Claim edge**  $P_i \rightarrow R_j$ : indicates that process  $P_i$  may **request** resource  $R_j$ ; represented by a dashed line
- ▶ **Claim edge** converts to **request edge** when a process **requests a resource**.



## Resource-Allocation Graph Scheme

- ▶ **Claim edge**  $P_i \rightarrow R_j$ : indicates that process  $P_i$  may **request** resource  $R_j$ ; represented by a dashed line
- ▶ **Claim edge** converts to **request edge** when a process **requests a resource**.
- ▶ **Request edge** converted to an **assignment edge** when the **resource is allocated** to the process.



## Resource-Allocation Graph Scheme

- ▶ **Claim edge**  $P_i \rightarrow R_j$ : indicates that process  $P_i$  may **request** resource  $R_j$ ; represented by a dashed line
- ▶ **Claim edge** converts to **request edge** when a process **requests a resource**.
- ▶ **Request edge** converted to an **assignment edge** when the **resource is allocated** to the process.
- ▶ When a resource is **released** by a process, **assignment edge** reconverts to a **claim edge**.

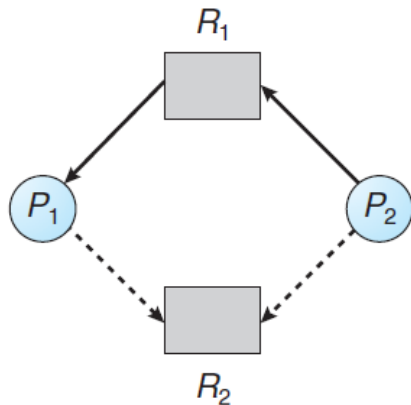


## Resource-Allocation Graph Scheme

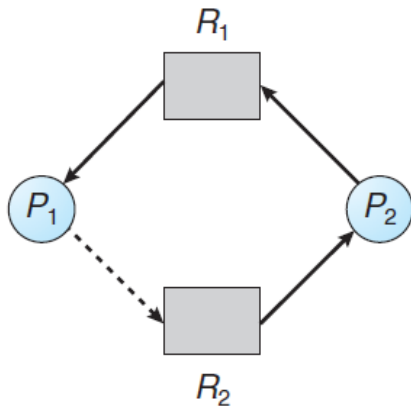
- ▶ **Claim edge**  $P_i \rightarrow R_j$ : indicates that process  $P_i$  may **request** resource  $R_j$ ; represented by a dashed line
- ▶ **Claim edge** converts to **request edge** when a process **requests a resource**.
- ▶ **Request edge** converted to an **assignment edge** when the **resource is allocated** to the process.
- ▶ When a resource is **released** by a process, **assignment edge** reconverts to a **claim edge**.
- ▶ Resources must be claimed **a priori** in the system.



# Resource-Allocation Graph



## Unsafe State In Resource-Allocation Graph





## Resource-Allocation Graph Algorithm

- ▶ 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** in the resource allocation graph.

# Banker's Algorithm



# Banker's Algorithm

- ▶ Multiple instances



# Banker's Algorithm

- ▶ Multiple instances
- ▶ Each process must a priori claim of the maximum use.



# Banker's Algorithm

- ▶ Multiple instances
- ▶ Each process must a priori claim of the maximum use.
- ▶ When a process requests a resource it may have to wait.



# Banker's Algorithm

- ▶ Multiple instances
- ▶ Each process must a priori claim of the maximum use.
- ▶ When a process requests a resource it may have to wait.
- ▶ 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, and  $m$  = number of resources types



## Data Structures for Banker's Algorithm

- ▶  $n$  = 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.



## Data Structures for Banker's Algorithm

- ▶  $n$  = 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*:  $n \times m$  matrix.
  - If  $Max[i, j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$ .



## Data Structures for Banker's Algorithm

- ▶  $n$  = 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*:  $n \times m$  matrix.
  - If  $Max[i, j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$ .
- ▶ *Allocation*:  $n \times m$  matrix.
  - If  $Allocation[i, j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$ .



## Data Structures for Banker's Algorithm

- ▶  $n$  = 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*:  $n \times m$  matrix.
  - If  $Max[i, j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$ .
- ▶ *Allocation*:  $n \times m$  matrix.
  - If  $Allocation[i, j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$ .
- ▶ *Need*:  $n \times m$  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  $i = 0, 1, \dots, n - 1$



## Safety Algorithm

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

Initialize:

*Work* = *Available*

*Finish*[ $i$ ] = *false* for  $i = 0, 1, \dots, n - 1$

2. Find an  $i$  such that both:

1. *Finish*[ $i$ ] = *false*

2.  $Need_i \leq Work$

If no such  $i$  exists, go to step 4.



## Safety Algorithm

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

Initialize:

*Work* = *Available*

*Finish*[ $i$ ] = *false* for  $i = 0, 1, \dots, n - 1$

2. Find an  $i$  such that both:

1. *Finish*[ $i$ ] = *false*

2.  $Need_i \leq Work$

If no such  $i$  exists, go to step 4.

3. *Work* = *Work* + *Allocation* <sub>$i$</sub>

*Finish*[ $i$ ] = *true*

Go to step 2





## Safety Algorithm

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

Initialize:

*Work* = *Available*

*Finish*[ $i$ ] = *false* for  $i = 0, 1, \dots, n - 1$

2. Find an  $i$  such that both:
  1. *Finish*[ $i$ ] = *false*
  2.  $Need_i \leq Work$If no such  $i$  exists, go to step 4.

3. *Work* = *Work* + *Allocation* <sub>$i$</sub>   
*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 $P_i$ (1/2)

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



## Resource-Request Algorithm for Process $P_i$ (1/2)

- ▶  $Request_i$  = request vector for process  $P_i$ . If  $Request_i[j] = k$ , then process  $P_i$  wants  $k$  instances of resource type  $R_j$ .
- ▶ 1. If  $Request_i \leq Need_i$ , go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.



## Resource-Request Algorithm for Process $P_i$ (1/2)

- ▶  $Request_i$  = request vector for process  $P_i$ . If  $Request_i[j] = k$ , then process  $P_i$  wants  $k$  instances of resource type  $R_j$ .
- ▶ 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 $P_i$ (2/2)

- ▶ 3. Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:

$$\text{Available} = \text{Available} - \text{Request};$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request};$$

$$\text{Need}_i = \text{Need}_i - \text{Request};$$



## Resource-Request Algorithm for Process $P_i$ (2/2)

- ▶ 3. Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request;$$

$$Need_i = Need_i - Request;$$

- If **safe**: the resources are allocated to  $P_i$



## Resource-Request Algorithm for Process $P_i$ (2/2)

- ▶ 3. Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:

$Available = Available - Request;$

$Allocation_i = Allocation_i + Request;$

$Need_i = Need_i - Request;$

- If **safe**: the resources are allocated to  $P_i$
- If **unsafe**:  $P_i$  must wait, and the old resource-allocation state is restored



## Banker's Algorithm Example (1/3)

- ▶ 5 processes:  $P_0$  through  $P_4$
- ▶ 3 resource types:
  - $A$  (10 instances),  $B$  (5 instances), and  $C$  (7 instances)
- ▶ Snapshot at time  $T_0$

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$
$P_0$	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 Example (2/3)

- ▶ The content of the matrix *Need* is defined to be *Max* – *Allocation*

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>		<u>Need</u>
	A B C	A B C	A B C		A B C
$P_0$	0 1 0	7 5 3	3 3 2	$P_0$	7 4 3
$P_1$	2 0 0	3 2 2		$P_1$	1 2 2
$P_2$	3 0 2	9 0 2		$P_2$	6 0 0
$P_3$	2 1 1	2 2 2		$P_3$	0 1 1
$P_4$	0 0 2	4 3 3		$P_4$	4 3 1

## Banker's Algorithm Example (2/3)

- ▶ The content of the matrix *Need* is defined to be  $Max - Allocation$

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>		<u>Need</u>
	A B C	A B C	A B C		A B C
$P_0$	0 1 0	7 5 3	3 3 2	$P_0$	7 4 3
$P_1$	2 0 0	3 2 2		$P_1$	1 2 2
$P_2$	3 0 2	9 0 2		$P_2$	6 0 0
$P_3$	2 1 1	2 2 2		$P_3$	0 1 1
$P_4$	0 0 2	4 3 3		$P_4$	4 3 1

- ▶ Is the system safe?

## Banker's Algorithm Example (2/3)

- ▶ The content of the matrix *Need* is defined to be *Max* – *Allocation*

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>		<u>Need</u>
	A B C	A B C	A B C		A B C
$P_0$	0 1 0	7 5 3	3 3 2	$P_0$	7 4 3
$P_1$	2 0 0	3 2 2		$P_1$	1 2 2
$P_2$	3 0 2	9 0 2		$P_2$	6 0 0
$P_3$	2 1 1	2 2 2		$P_3$	0 1 1
$P_4$	0 0 2	4 3 3		$P_4$	4 3 1

- ▶ Is the system safe?  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  satisfies safety criteria.



## Banker's Algorithm Example (3/3)

- ▶  $P_1$  Request  $(1, 0, 2)$
- ▶ Check that  $Request \leq Available$ :  $(1, 0, 2) \leq (3, 3, 2) \Rightarrow true$



## Banker's Algorithm Example (3/3)

- ▶  $P_1$  Request  $(1, 0, 2)$
- ▶ Check that  $Request \leq Available$ :  $(1, 0, 2) \leq (3, 3, 2) \Rightarrow true$

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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 2	6 0 0	
$P_3$	2 1 1	0 1 1	
$P_4$	0 0 2	4 3 1	

## Banker's Algorithm Example (3/3)

- ▶  $P_1$  Request  $(1, 0, 2)$
- ▶ Check that  $Request \leq Available$ :  $(1, 0, 2) \leq (3, 3, 2) \Rightarrow true$

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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 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  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  satisfies safety requirement.

## Banker's Algorithm Example (3/3)

- ▶  $P_1$  Request  $(1, 0, 2)$
- ▶ Check that  $Request \leq Available$ :  $(1, 0, 2) \leq (3, 3, 2) \Rightarrow true$

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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 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  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  satisfies safety requirement.
- ▶ Can request for  $(3, 3, 0)$  by  $P_4$  be granted?

## Banker's Algorithm Example (3/3)

- ▶  $P_1$  Request  $(1, 0, 2)$
- ▶ Check that  $Request \leq Available$ :  $(1, 0, 2) \leq (3, 3, 2) \Rightarrow true$

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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 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  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  satisfies safety requirement.
- ▶ Can request for  $(3, 3, 0)$  by  $P_4$  be granted?
- ▶ Can request for  $(0, 2, 0)$  by  $P_0$  be granted?



# Deadlock Detection



# 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  $P_j$ .



## Single Instance of Each Resource Type

- ▶ Maintain **wait-for** graph.
  - **Nodes** are **processes**.
  - $P_i \rightarrow P_j$  if  $P_i$  is **waiting** for  $P_j$ .
  
- ▶ Periodically invoke an algorithm that searches for a **cycle in the graph**.



## Single Instance of Each Resource Type

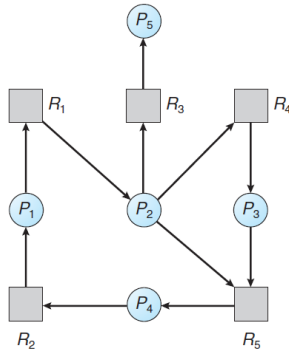
- ▶ Maintain **wait-for** graph.
  - **Nodes** are **processes**.
  - $P_i \rightarrow P_j$  if  $P_i$  is **waiting** for  $P_j$ .
- ▶ Periodically invoke an algorithm that searches for a **cycle in the graph**.
- ▶ If there is a **cycle**, there exists a **deadlock**.



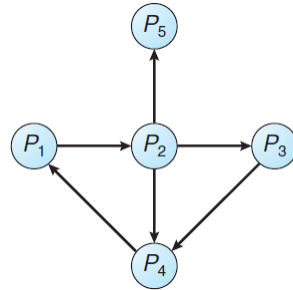
## Single Instance of Each Resource Type

- ▶ Maintain **wait-for** graph.
  - **Nodes** are **processes**.
  - $P_i \rightarrow P_j$  if  $P_i$  is **waiting** for  $P_j$ .
- ▶ 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  $O(n^2)$  operations, where  $n$  is the number of **vertices** in the graph.

# Resource-Allocation Graph and Wait-for Graph



Resource-allocation graph



Corresponding Wait-for graph



## Data Structures for Deadlock Detection

- ▶ *Available*: vector of length  $m$ , indicates the number of **available resources** of each type.





## Data Structures for Deadlock Detection

- ▶ *Available*: vector of length  $m$ , indicates the number of **available resources** of each type.
- ▶ *Allocation*:  $n \times m$  matrix, defines the **number of resources** of each type currently **allocated** to each process.



## Data Structures for Deadlock Detection

- ▶ *Available*: vector of length  $m$ , indicates the number of **available resources** of each type.
- ▶ *Allocation*:  $n \times m$  matrix, defines the **number of resources** of each type currently **allocated** to each process.
- ▶ *Request*:  $n \times m$  matrix, indicates the **current request** of each process.
  - If  $Request[i, j] = k$ , then  $P_i$  **requesting**  $k$  more instances of resource type  $R_j$ .



## Detection Algorithm (1/2)

- ▶ 1. Let *Work* and *Finish* be vectors of length  $m$  and  $n$ , respectively.  
Initialize:
  - a. *Work* = *Available*
  - b. For  $i = 1, 2, \dots, n$ , if *Allocation<sub>i</sub>*  $\neq 0$ , then *Finish*[ $i$ ] = *false*;  
otherwise, *Finish*[ $i$ ] = *true*



## Detection Algorithm (1/2)

- ▶ 1. Let *Work* and *Finish* be vectors of length  $m$  and  $n$ , respectively. Initialize:
  - a.  $Work = Available$
  - b. For  $i = 1, 2, \dots, n$ , 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.  $Request_i \leq Work$



## Detection Algorithm (1/2)

- ▶ 1. Let *Work* and *Finish* be vectors of length  $m$  and  $n$ , respectively. Initialize:
  - a.  $Work = Available$
  - b. For  $i = 1, 2, \dots, n$ , 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 (2/2)

- ▶ 3.  $Work = Work + Allocation;$   
 $Finish[i] = true$   
go to step 2



## Detection Algorithm (2/2)

- ▶ 3.  $Work = Work + Allocation;$   
 $Finish[i] = true$   
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**.



## Detection Algorithm (2/2)

- ▶ 3.  $Work = Work + Allocation;$   
 $Finish[i] = true$   
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  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state.





## Detection Algorithm Example (1/2)

- ▶ 5 processes:  $P_0$  through  $P_4$
- ▶ 3 resource types:
  - $A$  (7 instances),  $B$  (2 instances), and  $C$  (6 instances)
- ▶ Snapshot at time  $T_0$

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	$A B C$	$A B C$	$A B C$
$P_0$	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	



## Detection Algorithm Example (1/2)

- ▶ 5 processes:  $P_0$  through  $P_4$
- ▶ 3 resource types:
  - $A$  (7 instances),  $B$  (2 instances), and  $C$  (6 instances)
- ▶ Snapshot at time  $T_0$

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	$A B C$	$A B C$	$A B C$
$P_0$	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	

- ▶ **Deadlock?**



## Detection Algorithm Example (1/2)

- ▶ 5 processes:  $P_0$  through  $P_4$
- ▶ 3 resource types:
  - $A$  (7 instances),  $B$  (2 instances), and  $C$  (6 instances)
- ▶ Snapshot at time  $T_0$

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$
$P_0$	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	

- ▶ **Deadlock?** Sequence  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  will result in  $Finish[i] = true$  for all  $i$



## Detection Algorithm Example (2/2)

- ▶  $P_2$  requests an additional instance of type  $C$

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>		<u>Request</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>		<i>A B C</i>
$P_0$	0 1 0	0 0 0	0 0 0	$P_0$	0 0 0
$P_1$	2 0 0	2 0 2		$P_1$	2 0 2
$P_2$	3 0 3	0 0 0		$P_2$	0 0 1
$P_3$	2 1 1	1 0 0		$P_3$	1 0 0
$P_4$	0 0 2	0 0 2		$P_4$	0 0 2

## Detection Algorithm Example (2/2)

- ▶  $P_2$  requests an additional instance of type  $C$

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>		<u>Request</u>
	$A B C$	$A B C$	$A B C$		$A B C$
$P_0$	0 1 0	0 0 0	0 0 0	$P_0$	0 0 0
$P_1$	2 0 0	2 0 2		$P_1$	2 0 2
$P_2$	3 0 3	0 0 0		$P_2$	0 0 1
$P_3$	2 1 1	1 0 0		$P_3$	1 0 0
$P_4$	0 0 2	0 0 2		$P_4$	0 0 2

- ▶ Can reclaim resources held by process  $P_0$ , but **insufficient resources** to fulfill other processes; requests

## Detection Algorithm Example (2/2)

- ▶  $P_2$  requests an additional instance of type  $C$

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>		<u>Request</u>
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$		$A\ B\ C$
$P_0$	0 1 0	0 0 0	0 0 0	$P_0$	0 0 0
$P_1$	2 0 0	2 0 2		$P_1$	2 0 2
$P_2$	3 0 3	0 0 0		$P_2$	0 0 1
$P_3$	2 1 1	1 0 0		$P_3$	1 0 0
$P_4$	0 0 2	0 0 2		$P_4$	0 0 2

- ▶ 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



# Recovery from Deadlock

- ▶ Process **termination**
- ▶ Resource **preemption**





## Process Termination

- ▶ Abort all **deadlocked** processes.



## Process Termination

- ▶ Abort **all** **deadlocked** processes.
- ▶ Abort **one process at a time** until the deadlock cycle is eliminated



## Process Termination

- ▶ Abort **all deadlocked** processes.
- ▶ Abort **one process at a time** until the deadlock cycle is eliminated
- ▶ In which order should we choose to abort?



## Process Termination

- ▶ Abort **all deadlocked** processes.
- ▶ Abort **one process at a time** until the deadlock cycle is eliminated
- ▶ In which order should we choose to abort?
  1. Priority of the process.
  2. How long process has computed, and how much longer to completion.
  3. Resources the process has used.
  4. Resources process needs to complete.
  5. How many processes will need to be terminated.



# Resource Preemption

- ▶ Selecting a **victim**: minimize cost



# Resource Preemption

- ▶ Selecting a **victim**: minimize cost
- ▶ **Rollback**: return to some **safe state**, restart process for that state.



# 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.

# Summary





## Summary

- ▶ Deadlock



## Summary

- ▶ Deadlock
- ▶ Four simultaneous conditions: mutual exclusion, hold and wait, no pre-emption, circular wait



## Summary

- ▶ Deadlock
- ▶ Four simultaneous conditions: mutual exclusion, hold and wait, no pre-emption, circular wait
- ▶ Deadlock prevention



## Summary

- ▶ Deadlock
- ▶ Four simultaneous conditions: mutual exclusion, hold and wait, no pre-emption, circular wait
- ▶ Deadlock prevention
- ▶ Deadlock avoidance: resource-allocation algorithm, banker's algorithm



## Summary

- ▶ Deadlock
- ▶ Four simultaneous conditions: mutual exclusion, hold and wait, no pre-emption, circular wait
- ▶ Deadlock prevention
- ▶ Deadlock avoidance: resource-allocation algorithm, banker's algorithm
- ▶ Deadlock detection: Wait-for graph



## Summary

- ▶ Deadlock
- ▶ Four simultaneous conditions: mutual exclusion, hold and wait, no pre-emption, circular wait
- ▶ Deadlock prevention
- ▶ Deadlock avoidance: resource-allocation algorithm, banker's algorithm
- ▶ Deadlock detection: Wait-for graph
- ▶ Deadlock recovery: process termination, resource preemption

# Questions?

## Acknowledgements

Some slides were derived from Avi Silberschatz slides.