

Operating Systems: CSE 3204

Chapter Two Process Management

Lecture 6: Deadlocks

November 11, 2025

1

Deadlock

- System Model
- Deadlock Characterization
- Methods for handling Deadlock
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

System Model

- A system contains a finite number of resource types ($R_1, R_2, \dots, R_m$) to be distributed among competing processes
- The resource types are partitioned in to several types (e.g files, I/O devices, CPU cycles, memory), each having a number of identical instances
- A process must request a resource before using it and release it after making use of it. Each process utilizes a resource as follows:
 - **Request**
 - A process requests for an instance of a resource type. If the resource is free, the request will be granted. Otherwise the process should wait until it acquires the resource
 - **Use**
 - The process uses the resource for its operations
 - **Release**
 - The process releases the resource

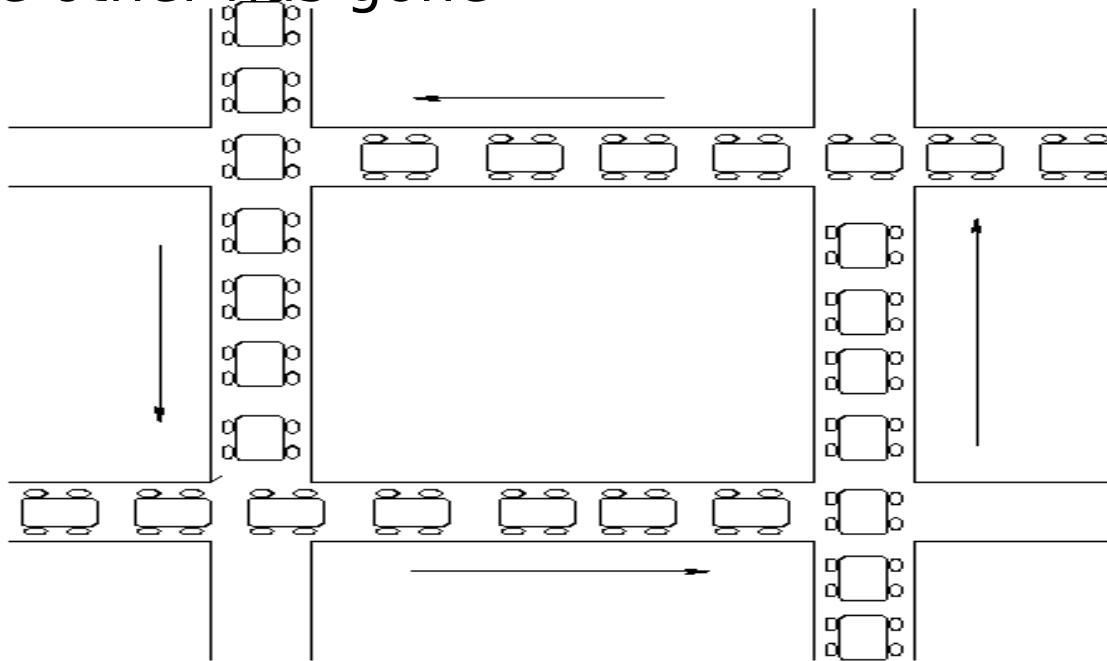
Deadlock

- **Deadlock** can be defined as a permanent blocking of processes that either compete for system resources or communicate with each other
- The set of blocked processes each hold a resource and wait to acquire a resource held by another process in the set
- All deadlocks involve conflicting needs for resources by two or more processes
- A set of processes or threads are deadlocked when each process or thread is waiting for a resource to be freed which is controlled by another process
- **Example 1**
 - Suppose a system has 2 disk drives
 - If P_1 is holding disk 2 and P_2 is holding disk 1 and if P_1 requests for disk 1 and P_2 requests for disk 2, then

November 11, 2025 deadlock occurs

Traffic gridlock is an everyday example of a deadlock situation.

- When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone



Deadlock characterization

Deadlock can arise **if four conditions hold simultaneously in a system:**

1. **Mutual exclusion:** only one process at a time can use a resource. No process can access a resource unit that has been allocated to another process
2. **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
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 a resource that is held by $P_2, \dots$,
 - P_{n-1} is waiting for a resource that is held by P_n , and
 - P_n is waiting for a resource that is held by P_0

Resource Allocation Graph(RAG)

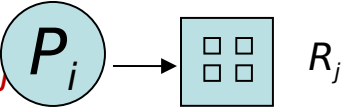
- Deadlock can be better described by using a **directed graph** called **resource allocation graph**
- The graph consists of a set of vertices V and a set of edges E
 V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.
- **Request edge:** directed edge $P_i \rightarrow R_j$
- **Assignment edge:** directed edge $R_j \rightarrow P_i$
- If a Resource Allocation Graph contains a cycle, then a deadlock may exist

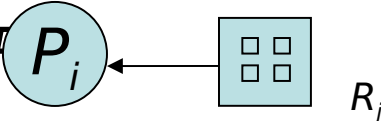
Resource Allocation Graph /RAG (cont.)

- Diagrammatically, processes and resources in RAG are represented as follow:

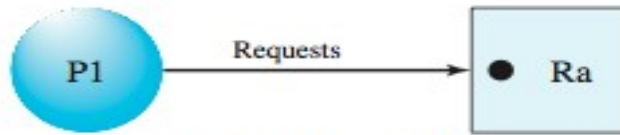
- Process 

- Resource Type with 4 instances 

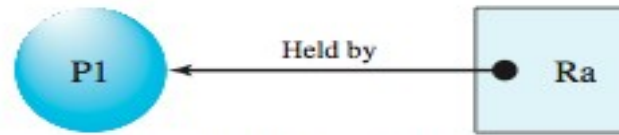
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

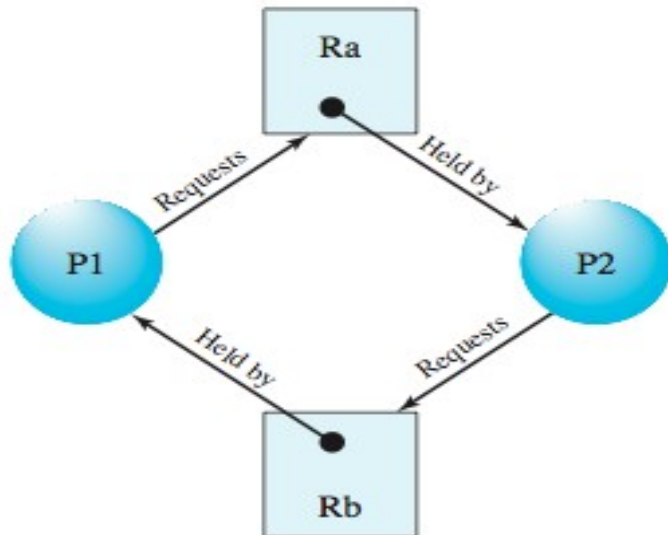
Example of Resource Allocation Graph



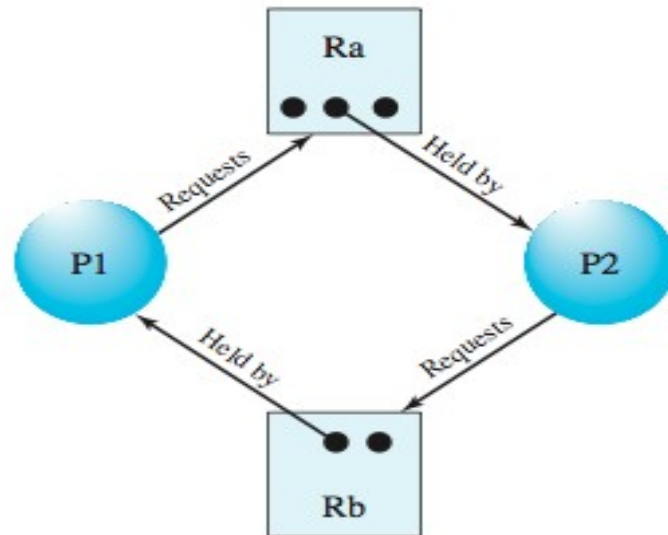
(a) Resource is requested



(b) Resource is held

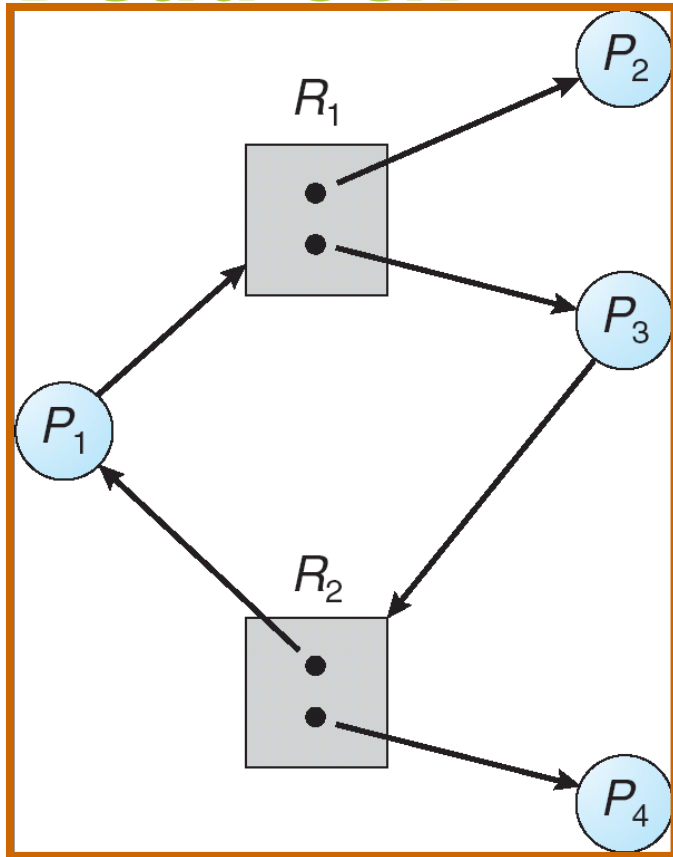


(c) Circular wait



(d) No deadlock

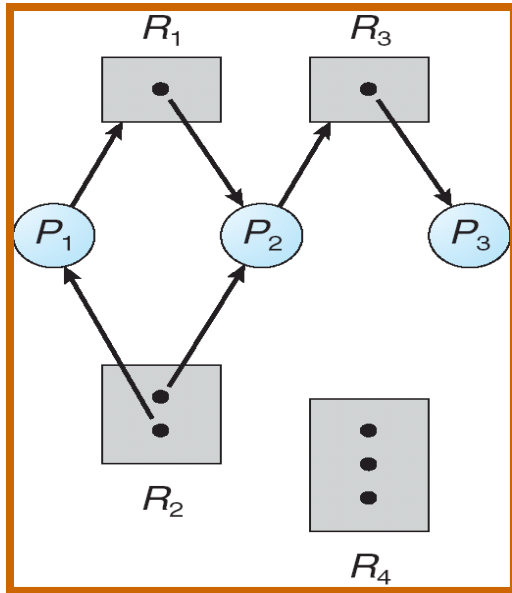
Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles then there's no deadlock
- If however a graph contains a cycle then there are two possible situations:
 - if there is only one instance per resource type, then deadlock can occur
 - if there are several instances per resource type, there's a possibility of no deadlock

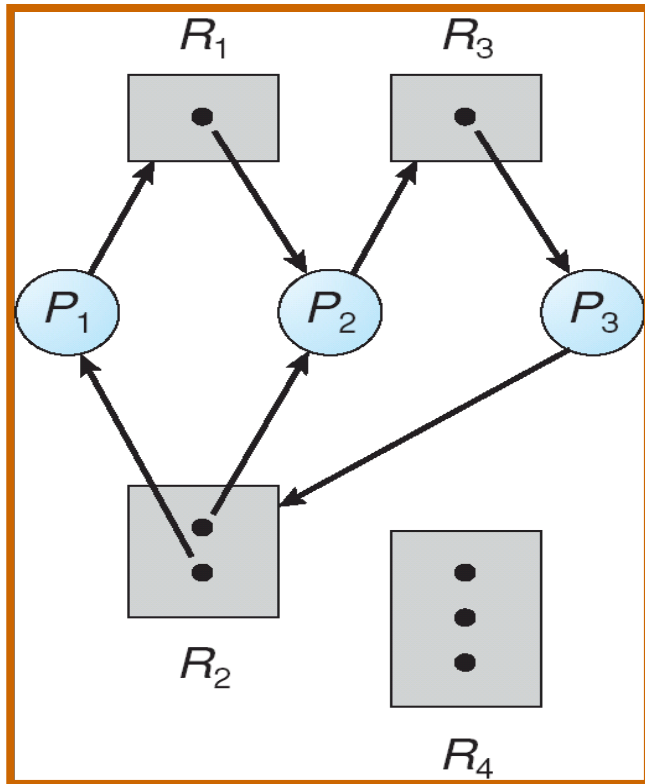
Example of Resource Allocation Graph



- The RAG shown here tells us about the following situation in a system:
 - $P = \{P_1, P_2, P_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$
 - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- **The process states**
 - P_1 is holding an instance of R_2 and is waiting for an instance of R_1
 - P_2 is holding an instance of R_1 and instance of R_2 , and is waiting for an instance of R_3
 - P_3 is holding an instance of R_3

Example of Resource Allocation Graph

• Resource Allocation Graph With a Deadlock



- There are two cycles in this graph
 - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
 - $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
- Processes P_1 , P_2 and P_3 are **deadlocked**:
- P_1 is waiting for P_2 to release R_1
- P_2 is waiting for R_3 held by P_3 and
- P_3 is waiting for either P_1 or P_2 to release R_2

Methods for handling Deadlocks

Deadlock problems can be handled in one of the following 3 ways:

1. Using a protocol that prevents or avoids deadlock by ensuring that a system will *never* enter a deadlock state **deadlock prevention** and **deadlock avoidance** scheme are used
2. Allow the system to enter a **deadlock state and then recover**
3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

- By ensuring at least one of the necessary conditions for deadlock will not hold, deadlock can be prevented.
 - This is mainly done by restraining how requests for resources can be made
- **Deadlock prevention** methods fall into two classes:
 - An **indirect** method of deadlock prevention prevents the occurrence of one of the three necessary conditions listed previously (items 1 through 3)
 - A **direct** method of deadlock prevention prevents the occurrence of a circular wait (item 4)

Deadlock Prevention (contd.)

- 1. Mutual Exclusion:** This is **not required for sharable resources**; however to prevent a system from deadlock, the mutual exclusion condition **must hold for non-sharable resources**
- 2. Hold and Wait:** in order to prevent the occurrence of this condition in a system, we must guarantee that whenever a process requests a resource, **it does not hold any other resources. Two protocols are used to implement this:**
 1. Require a process to request and be allocated all its resources before it begins execution or
 2. Allow a process to request resources only when the **process has none**
 - Both protocols have **two main disadvantages:**
 - o Since resources may be allocated but not used for a long period, **resource utilization will be low**
 - o A process that needs several popular resources has to wait indefinitely because one of the resources it needs is allocated to another process. Hence **starvation is possible.**

Deadlock Prevention (contd.)

3. No Preemption

- If a process holding certain resources is denied further request, that process **must release its original resources allocated to it**
- If a process requests a resource allocated to another process waiting for some additional resources, and the requested resource is not being used, then the resource will be **preempted from the waiting process and allocated to the requesting process**
- 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
- This approach is practical to resources whose state can easily saved and retrieved easily

Deadlock Prevention (contd.)

4. Circular Wait

- A linear ordering of all resource types is defined and each process requests resources **in an increasing order of enumeration**
- So, if a process initially is allocated instances of resource type R , then it can subsequently request instances of resources types following R in the ordering

$$R_i \rightarrow R_{i+1} \rightarrow R_{i+2} \rightarrow R_{i+3}$$

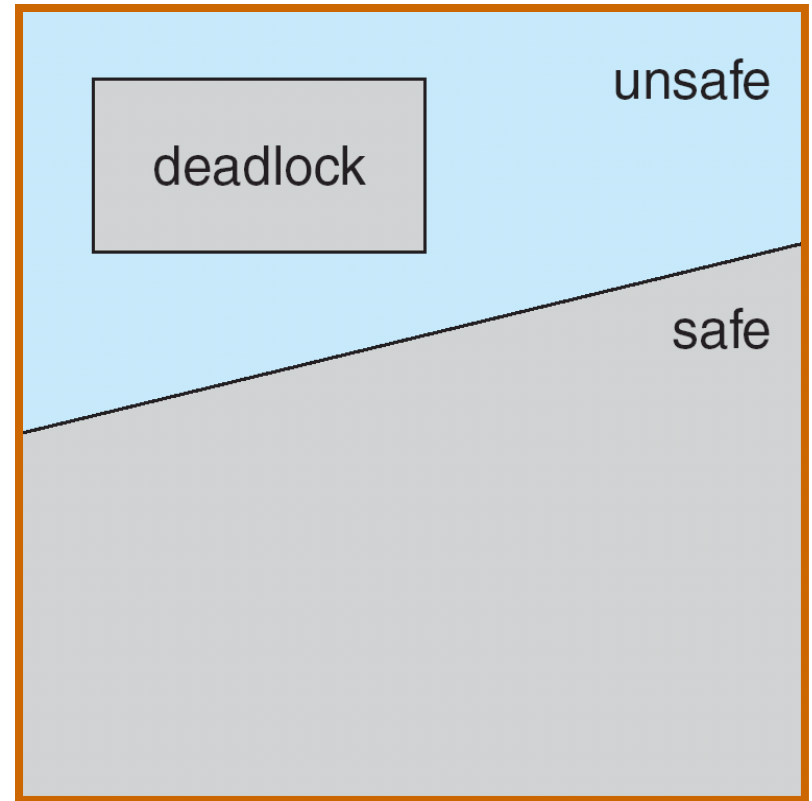
Deadlock Avoidance

- **Deadlock avoidance** scheme requires each process to declare the **maximum number of resources of each type that it may need in advance**
 - Having this full information about the sequence of requests and release of resources, we can know whether or not the system is entering **unsafe** state
 - The **deadlock-avoidance** algorithm **dynamically examines the resource-allocation state** to ensure that there can never be a **circular-wait condition**
 - Resource-allocation state is defined by **the number of available** and **allocated resources**, and **the maximum demands** of the processes
 - **A state is safe** if the system can allocate resources to each process in some order avoiding a deadlock.
 - **A deadlock state is an unsafe state.**

Deadlock avoidance: Safe State

Basic Facts

- If a system is in a **safe state**, then there are **no deadlocks**.
- If a system is **in unsafe state**, then there is a **possibility of deadlock**.
- **Deadlock avoidance** method ensures that a system will never enter **an unsafe state**.



Deadlock Avoidance Algorithms

- Based on the concept of **safe state**, we can define algorithms that ensures the system **will never deadlock**.
- If there **is a single instance** of a resource type,
 - Use a **resource-allocation graph**(RAG)
- If there are **multiple instances** of a resource type,
 - Use the Dijkstra's **banker's algorithm**

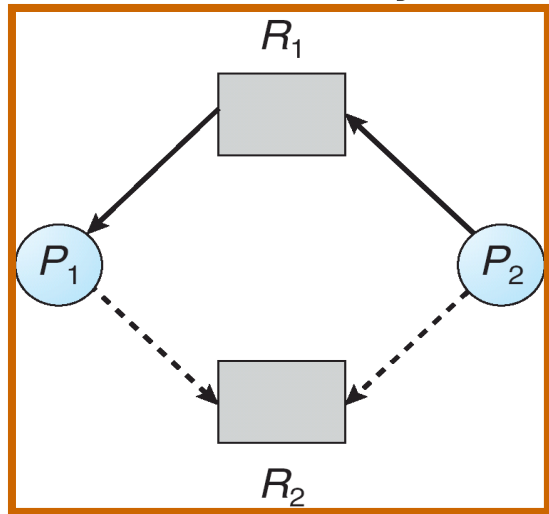
Deadlock Avoidance Algorithms (contd.)

Resource-Allocation Graph Scheme

- A new type of edge (*Claim edge*), in addition to the request and assignment edge is introduced.
- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some point in the future. The edge resembles a request edge but is represented by a dashed line in the graph
 - Claim edge is converted to request edge when a process requests a resource
 - Request edge is 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
 - If no cycle exists in the allocation, then system is in safe state otherwise the system is in unsafe state
- Resources must be claimed *a priori* in the system
 - i.e, before a process starts executing, all its claim edge must show up in the allocation graph

Deadlock Avoidance Algorithms (contd.)

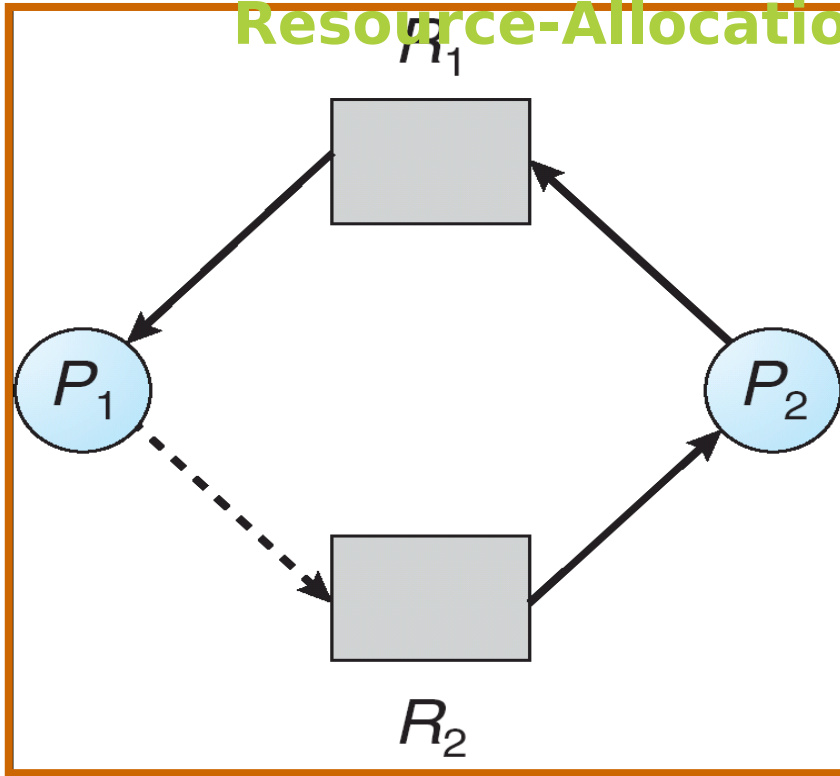
- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.



- If we suppose P_2 requests R_2 . We can not allocate it since it will create a cycle.

Deadlock Avoidance Algorithms (contd.)

Resource-Allocation Graph algorithm



- If P_1 requests for R_2 and P_2 requests for R_1 , then deadlock will occur

Unsafe State Resource-Allocation

Deadlock Avoidance Algorithms (contd.)

Banker's Algorithm

- This algorithm is used when there are multiple instances of resources
- When a process enters a system, it must declare the maximum number of each instance of resource types it may need
 - The number however may not exceed the total number of resource types in the system
- 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

November 11, 2025

24

Deadlock Avoidance Algorithms (contd.)

Banker's Algorithm

- The following data structures are used in the 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**: $n \times m$ 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

Deadlock Avoidance Algorithms (contd.)

Banker's Algorithm

The algorithm is as follows:

1. Process p_i makes requests for resources. Let $\text{Request}(i)$ be the corresponding request vector. So, if p_i wants k instances of resource type r_j , then $\text{Request}(i)[j] = k$
2. If $\text{Request}(i) \not\leq \text{Need}(i)$, there is an error.
3. Otherwise, if $\text{Request}(i) \not\leq \text{Available}$, then p_i must wait
4. Otherwise, Modify the data structures as follows :
 - $\text{Available} = \text{Available} - \text{Request}(i)$
 - $\text{Allocation}(i) = \text{Allocation}(i) + \text{Request}(i)$
 - $\text{Need}(i) = \text{Need}(i) - \text{Request}(i)$
5. Check whether the resulting state is safe. (Use the safety algorithm presented in the next slide)
6. If the state is safe, do the allocation. Otherwise, p_i must wait for $\text{Request}(i)$

Deadlock Avoidance Algorithms (contd.)

Safety Algorithm

- it is used to identify whether or not a system is in a safe state. The algorithm can be described as follow:
 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 and i such that both:
 - (a) $Finish[i] = false$
 - (b) $Need_i \leq Work$If no such i exists, go to step 4.
 3. $Work = Work + Allocation_i$
 $Finish[i] = true$
go to step 2.
 4. If $Finish[i] == true$ for all i , then the system is in a safe state.

Deadlock Avoidance Algorithms (contd.)

Resource-Request Algorithm

- This algorithm determines if a request can be safely granted. The algorithm is described below
- *If Request = request vector for process P_i . If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j .*

1. *If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.*
2. *If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.*
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

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

- *If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored*

Deadlock Avoidance Algorithms (contd.)

Examples of Banker's algorithm:

Banker's Algorithm

- Assume a system has
 - 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			

Deadlock Avoidance Algorithms (contd.)

Example (contd.) Banker's Algorithm

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

	<u>Need</u>		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria

Deadlock Avoidance Algorithms (contd.)

Example: P_1 Request (1,0,2) Banker's Algorithm

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ 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	1	6	0	0	
P_3	2	1	1	0	1	1	
P_4	0	0	2	4	3	1	

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ 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

- If a system does not implement either deadlock prevention or avoidance, deadlock may occur. Hence the system must provide
 - A deadlock detection algorithm that examines the state of the system if there is an occurrence of deadlock
 - An algorithm to recover from the deadlock

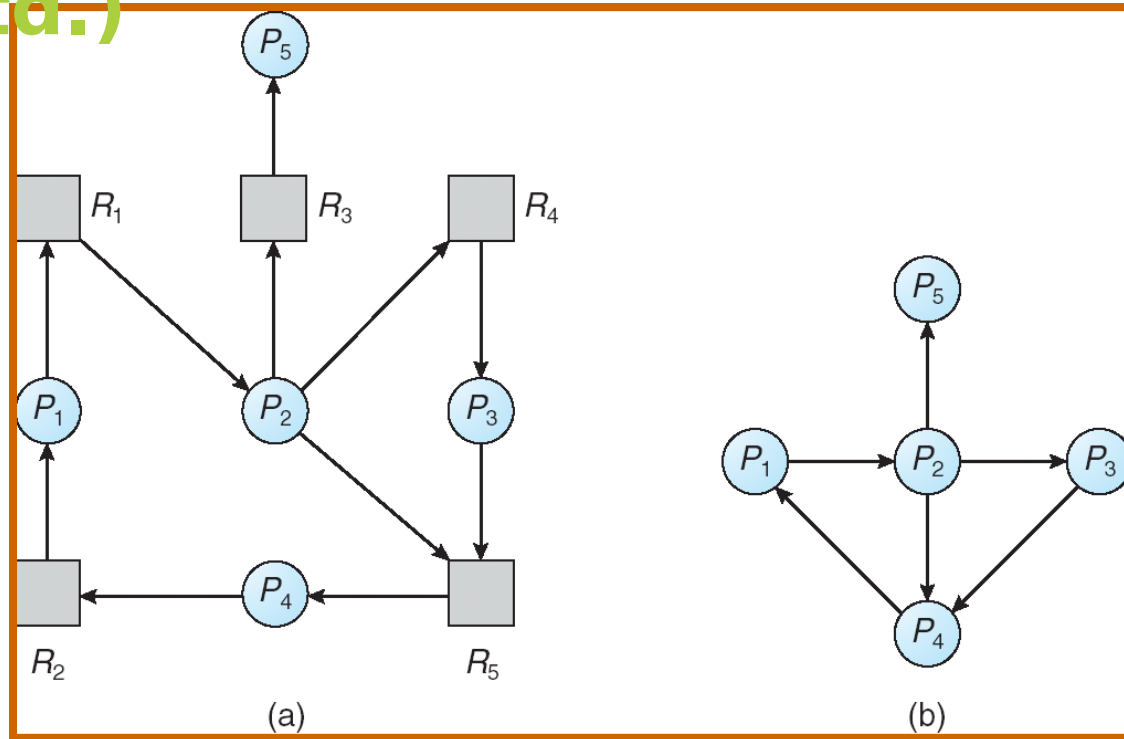
Deadlock Detection:

1. Single Instance of Each Resource Type

- If there are single instances of each resources in a system, then an algorithm that uses a type of resource allocation graph called *wait-for* graph will be used
- The *wait-for* graph is obtained from the resource allocation graph by removing the resource nodes and collapsing the corresponding edges
- If a process P_i points to P_j in a *wait-for* graph, it indicates that P_i is waiting for P_j to release a resource P_j needs
- To detect deadlocks, the system needs to 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.

Deadlock Detection:

1. Single Instance of Each Resource Type (contd.)



Resource-Allocation Graph Corresponding wait-for graph

Deadlock Detection:

2. Several Instances of a Resource Type

- When there are multiple instances of a resource type in a resource allocation system, the wait-for graph is not applicable. Hence, a deadlock detection algorithm is used
- The algorithm uses several data structures similar to the ones in banker's algorithm
- **Available:** A vector of length m indicates the 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. If $Request[i_j] = k$, then process P_i is requesting k more instances of resource type R_j .

Deadlock Detection Algorithm

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.
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**.

Deadlock Detection Algorithm (contd.)

Example

- Processes P_0 through P_4 ; three 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			

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

Deadlock Detection Algorithm (cont.)

- **Example** an additional instance of type C.

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

- **State of system?**
 - Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes' requests.
 - Deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4 .

Deadlock Detection Algorithm

Usage

- When, and how often, to invoke the detection algorithm depends on:
 1. How often a deadlock is likely to occur?
 2. How many processes will be affected (need to be rolled back)?
- If deadlock occurs frequently, then the algorithm is invoked frequently,
 - o Resources allocated to deadlocked processes will be idle until the deadlock can be broken
 - o The number of processes in the deadlock may increase
 - If the algorithm is invoked for every resource request not granted, it will incur a computation time overhead on the system
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock
- A less expensive alternative is to invoke the algorithm at a less frequent time
 - (e.g once per hour)

Recovery from Deadlock

- Once a deadlock has been detected, recovery strategy is needed. There are two possible recovery approaches:
 - Process termination
 - Resource preemption

Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose a process to abort? Chose the process with
 - Least amount of processor time consumed so far
 - Least amount of output produced so far
 - Most estimated time remaining
 - Least total resources allocated so far
 - Lowest priority

Recovery from Deadlock

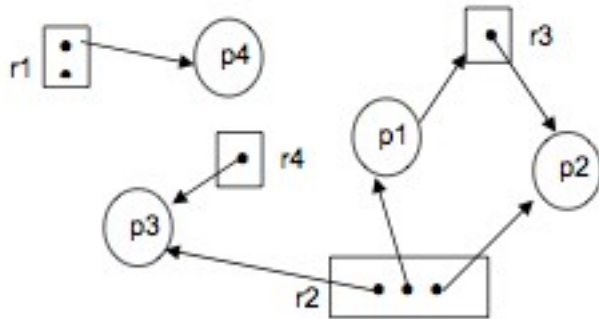
Resource Preemption

- In this recovery strategy, we successively preempt resources and allocate them to another process until the deadlock is broken
- While implementing this strategy, there are three issues to be considered
 - **Selecting a victim** – which resources and process should be selected to minimize cost just like in process termination. The cost factors may include parameters like the number of resources a deadlocked process is holding, number of resources it used so far
 - **Rollback** – if a resource is preempted from a process, then it can not continue its normal execution
 - The process must be rolled back to some safe state and started
 - **Starvation** – same process may always be picked as victim several times. As a result, starvation may occur. The best solution to this problem is to only allow a process to be picked as a victim for a limited finite number of times. This can be done by including the number of rollback in the cost factor

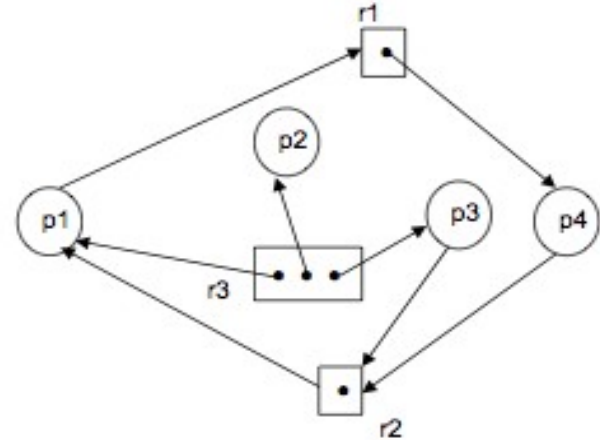
Exercises

- For each of the resource allocation graphs below, determine whether there is a deadlock or not. Give explanations for each of your answers.

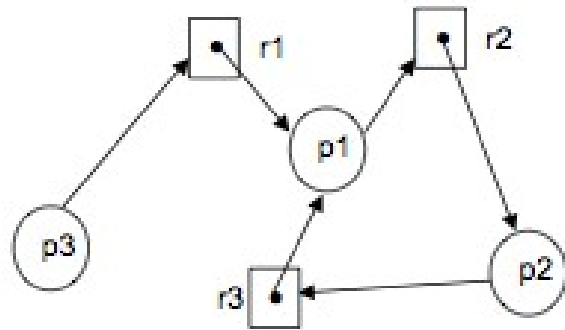
a.



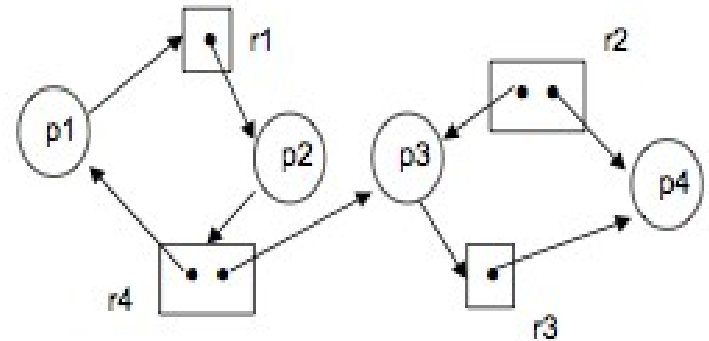
b.



c.



d.



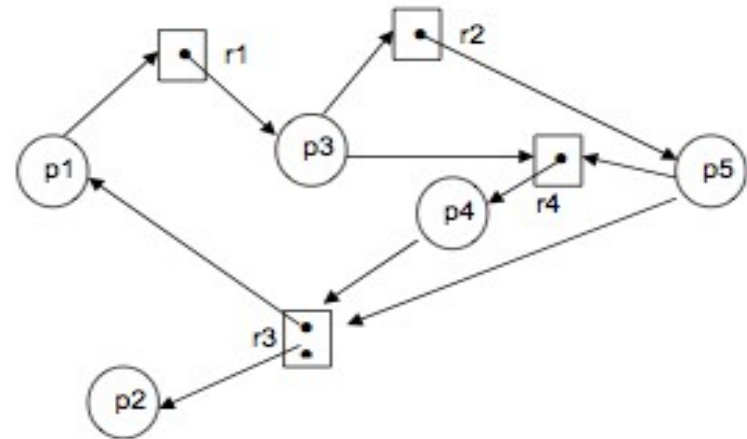
Exercises(cont...)

2. There are four processes in a system and they are going to share nine tape drives. Their current and maximum number of allocation numbers are as follows :

Processes	Allocation	Maximum
P1	3	6
P2	1	2
P3	4	9
P4	0	2

- Is the system in a safe state? Why or why not?
- Is the system deadlocked? Why or why not?

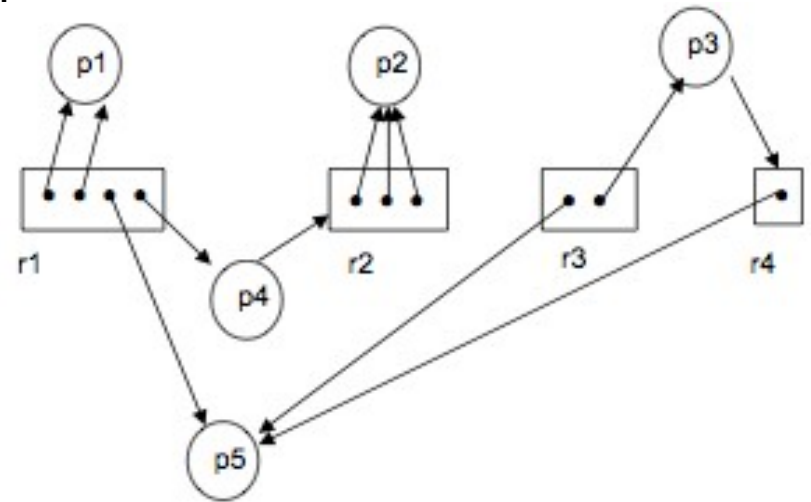
3. Explain if the system in the given resource allocation graph here is deadlocked. If not, give an execution order of the processes which successfully terminates



Exercises(cont...)

4. Given the resource allocation graph be'

- Apply the deadlock detection algorithm and either indicate why the system is deadlocked, or specify a safe allocation sequence
- If the process P2 also request 2 instances of resource r1, does the system enter a deadlock? Why?



5. Consider the following snapshot of a system:

Process	Allocation	Max	Available
	A B C D	A B C D	ABCD
P1	0 0 1 2	0 0 1 2	1 5 2 0
P2	1 0 0 0	1 7 5 0	
P3	1 3 5 4	2 3 5 6	
P4	0 6 3 2	0 6 5 2	
P5	0 0 1 4	0 6 5 6	

- Use the banker's algorithm to answer the following questions:
 - What is the content of the matrix Need?
 - Is the system in a safe state?
 - If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?