Operating Systems Concepts

Dead Locks (cont.)

CS 4375, Fall *2025*
Instructor: MD Armanuzzaman (***Arman***)
marmanuzzaman@utep.edu
November 12, 2025

Summery

- The Deadlock Problem
  - **Characterization of Deadlock**
  - **Resource Allocation Graph**
  - **Handling Deadlocks**
    - **Deadlock Prevention**
    - Deadlock Avoidance
    - Deadlock Detection
    - Deadlock Recovery

Agenda

- The Deadlock Problem
  - Characterization of Deadlock
  - Resource Allocation Graph
  - Handling Deadlocks
    - Deadlock Prevention
    - **Deadlock Avoidance**
    - **Deadlock Detection**
    - **Deadlock Recovery**

Deadlock Avoidance

- Deadlock Prevention: Prevents deadlocks by restraining resources and making sure **one of 4 necessary conditions for a deadlock does not hold**. (system design)
  - Possible side effect: low device utilization and reduced system throughput
- Deadlock Avoidance: Requires that the system has some additional a priori information available. (dynamic request check)
  - *E.g.* request disk and then printer
  - *E.g.* request at most *n* resources.
  - Allows more concurrency

- Similar to the difference between **a traffic light** and a **police officer** directing the traffic!

Deadlock Avoidance

- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- 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.

Deadlock Avoidance: Safe State

- A state is safe if the system can **allocate** resources to each process (up to its maximum) in some order and can still **avoid** a deadlock.
- When a process requests an available resource, system **must decide** if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a safe sequence of all processes.

Deadlock Avoidance: Safe State

- Sequence <P_1, P_2, …, P_n> is safe if for each P_i, the resources that P_i can still request, can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.
- If no such sequence exists, the state is unsafe!

Deadlock Avoidance: Safe State Example

- We have five processes P_1 through P_5, three resource types are available:

  A (7 instances), B (2 instances), and C (6 instances)

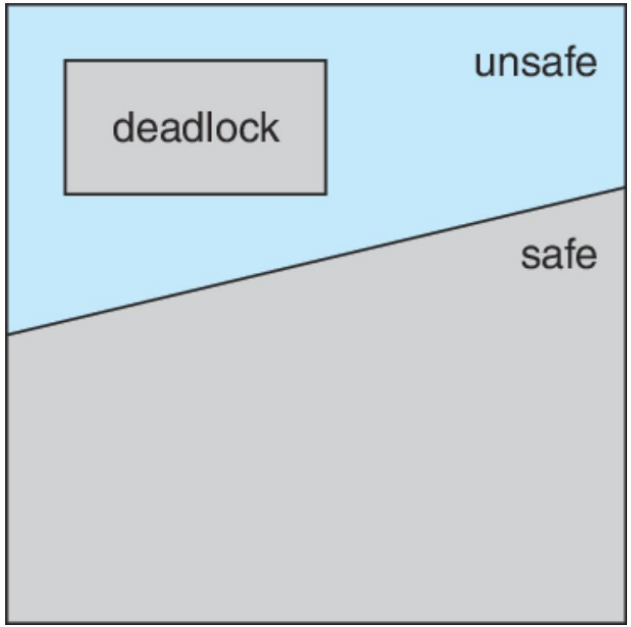
- Snapshot at time T_0 looks like:

	Allocation			Request			Available			Work		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	0	0	0	0	0	0	0	0	0
P_2	2	0	0	2	0	2						
P_3	3	0	3	0	0	0						
P_4	2	1	1	1	0	0						
P_5	0	0	2	0	0	2						

- Sequence $<P_1, P_3, P_4, P_2, P_5>$ represents a safe sequence for the current state

Deadlock Avoidance: Safe State Facts

- If a system is in safe state → There is no deadlock
- If a system is in unsafe state → Possibility of deadlock
- Deadlock Avoidance → Ensure that system will never enter usafe state



Deadlock Avoidance: Safe State Example

- Consider a system with 3 processes and 12 disks.
- Snapshot at time T_0 looks like:

	Maximum Needs	Current Allocation
P_1	10	5
P_2	4	2
P_3	9	2

- Snapshot at time T_1 looks like:

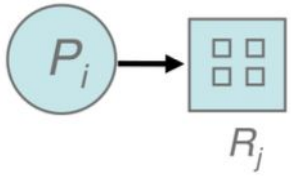
	Maximum Needs	Current Allocation
P_1	10	5
P_2	4	2
P_3	9	3

Resource Allocation Graph

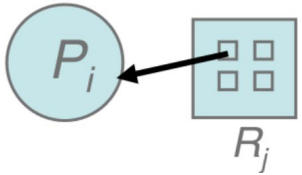
- Process

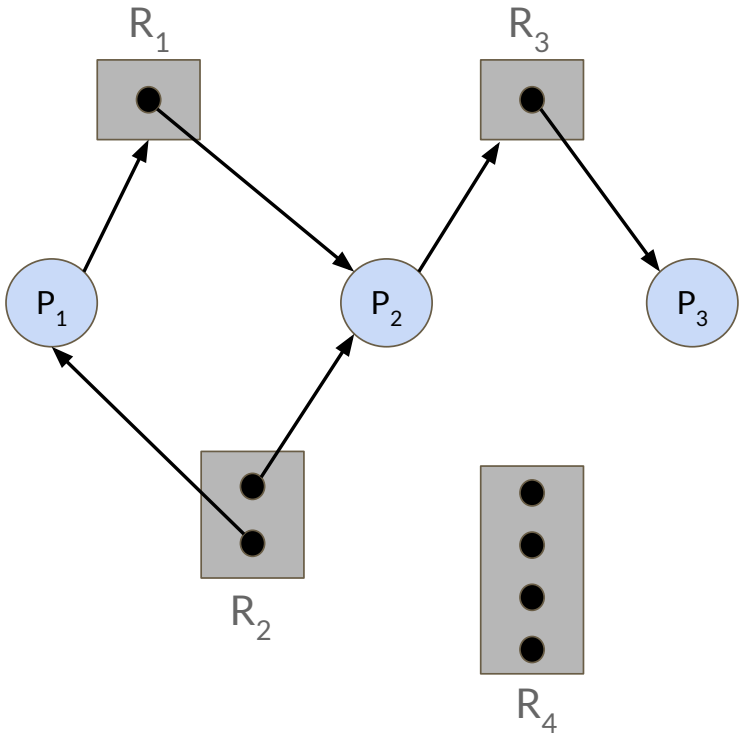
- Resource type with 4 instances

- P_i requests instance of R_j



- P_i is holding an instance of R_j

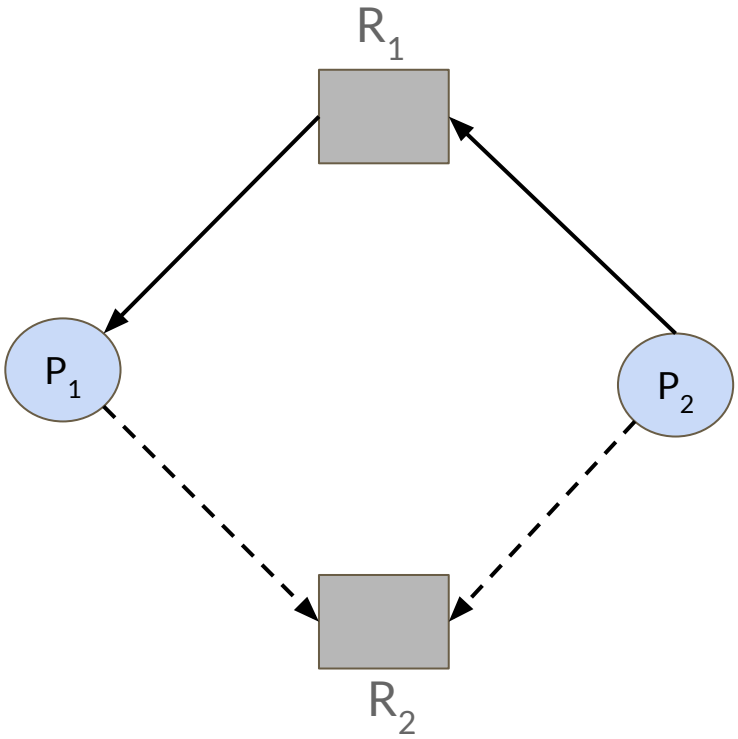




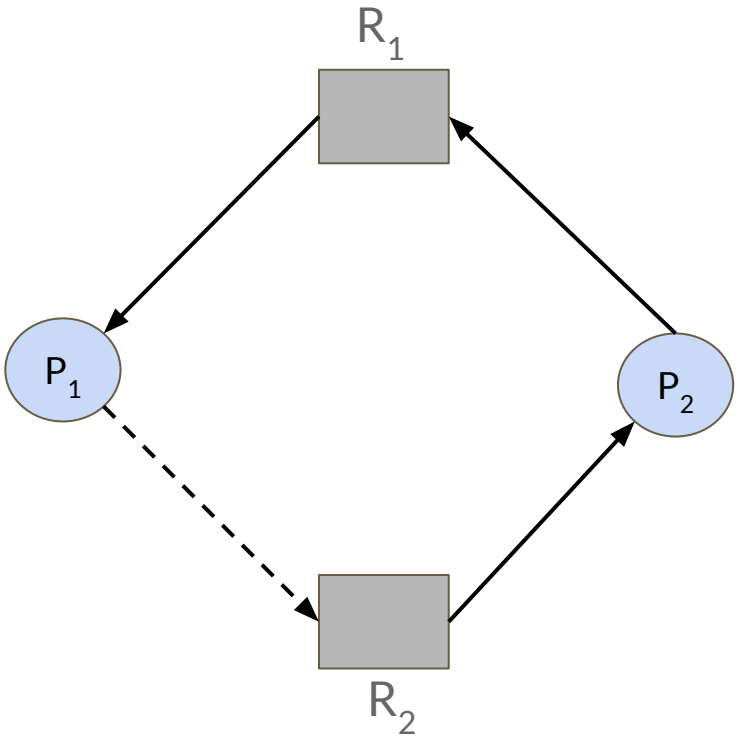
Deadlock Avoidance: Resource Allocation Graph

- Claim edge $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j (dashed line)
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- If a new allocation results in a cycle in the graph (unsafe state), we cannot allow that allocation.
- Resources must be claimed **a priori** in the system

Deadlock Avoidance: Resource Allocation Graph



Deadlock Avoidance: Resource Allocation Graph



Deadlock Avoidance: Banker's Algorithm

- Works for multiple resource instances.
- Each process declares maximum # of resources it may need.
- When a process requests a resource, it may have to wait if this leads to an unsafe state.
- When a process gets all its resources, it must return them in a finite amount of time.

Banker's Algorithm: Data Structure

- Let ***n*** = number of processes, and let ***m*** = number of resources types
- Available: Vector of length ***m***. If ***Available[j] = k***, there are ***k*** instances of resource type R_j available
- Max: ***n x m*** matrix. If ***Max[i, j] = k***, then process P_i may request at most ***k*** instances of resource type R_j
- Allocation: ***n x m*** matrix. If ***Allocation[i, j] = k***, then process P_i is currently allocated ***k*** instances of resource type R_j
- Need: ***n x m*** matrix. If ***Need[i, j] = k***, then process P_i may need ***k*** more instances of resource type R_j to complete its task

$$Need[i, j] = Max[i, j] - Allocation[i, j]$$

Banker's Algorithm: 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, ..., n-1*

2. Find an *i* such that both:

   a. ***Finish[i] == false***

   b. $\boldsymbol{Need_i \leq Work}$

   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

Banker's Algorithm: 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, ..., n-1*
2. Find an *i* such that both:

   a. ***Finish[i] == false***

   b. $\textbf{\textit{Need}}_i \leq \textbf{\textit{Work}}$

   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

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in safe state.

Banker's Algorithm: Resource-Request for Process P_i

- Let **Request**$_i$ be the 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:
     a. $Available = Available - Request_i$
     b. $Allocation_i = Allocation_i + Request_i$
     c. $Need_i = Need_i - Request_i$
  4. If safe → the resources are allocated to P_i
  5. If unsafe → P_i must wait, and the old resource-allocation state is restored

Banker's Algorithm: Example

- We have five processes P_1 through P_5, three resource types are available: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_0 looks like:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	3	3	2
P_2	2	0	0	3	2	2			
P_3	3	0	2	9	0	2			
P_4	2	1	1	2	2	2			
P_5	0	0	2	4	3	3			

Banker's Algorithm: Example

- We have five processes P_1 through P_5, three resource types are available: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_0 looks like:

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	3	3	2	7	4	3
P_2	2	0	0	3	2	2				1	2	2
P_3	3	0	2	9	0	2				6	0	0
P_4	2	1	1	2	2	2				0	1	1
P_5	0	0	2	4	3	3				4	3	1

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

Banker's Algorithm: Example

- We have five processes P_1 through P_5, three resource types are available: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_0 looks like:

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	3	3	2	7	4	3
P_2	2	0	0	3	2	2				1	2	2
P_3	3	0	2	9	0	2				6	0	0
P_4	2	1	1	2	2	2				0	1	1
P_5	0	0	2	4	3	3				4	3	1

- The system is in a safe state since the sequence $<P_2, P_4, P_5, P_3, P_1>$ satisfies safety

Banker's Algorithm: Example, P2 Requests (1, 0, 2)

- Check that Request ≤ Available. *I.e* (1, 0, 2) ≤ (3, 3, 2) → true
- Snapshot at time T_0 looks like:

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	3	3	2	7	4	3
P_2	2	0	0	3	2	2				1	2	2
P_3	3	0	2	9	0	2				6	0	0
P_4	2	1	1	2	2	2				0	1	1
P_5	0	0	2	4	3	3				4	3	1

Banker's Algorithm: Example, P2 Requests (1, 0, 2)

- Check that Request ≤ Available. *I.e* (1, 0, 2) ≤ (3, 3, 2) → true
- Snapshot at time T_1 looks like:

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	7	5	3	2	3	0	7	4	3
P_2	3	0	2	3	2	2				0	2	0
P_3	3	0	2	9	0	2				6	0	0
P_4	2	1	1	2	2	2				0	1	1
P_5	0	0	2	4	3	3				4	3	1

Banker's Algorithm: Example

- Executing safety algorithm shows that sequence $<P_2, P_4, P_5, P_3, P_1>$ satisfies safety requirement.
- Can request for (3, 3, 0) by P_5 be granted?
- Can request for (0, 2, 0) by P_1 be granted?

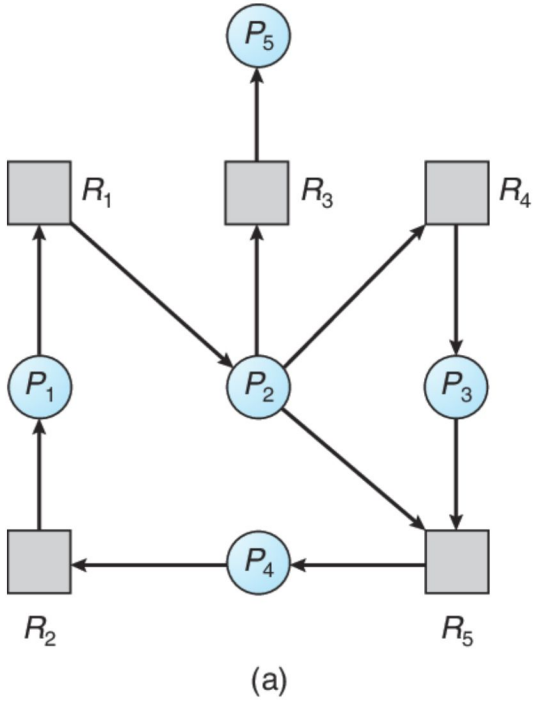
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

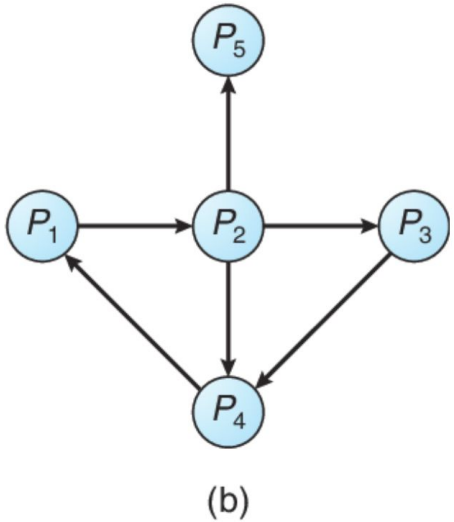
Deadlock Detection: Single-Instance Resources

- Maintain ***wait-for*** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

Deadlock Detection: Single-Instance Resources



Resource Allocation Graph



Corresponding wait-for graph

Deadlock Detection: Single-Instance Resources

- Periodically invoke an algorithm that searches for a cycle in the graph.
- 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.
- Only good for single-instance resource allocation systems.

Deadlock Detection: Several-Instance Resources

- **Available**: Vector of length ***m***, indicates the number of available resources of each type.
- **Allocation**: Matrix of size ***n x m***, defines the number of resources of each type currently allocated to each process.
- ***Request***: Matrix of size ***n x m***, 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: Detection Algorithm

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

   Work = Available

   Finish[i] = false *for i = 0, 1, ..., n-1*
2. Find an *i* such that both:

   a. ***Finish[i] == false***

   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*, $0 \le i < n$, then the system is in deadlock (with all P_i*s*)

Deadlock Detection: Detection Algorithm

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

   Work = Available

   Finish[i] = false *for i = 0, 1, ..., n-1*
2. Find an *i* such that both:

   a. ***Finish[i] == false***

   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*, $0 \le i < n$, then the system is in deadlock (with all P_i*s*)

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state.

Deadlock Detection: Detection Algorithm Example

- We have five processes P_1 through P_5, three resource types are available: A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0 looks like:

	Allocation			Request			Available			Work		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	0	0	0	0	0	0	0	0	0
P_2	2	0	0	2	0	2						
P_3	3	0	3	0	0	0						
P_4	2	1	1	1	0	0						
P_5	0	0	2	0	0	2						

- Sequence $<P_1, P_3, P_4, P_2, P_5>$ will result in ***Finish[i] = true*** for all *i*

Deadlock Detection: Detection Algorithm Example

- P_3 requests an additional instance of type *C*
- What is the state of system?

	Allocation			Request			Available			Work		
	A	B	C	A	B	C	A	B	C	A	B	C
P_1	0	1	0	0	0	0	0	0	0	0	0	0
P_2	2	0	0	2	0	2						
P_3	3	0	3	0	0	1						
P_4	2	1	1	1	0	0						
P_5	0	0	2	0	0	2						

- Deadlock exists, consisting of processes *P_2, P_3, P_4, and P_5*

Deadlock Recovery

- Process termination
- Resource preemption

Deadlock Recovery: Process Termination

- Abort all deadlocked processes
  - Expensive!
- Abort one process at a time until the deadlock cycle is eliminated
  - Overhead of deadlock detection algorithm.
- In which order should we choose to abort?
  - Priority of the process
  - How long the process has computed, and how much longer to completion
  - Resources the process has used
  - Resources the process needs to complete
  - How many processes will need to be terminated?
  - Is the process interactive or batch?

Deadlock Recovery: Resource preemption

- Selecting a victim
  - Minimize cost
- Rollback
  - Return to some safe state, restart process from that state
- Starvation
  - Same process may always be picked as victim
    - Include number of rollback in cost factor

Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings
- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from University of Nevada, Reno
- Farshad Ghanei from Illinois Tech
- T. Kosar and K. Dantu from University at Buffalo

Announcement

- Reduced homeworks!!
  - Good news for everyone
- Homework 5
  - Will be released next week
- Quiz 6
  - Next Class- Lecture 20 - 22