CS330 - Operating Systems

Deadlocks

13-10-2025

Deadlocks

Deadlocks

- Deadlock happens when
  - processes compete for access to limited resources
  - and are not synchronized properly

Thread 1:
```
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

Thread 2:
```
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```

- Conditions for deadlock
  - Mutual exclusion
  - Hold-and-wait
  - No preemption
  - Circular wait

Handling Deadlocks

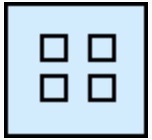
- Prevention
- Avoidance
- Recovery
- Ignore!

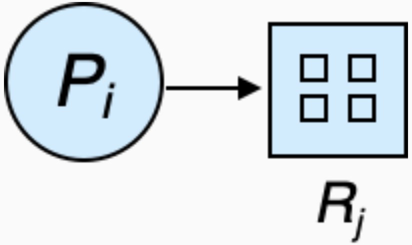
Preventing Deadlocks

- Make it impossible to have deadlocks
- Eliminate one of the four conditions

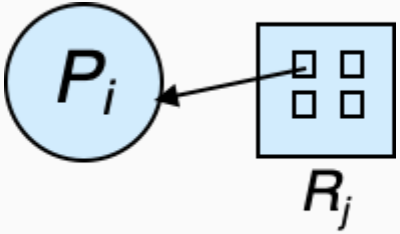
- Resource Allocation Graphs
  - System is a graph - A set of vertices V and a set of edges E
  - Processes and resources are vertices -
    $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes
    $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types
  - request edge : directed edge $P_i \rightarrow R_j$
  - assignment edge : directed edge $R_j \rightarrow P_i$

Resource Allocation Graphs

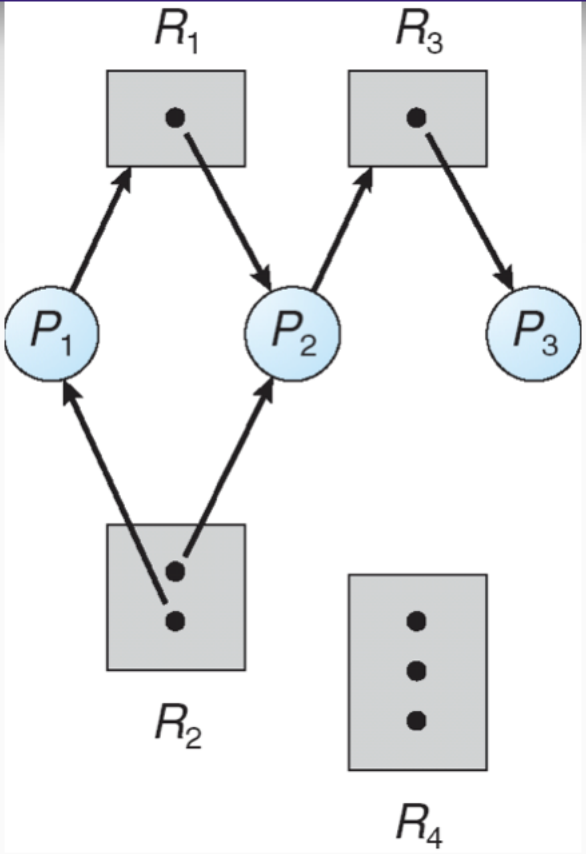
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j



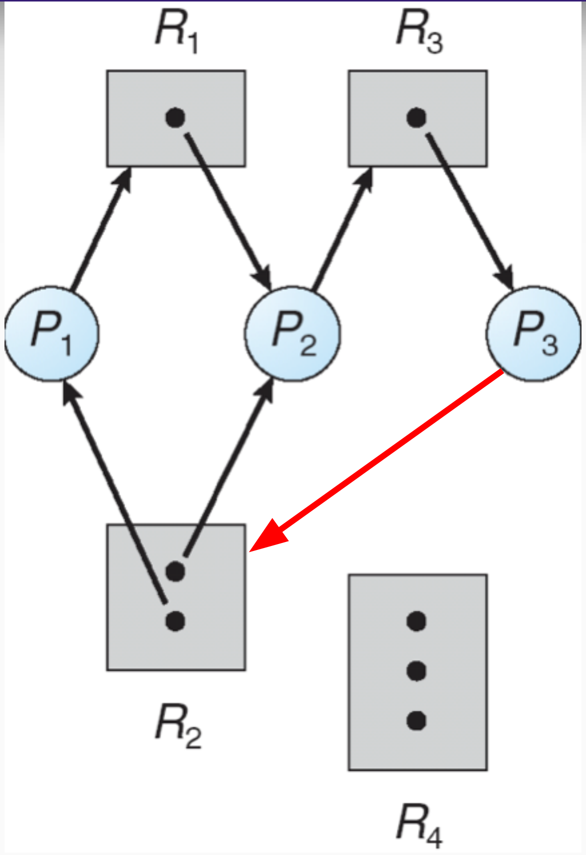
- P_i is holding an instance of R_j



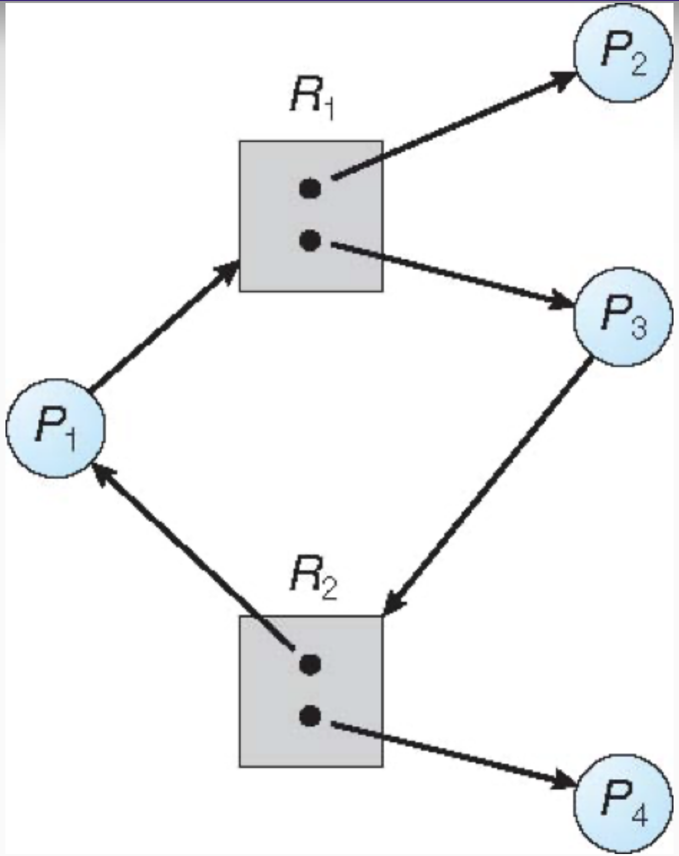
Resource Allocation Graphs



Resource Allocation Graphs



Resource Allocation Graphs



Resource Allocation Graphs

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

Preventing Deadlocks

- Mutual exclusion
  - Must hold only for non-sharable resources
- Hold and wait
  - must guarantee that whenever a process requests a resource, it does not hold any other resources
    - All resources allocated upfront
    - Can request resource when it does not hold any
    - Low resource utilization; may lead to starvation

Preventing Deadlocks

- No Preemption
  - Release all resources currently being held
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process restarts 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

Avoiding Deadlocks

- Control the allocation of resources such that it does not lead to deadlock
  - Requires that the system has some prior information available
  - Simplest model requires that each process **declares** the maximum number of resources of each type that it may need
  - Dynamically examine 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

Avoiding Deadlocks - Safe state

- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the system such that 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

Avoiding Deadlocks - Safe state

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

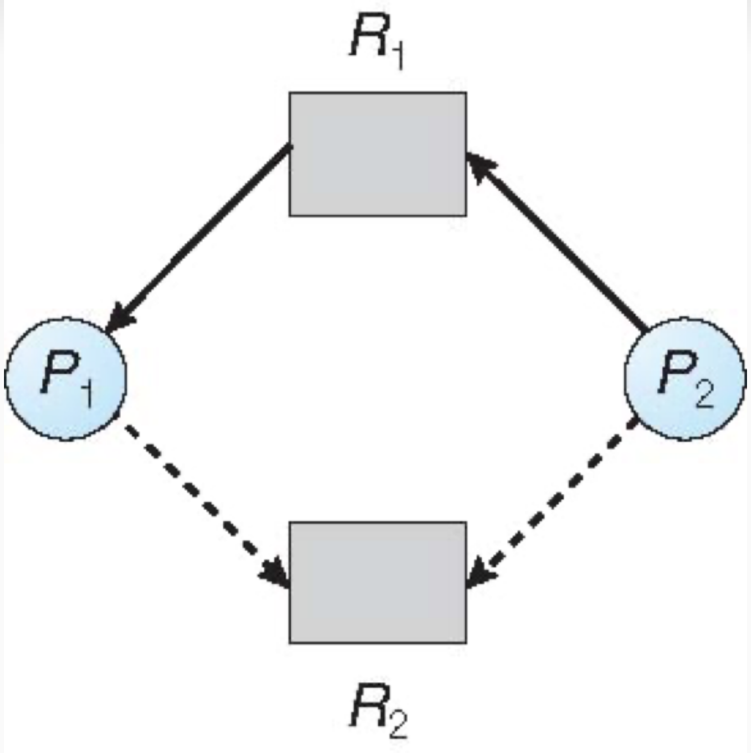
Avoiding Deadlocks

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

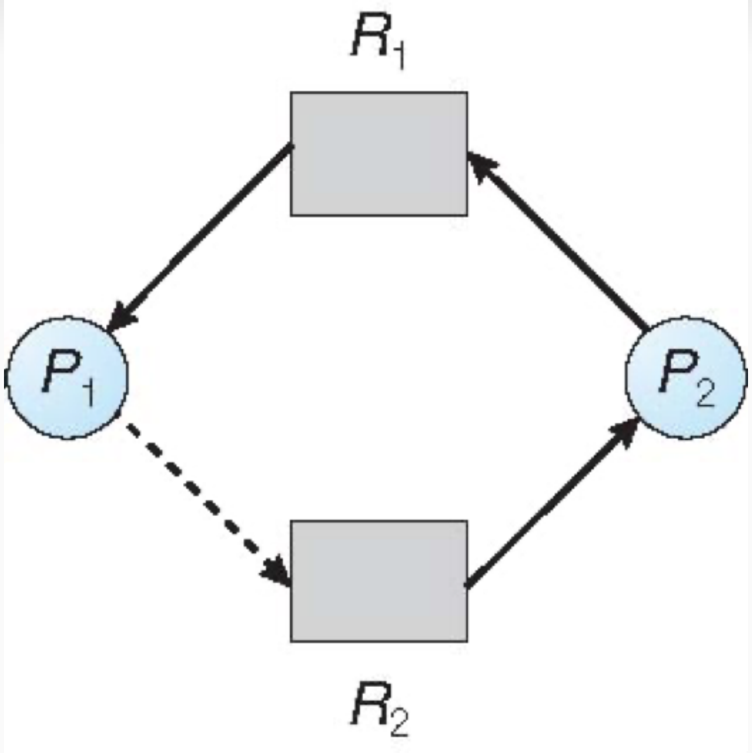
Avoiding Deadlocks

- Resources must be claimed a priori in the system
- **Claim edge** $P_i \dashrightarrow R_j$ indicates that process P_i **may** request resource R_j
- 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

Avoiding Deadlocks



Avoiding Deadlocks



Avoiding Deadlocks

- 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

Avoiding Deadlocks

- Banker's Algorithm
  - Multiple instances
  - Each process must claim maximum use at the beginning
  - 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
  - Use different data structures to enforce this
    - Max matrix
    - Allocation matrix
    - Need matrix
    - Available list

Banker's Algorithm

- 5 processes P0 through P4;
- 3 resource types:
  - A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Banker's Algorithm

- Available:

A	B	C
3	3	2

- Snapshot at time T:

	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Banker's Algorithm

- Available:

A	B	C
3	3	2

- Snapshot at time T:

Is the system in safe state?

	Allocation			Max			Need		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Banker's Algorithm

- Available:

A	B	C
3	3	2

- Snapshot at time T:

Yes, there exists a sequence <P1, P3, P4, P2, P0>

	Allocation			Max			Need		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Deadlock Detection

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

Deadlock Detection with Single Instances

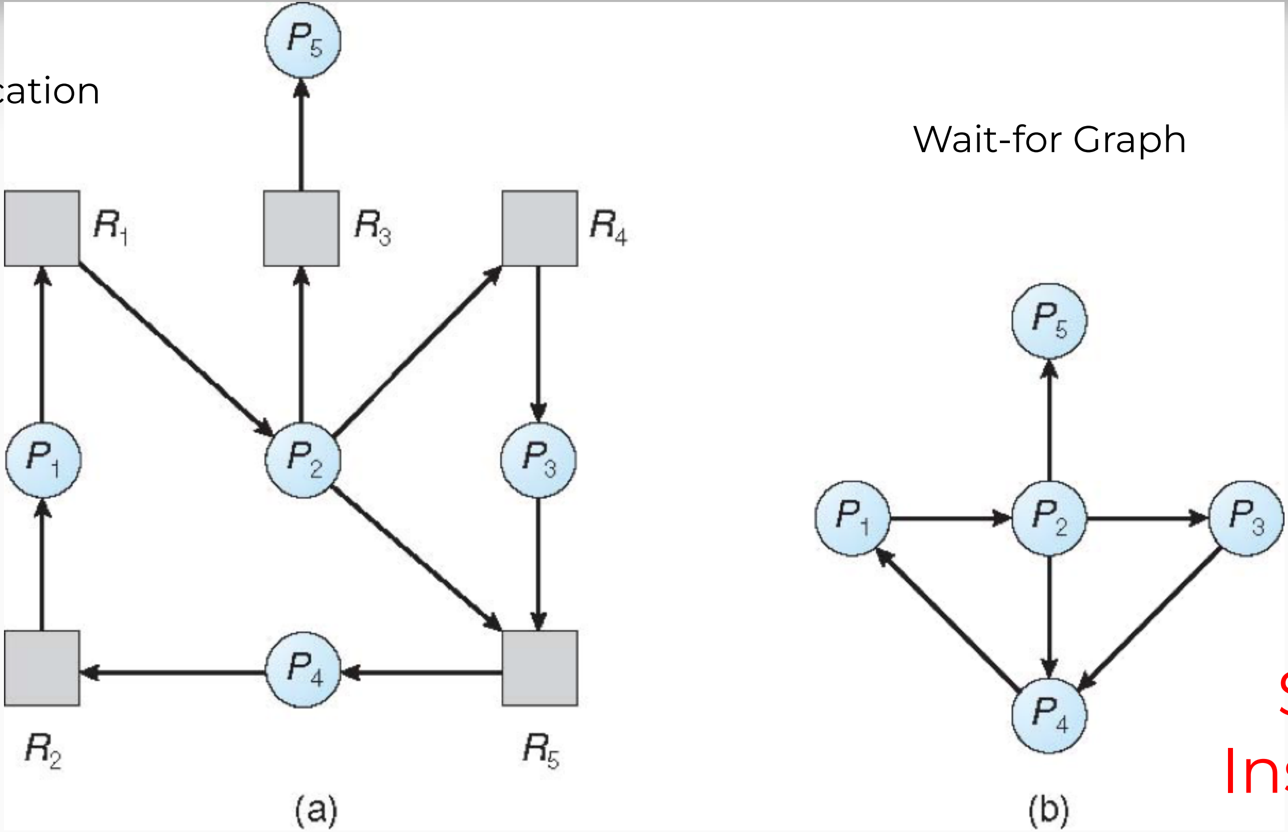
- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

- Periodically, invoke an algorithm that searches for cycles in the graph. If there is a cycle, there exists a deadlock!
- What is the complexity of the algorithm?

Deadlock Detection with Single Instances

Resource-Allocation Graph

Wait-for Graph



Several Instances?

Deadlock Recovery

- Abort all deadlocked processes (or)
- Abort 1 process at a time until the 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
- Select a victim while minimizing cost
- **Rollback** to some safe state, restart process for that state
- Same process may always be picked as victim (starvation)