COSC 4600
Operating Systems Design

Spring 2019

Chapter 7: Deadlock

The Deadlock Problem

- For a set of blocked processes, each holds a resource and waits to acquire a resource held by another process in the set.
- Example
  - System has 2 disk drives.
  - P_1 and P_2 each hold one disk drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  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

Deadlock can arise if four conditions hold simultaneously.

- **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 that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that
  - P_0 is waiting for a resource that is held by P_1,
  - P_1 is waiting for P_2,
  - …,
  - and P_n is waiting for a resource that is held by P_0.

Methods for Handling Deadlocks

- Prevention and avoidance
  - Ensure that the system will *never* enter a deadlock state.
- Detection and recovery
  - Allow the system to enter a deadlock state and then recover.
- Ostrich strategy
  - Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems.

Deadlock Prevention

Restrain the ways request can be made.

❑ Mutual Exclusion

- must hold for nonsharable resources.

❑ Hold and Wait

- Require process to request and be allocated all its resources before it begins execution
- Problem?
  - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- **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.
  - Problems?

- **Circular Wait**
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
  - Problems?

Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

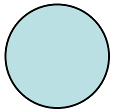
Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.

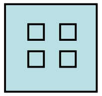
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

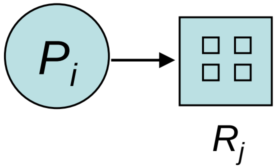
- Process



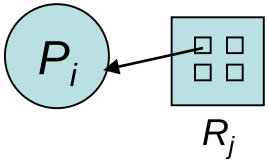
- Resource Type with 4 instances



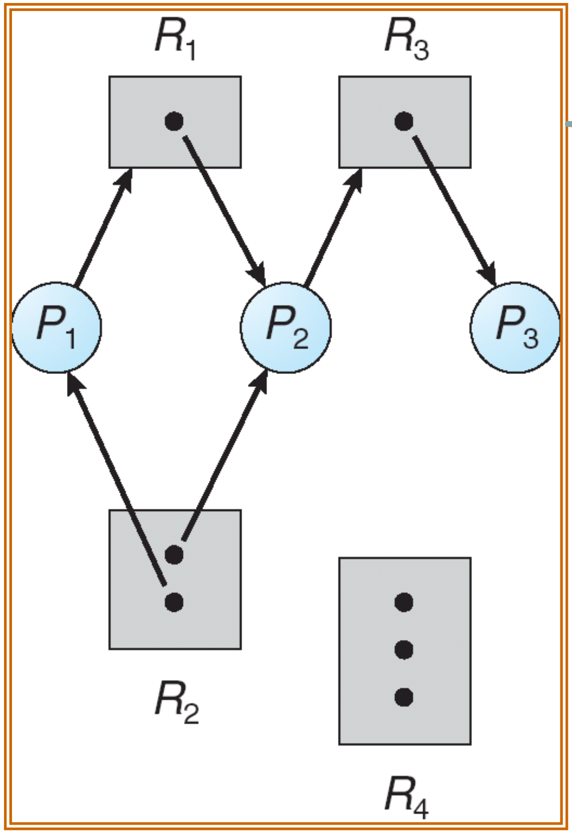
- P_i requests instance of R_j



- P_i is holding an instance of R_j



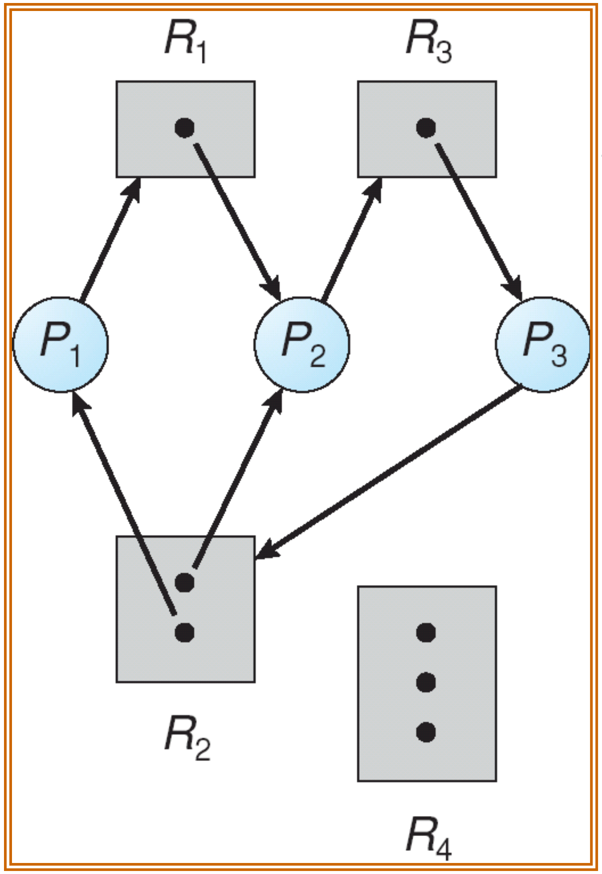
Resource Allocation Graph: Example-1



Is there deadlock?

NO

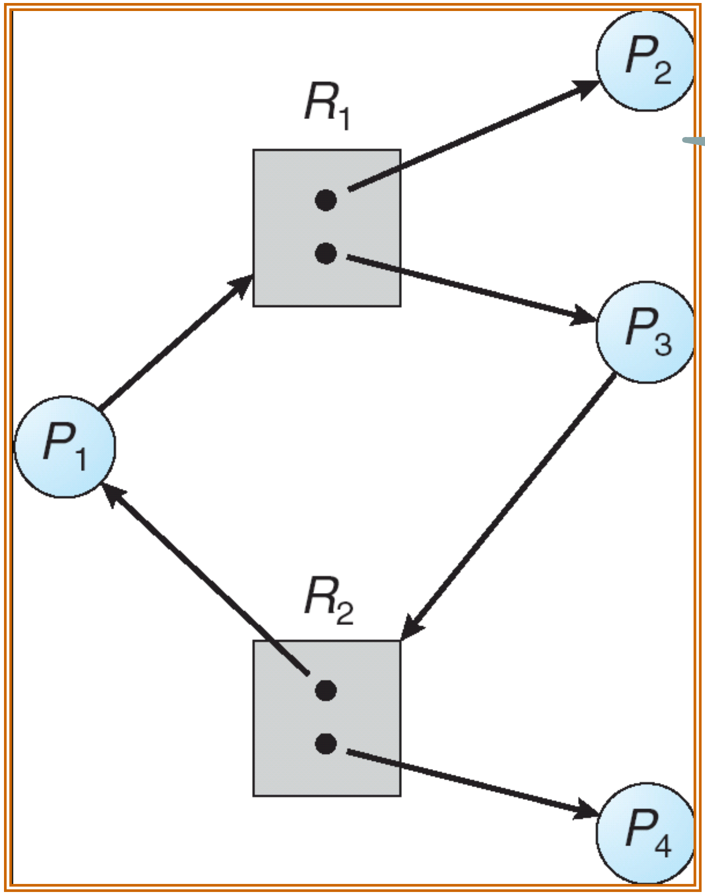
Resource Allocation Graph: Example-2



Is there deadlock?

Yes!

Graph With A Cycle



Is there deadlock?

NO

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.

- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

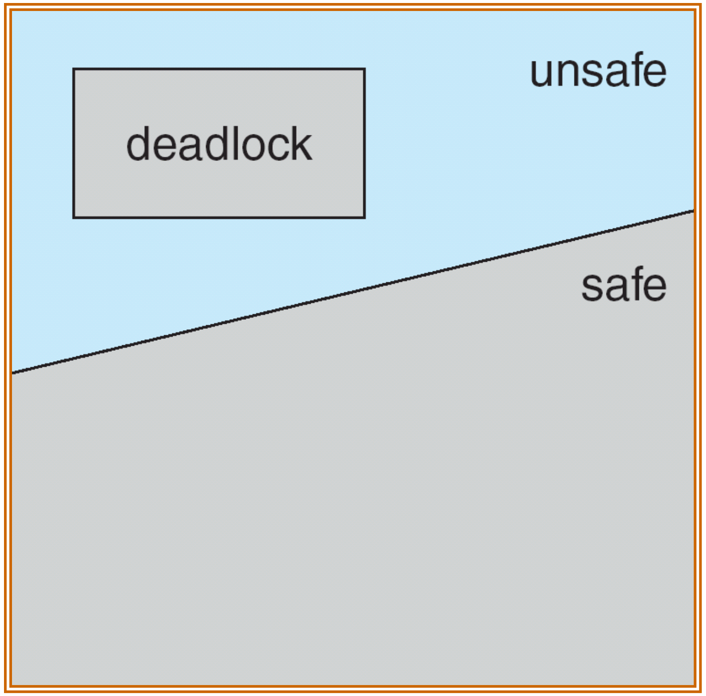
Safe State

- 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 sequence which ensures all processes to finish.

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.



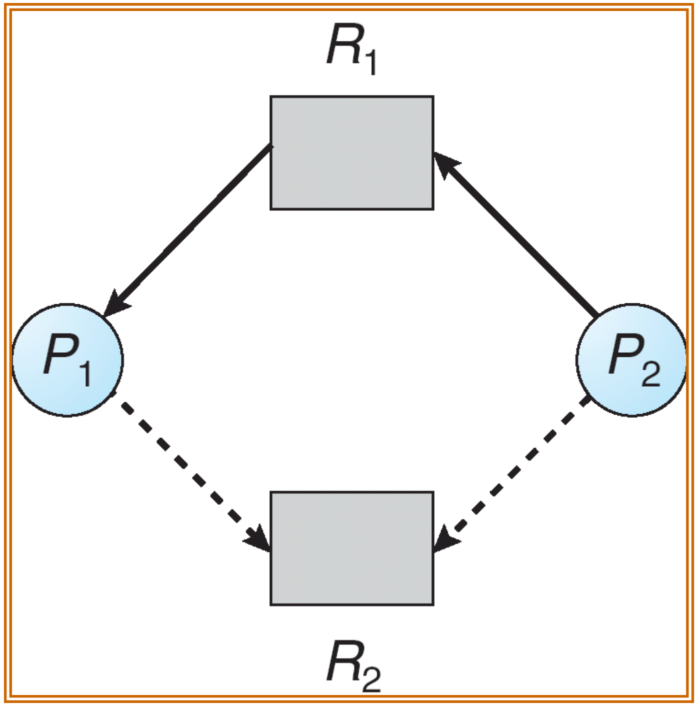
Avoidance algorithms

- 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 Scheme

- ***Claim edge*** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j some time in the future
  - represented by a dashed line.
  - Before P_i starts executing, all its claim edges must already appear in the resource-allocation graph.

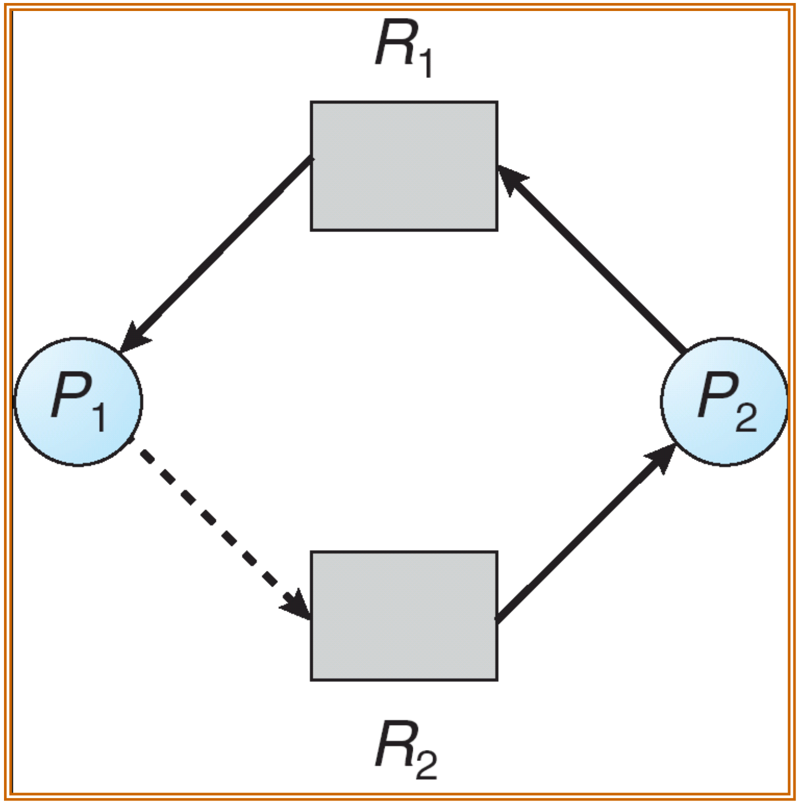


- Claim edge converts to request edge when a process requests a resource.

- Request edge converted to an assignment edge when the resource is allocated to the process.

Resource-Allocation Graph 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

- Multiple instances.
- Each process must a priori claim 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 the Banker's 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 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 P_i is currently allocated k instances of $R_{j.}$
- ***Need***: n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

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

Resource-Request Algorithm for Process P_i

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:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

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

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

Example of Banker's Algorithm

- 5 processes
- 3 resource types:

  A (10 instances), *B* (5instances), and *C* (7 instances).
- Snapshot at time T_0:

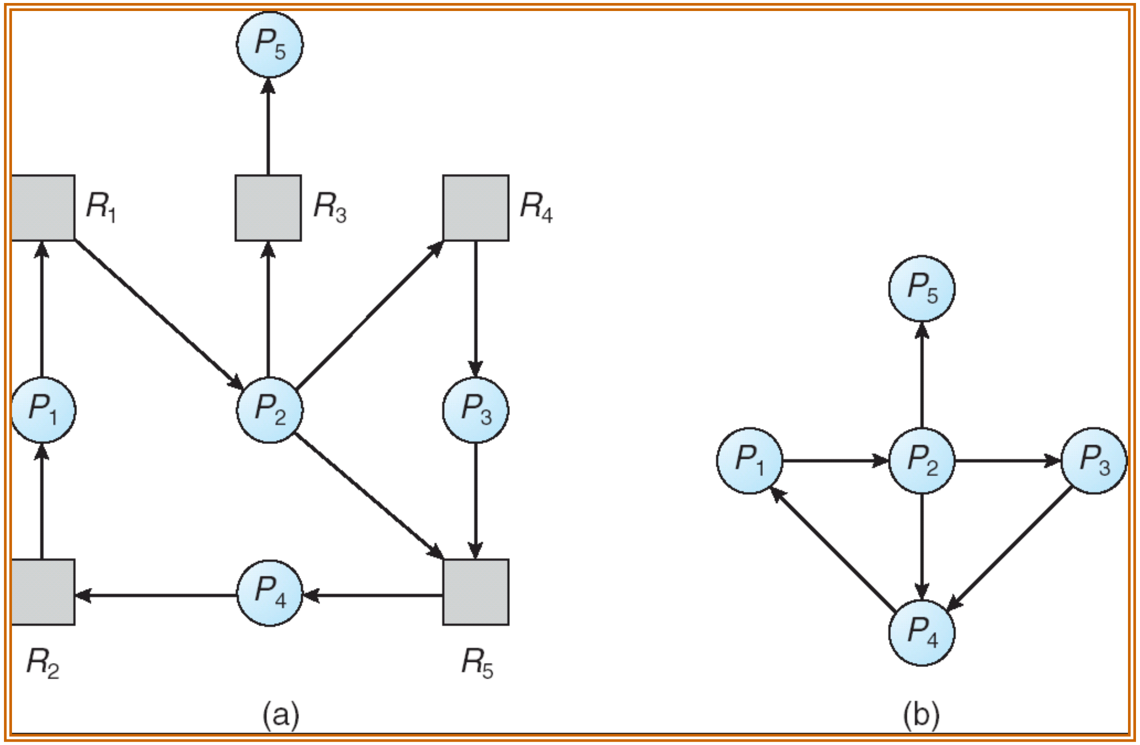
	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	3 3 2
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria.

Deadlock Detection

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

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Single Instance of Each Resource Type

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle