Deadlock

Dr. Manmath N. Sahoo
Dept. of CSE, NIT Rourkela

Shareable vs Non-shareable resources

- Sharable resources
  - can be used by multiple processes during their execution.
  - e.g., CPU, I/O bus.
- Non-sharable resources
  - cannot be used by multiple processes during their execution.
  - e.g., printer

Static vs. Dynamic resource allocation

- **Static**: Allocate all the required resources prior to the execution of the process.
- **Dynamic**: Allocate only the initially required resources to start up the process; additional resources will be allocated dynamically.

Static	Dynamic
▶ Simpler to implement ▶ Process is guaranteed to complete in definite time ▶ Resource utilization less ▶ Deadlock will never occur	▶ Comparatively complex ▶ No certainty about the process completion ▶ Better resource utilization ▶ May lead to **deadlock**

Deadlock

- Deadlock is a situation where a set of processes are blocked.
- Each process in the set waits for some event from some other process in the same set.
- Example #1
  - A system has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs the other one
- Example #2
  - Semaphores *A* and *B*, initialized to 1

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

Deadlock Characterization

Deadlock can arise if <u>four</u> 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 $\{P_0, P_1, \ldots, P_n\}$ 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, …, 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

Methods for Handling Deadlocks

- Prevention
  - Ensure that the system will never enter a deadlock state
- Avoidance
  - Ensure that the system will never enter an unsafe state
- Detection
  - Allow the system to enter a deadlock state and then recover

Deadlock Prevention

Restrain at least one of the causes of deadlock

- Mutual Exclusion – The mutual-exclusion condition must hold for non-sharable resources.
- Hold and Wait – we must guarantee that whenever a process requests a resource, it does not hold any other resources
  - There is no guarantee that a file that is released by a process will be in the same state when it will get for the next time.

Deadlock Prevention

- 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 preempted.
  - Preempted resources are added to the list of available resources.
  - A process will be restarted only when it can regain its old resources, and the new ones that it is requesting

Deadlock Prevention

- Circular Wait
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration. For example:
    1. R1
    2. R2
    3. R3
    4. R4

If P_i holds R3 and requests for R2 then it has release R3 first and re-request for R2 then R3

What if R3 is a file!!

Deadlock Avoidance and Safe State

- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a ***safe state***
- A system is in a safe state only if there exists a ***safe sequence of execution***
- Total number of resources=12

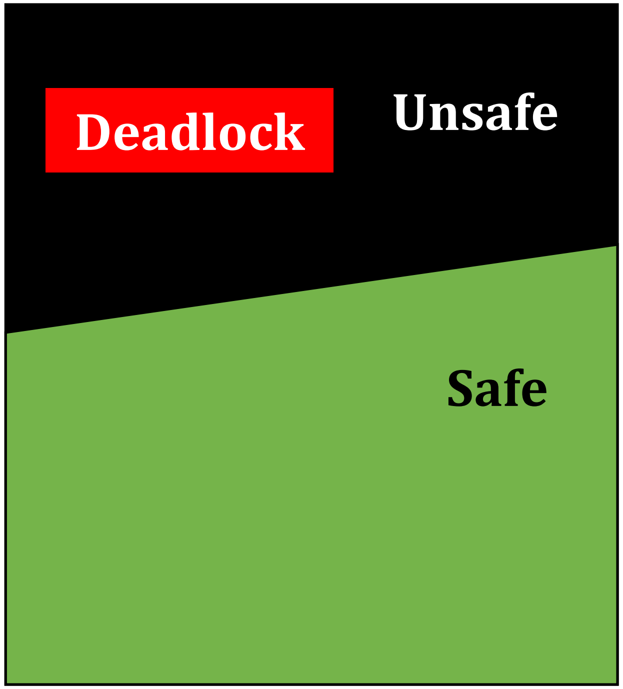
Process	Max Needs	Allocated	Current Needs
P0	10	**5**	**5**
P1	4	2	2
P2	7	3	4

- P0 requests one more resource dynamically. Can it be granted?
  - **<P1,P2,P0> is a safe sequence of execution**

Safe State (continued)

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

Safe, Unsafe, Deadlock State



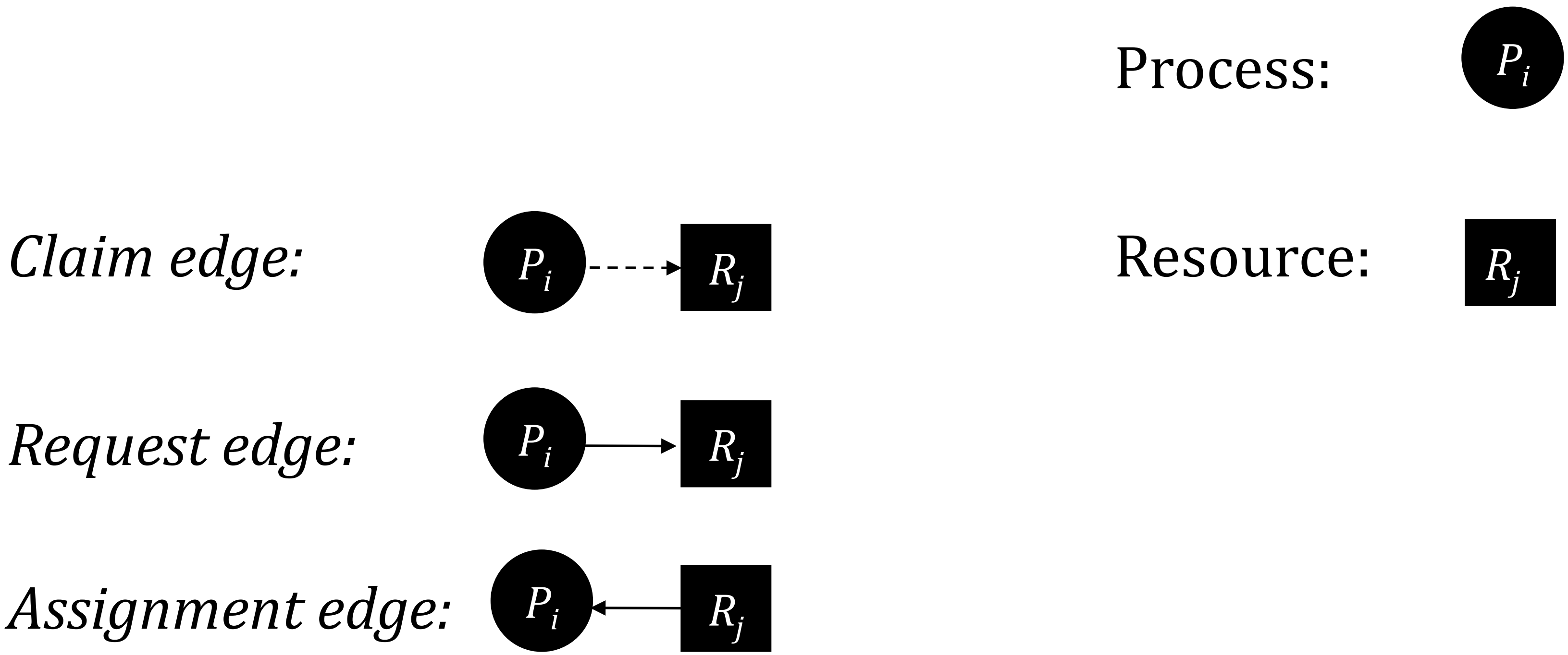
Avoidance algorithms

- For a <u>single</u> instance of a resource type, use a ***resource-allocation graph (RAG)***

- For <u>multiple</u> instances of a resource type, use the ***banker's algorithm***

RAG for Deadlock avoidance

- $G = (V, E)$
- V: Processes and Resources
- E: Request edge, Allocation edge, Claim edge



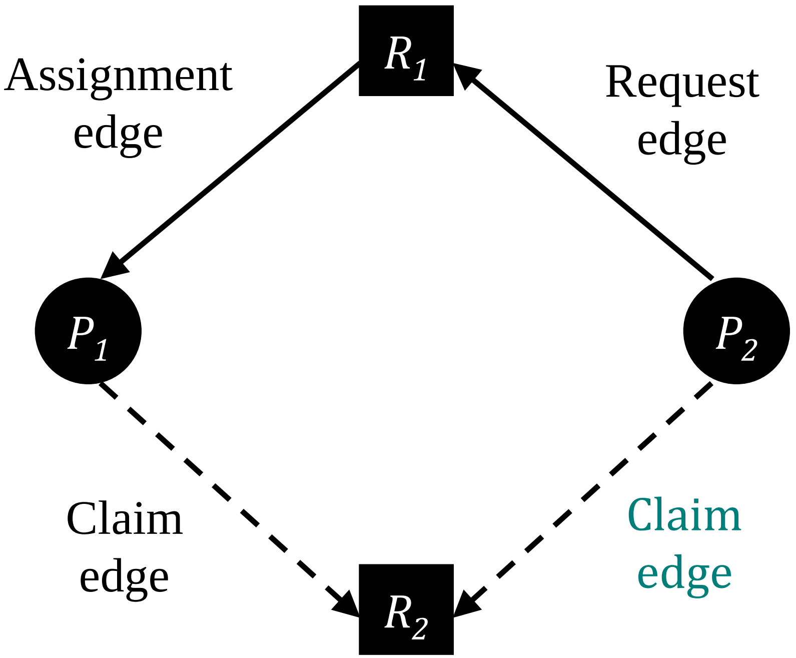
RAG for Deadlock avoidance

- *Claim edge* $P_i \dashrightarrow R_j$ indicates that process P_j may request resource R_j; which is represented by a dashed line

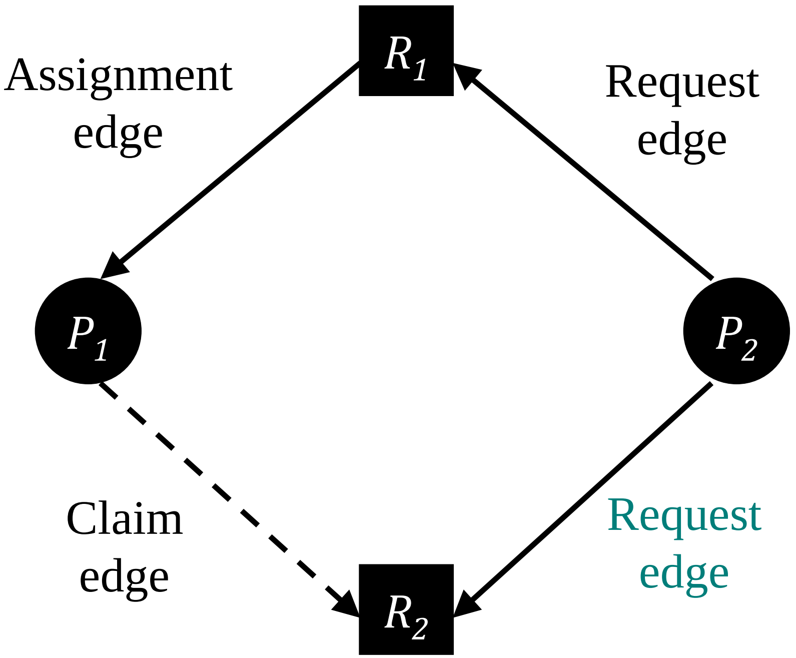
- A <u>claim edge</u> converts to a <u>request edge</u> when a process **requests** a resource

- A <u>request edge</u> converts to an <u>assignment edge</u> when the resource is **allocated** to the process

RAG for Deadlock avoidance

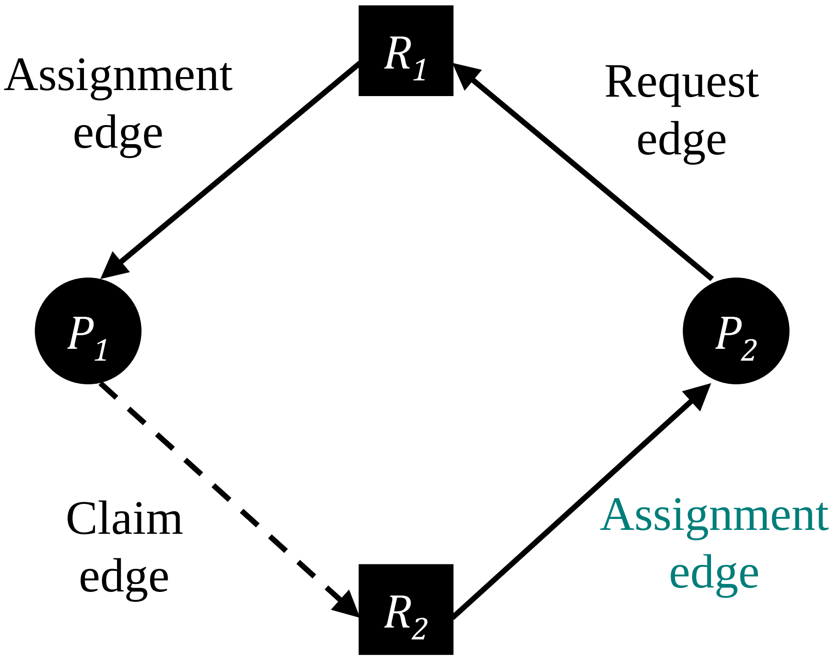


RAG for Deadlock avoidance



RAG for Deadlock avoidance

Presence of Cycle in RAG may lead to DEADLOCK

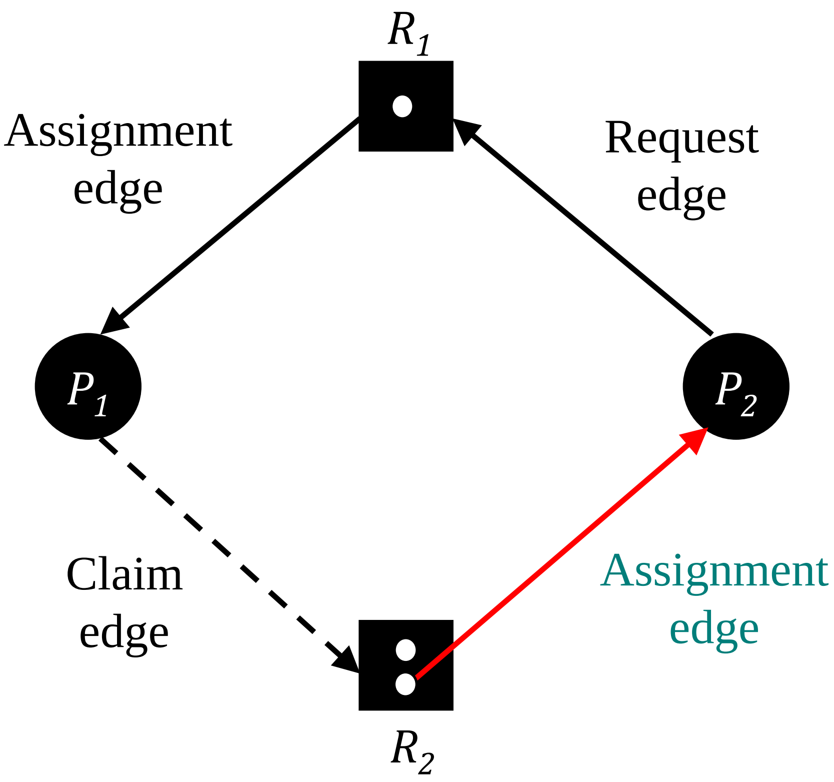


RAG for Deadlock avoidance

- 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

RAG for Deadlock avoidance

- RAG algorithm doesn't work for multiple instances of resources.



Banker's Algorithm

- When a process makes a request for a set of resources,
- Resource Request Algorithm:
  - assume that the request is granted,
  - Update the system state accordingly,
  - Then, determine if the result is a safe state (Safety algo)
    - If so, grant the request
    - if not, block the process until it is safe to grant the request.

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]
```

Safety Algorithm

1. Initialize Work = Available

   Initialize Finish[i] = False, for i = 1,2,3,..n
2. Find an i such that:

   Finish[i] == False and Need[i] <= Work

   If no such i exists, go to step 4.
3. Work = Work + Allocation[i]

   Finish[i] = True

   goto step 2
4. if Finish[i] == True for all i, then the system is in a safe state.

Banker's Algorithm: Example

Initially Available=(10,5,7)

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	3	3	2
P1	2	0	0	1	2	2			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

P1 makes $\mathbf{Request_1 = (1,0,2)}$

Banker's Algorithm: Example

Updated State.

Let's check for a safe sequence.

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

P1 makes $\mathbf{Request_1 = (1,0,2)}$

Banker's Algorithm: Example

P1 can be completed.

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

P1 makes $\mathbf{Request_1 = (1,0,2)}$

Banker's Algorithm: Example

P3 can be completed.

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	7	4	3
P1	0	0	0	0	0	0			
P2	3	0	2	6	0	0			
P3	0	0	0	0	0	0			
P4	0	0	2	4	3	1			

P1 makes $\mathbf{Request_1 = (1,0,2)}$

Banker's Algorithm: Example

P4 can be completed.

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

P1 makes **$\texttt{Request}_1$ = (1,0,2)**

Banker’s Algorithm: Example

P0 can be completed.

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	0	0	0	0	0	7	5	5
P1	0	0	0	0	0	0			
P2	3	0	2	6	0	0			
P3	0	0	0	0	0	0			
P4	0	0	0	0	0	0			

P1 makes $\mathbf{Request_1 = (1,0,2)}$

Banker's Algorithm: Example

P2 can be completed.

Process		Allocation				Need				Available		
		A	B	C		A	B	C		A	B	C
P0		0	0	0		0	0	0		10	5	7
P1		0	0	0		0	0	0				
P2		0	0	0		0	0	0				
P3		0	0	0		0	0	0				
P4		0	0	0		0	0	0				

P1 makes **$\mathbf{Request_1}$ = (1,0,2)**

Banker's Algorithm: Example

Safe Sequence: <P1,P3,P4,P0,P2>

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

P1 makes **Request$_1$ = (1,0,2) --> GRANTED**

Banker's Algorithm: Example

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

Now, P4 makes $\texttt{Request}_4$ **= (3,3,0) --> NOT AVAILABLE**

Banker's Algorithm: Example

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

Now, P0 makes $\mathbf{Request_0 = (0,2,0)}$

Banker's Algorithm: Example

Updated State.

Let's check for a safe sequence.

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	3	0	7	2	3	2	1	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

Now, P0 makes $\mathbf{Request_0 = (0,2,0)}$

Banker's Algorithm: Example

No process can complete

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	3	0	7	2	3	2	1	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

Now, P0 makes **$\texttt{Request}_0$ = (0,2,0)--> NOT GRANTED**

Banker's Algorithm: Example

Revert back to previous state

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

Disadvantage: Time complexity is very high.

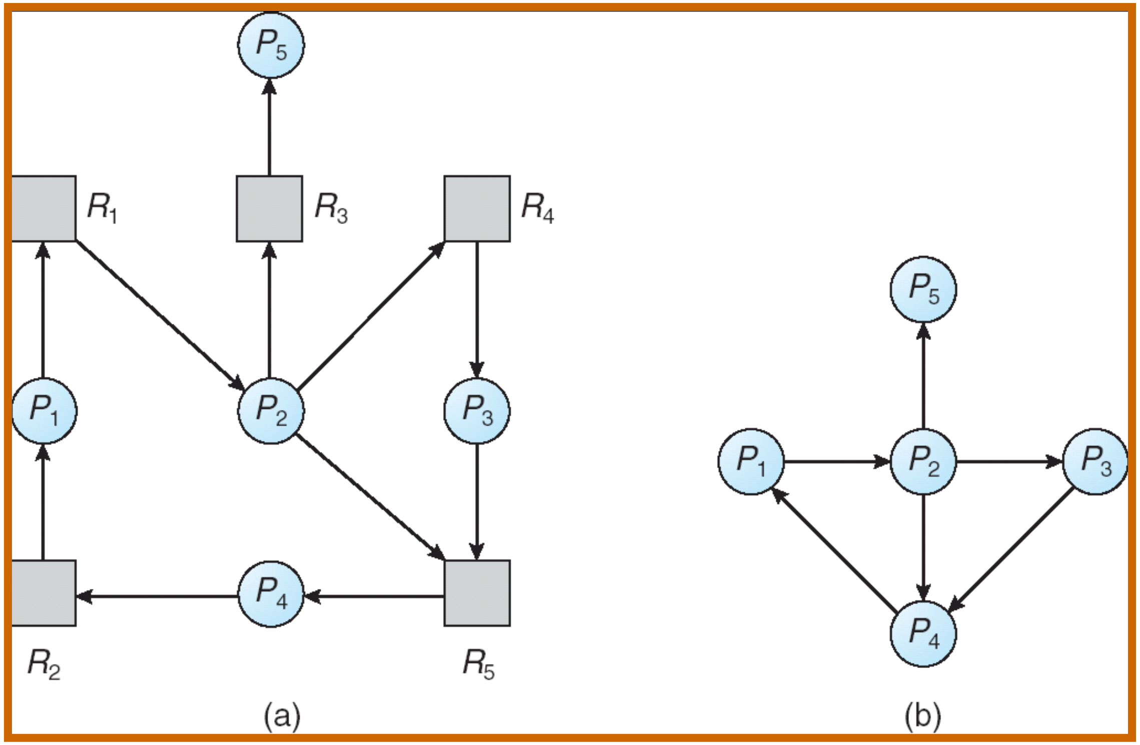
Deadlock Detection and Recovery

- An algorithm that examines the state of the system to detect whether a deadlock has occurred
- And an algorithm to recover from the deadlock

Deadlock Detection: Single Instance of Each Resource Type

- Requires the creation and maintenance of a *wait for graph (WFG)*
  - variant of the resource-allocation graph
  - The graph is obtained by **removing** the resource nodes from a RAG and **collapsing** the appropriate edges
  - Consequently; all 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 order of n^2 operations, where n is the number of vertices in the graph

Deadlock Detection: Single Instance of Each Resource Type



Resource-Allocation Graph Corresponding wait-for graph

Deadlock Detection: Multiple Instances of Resource Types

Variation of Banker's Algorithm

1. Initialize Work = Available

   Initialize Finish[i] = False, for i = 1,2,3,..n

2. Find an i such that:

   Finish[i] == False and Request[i] <= Work

   If no such i exists, go to step 4.

3. Work = Work + Allocation[i]

   Finish[i] = True

   goto step 2

4. if Finish[i] == true for all i, then the system is not in deadlocked state

Deadlock Detection Algorithm: Example

Sequence *<P0, P2, P3, P1, P4>* results in *Finish[i] == true* for all *i*
⇒ ***System is not in deadlocked state***

Process	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Deadlock Detection Algorithm: Example

Finish[0] is Ture but **P1, P2, P3, P4 are deadlocked.**

Process	Allocation			Request			Available		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	**1**			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Recovery from Deadlock

- Two Approaches
  - Process termination
  - Resource preemption

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
  - This approach will break the deadlock, but at great expense
- Abort one process at a time until the deadlock cycle is eliminated
  - This approach incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be re-invoked to determine whether any processes are still deadlocked
- Many factors may affect which process is chosen for termination
  - What is the priority of the process?
  - How long has the process run so far and how much longer will the process need to run before completing its task?
  - How many and what type of resources has the process used?
  - How many more resources does the process need in order to finish its task?
  - How many processes will need to be terminated?

Recovery from Deadlock: Resource Preemption

- With this approach, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- When preemption is required to deal with deadlocks, then three issues need to be addressed:
  - **Selecting a victim** – Which resources and which processes are to be preempted?
  - **Rollback** – If we preempt a resource from a process, what should be done with that process?
  - **Starvation** – How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

Summary

- Both deadlock avoidance and deadlock detection and recovery are expensive
- **Do nothing**: ignore the problem altogether and pretend that deadlocks never occur in the system (used by Windows and Unix)
  - System administrator/user will restart the processes