Principles of Operating Systems

Lecture 5 - Deadlocks

Ardalan Amiri Sani (ardalan@uci.edu)

[*lecture slides contains some content adapted from previous slides by Prof. Nalini Venkatasubramanian, and course text slides © Silberschatz*]

The Deadlock Problem

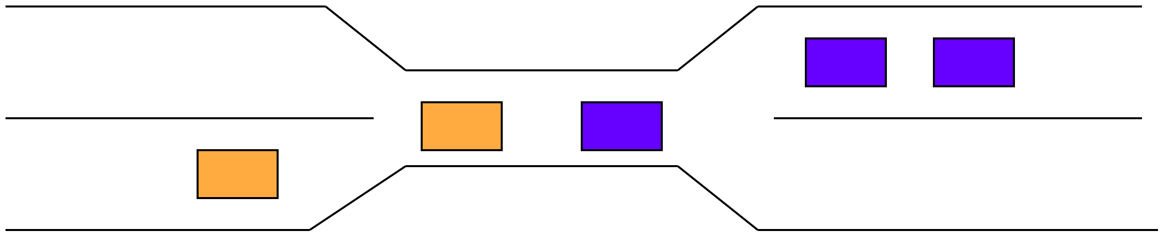
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
  - Example 1
    - Consider two files. P1 and P2 each holds exclusive access to one file and needs access to the other file.
  - Example 2
    - Semaphores A and B each initialized to 1

P0	*P1*
wait(A)	*wait(B)*
wait(B)	*wait(A)*

Definitions

- A process is *deadlocked* if it is waiting for an event that will never occur.
  - Typically, more than one process will be involved in a deadlock

Example - Bridge Crossing



- Assume traffic in one direction.
  - Each section of the bridge is viewed as a resource.

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  semaphores, files, ...
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Conditions for Deadlock

Deadlock can arise if four conditions hold simultaneously.

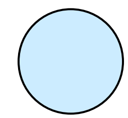
- **Mutual exclusion:** only one process at a time can use a resource (more accurately, resource instance)
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by 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.

Resource Allocation Graph

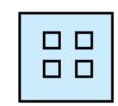
- A set of vertices V and a set of edges E
- V is partitioned into 2 types
  - P = {P1, P2,…,Pn} - the set of processes in the system
  - R = {R1, R2,…,Rn} - the set of resource types in the system
- Two kinds of edges
  - Request edge - Directed edge Pi $\longrightarrow$ Rj
  - Assignment edge - Directed edge Rj $\longrightarrow$ Pi

Resource Allocation Graph

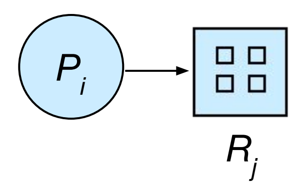
- Process



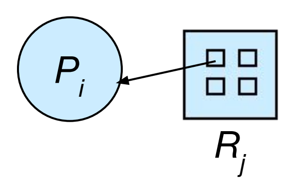
- Resource Type with 4 instances



- P_i requests an instance of R_j



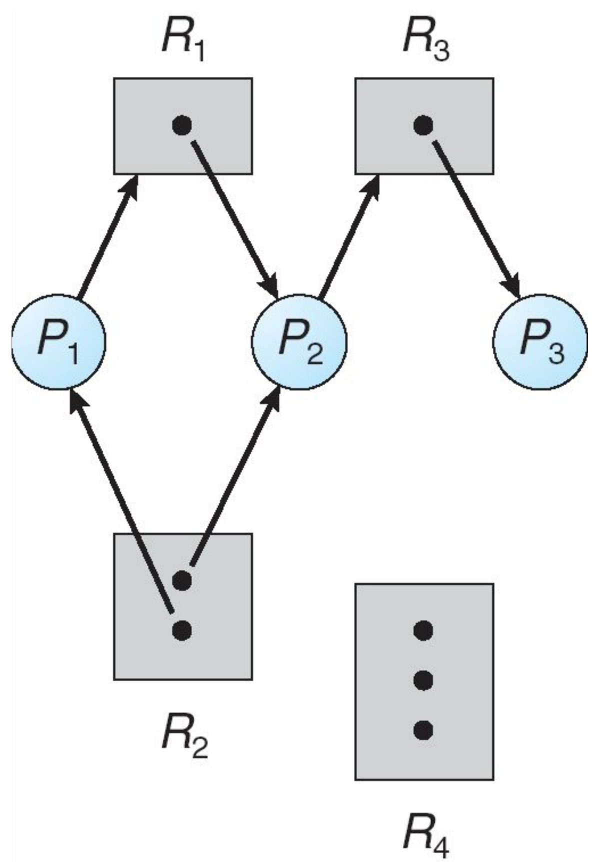
- P_i is holding an instance of R_j



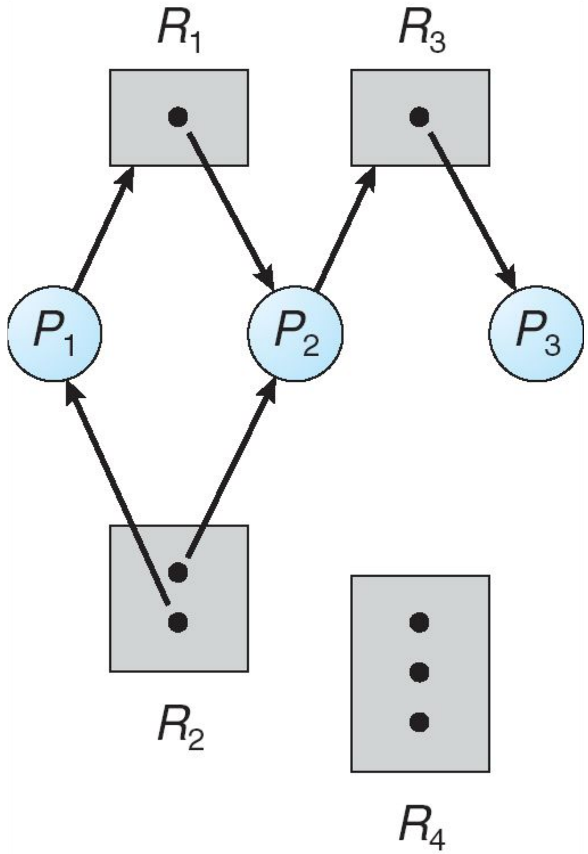
Basic facts

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

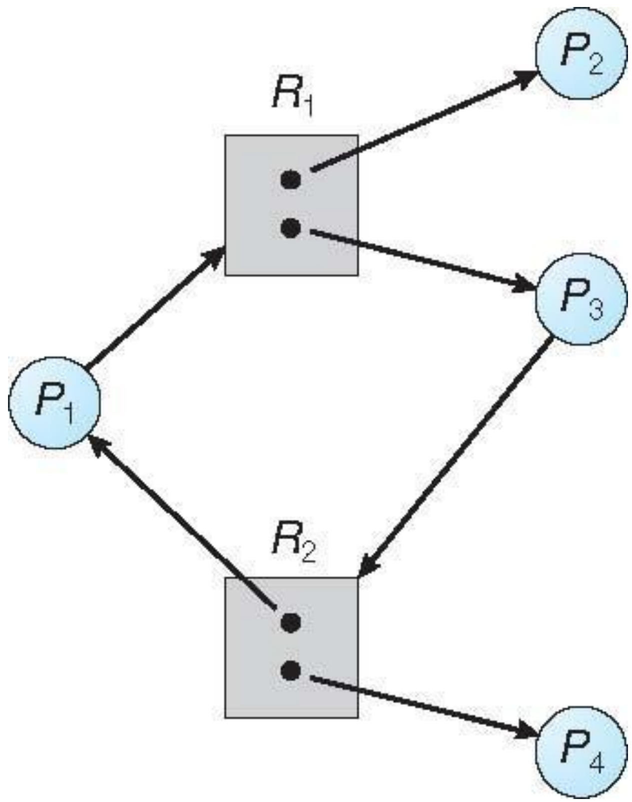
Deadlock?



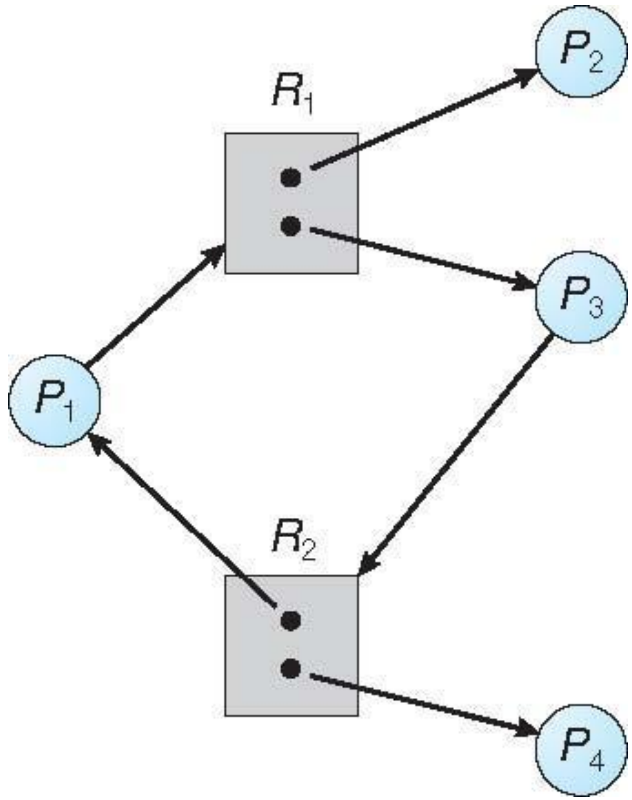
Graph with no cycles, hence no deadlock



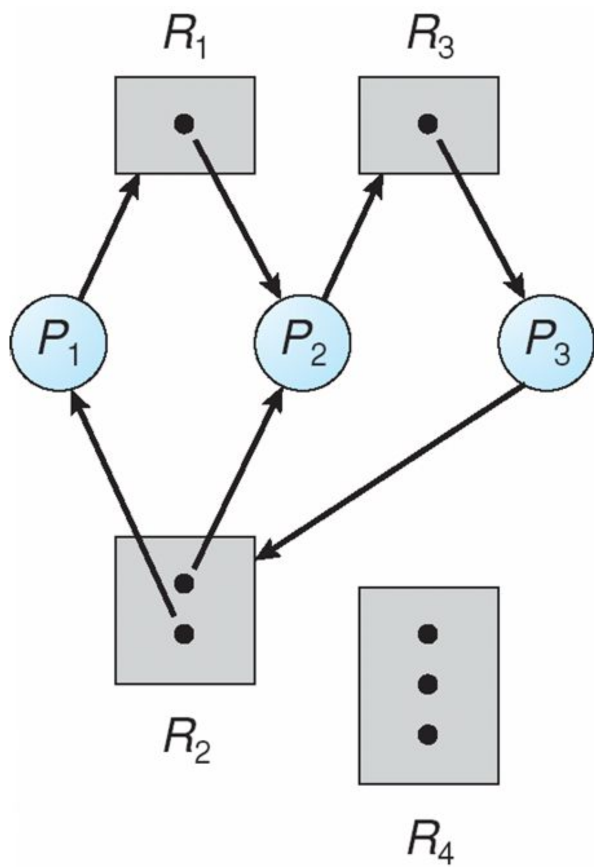
Deadlock?



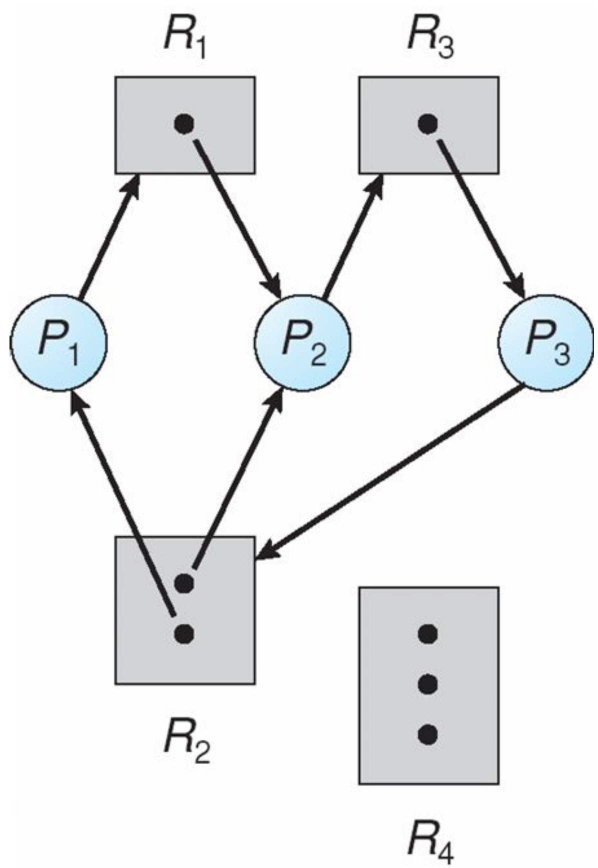
Graph with a cycle (but no deadlock)



Deadlock?



Graph with cycles and deadlock



Methods for handling deadlocks

- Ensure that the system will never enter a deadlock state.
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to potentially enter a deadlock state, detect it and then recover
  - Deadlock detection
  - Deadlock recovery

Deadlock Prevention

Restrain the ways request can be made to prevent the conditions of a deadlock from happening

- **Mutual Exclusion** – ?

Deadlock Prevention

Restrain the ways request can be made to prevent the conditions of a deadlock from happening

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – ?

Deadlock Prevention

Restrain the ways request can be made to prevent the conditions of a deadlock from happening

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention (cont.)

- **No Preemption** – ?

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
  - 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
- **Circular Wait** – ?

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

Deadlock Avoidance

- Each process tells the system maximum number of resources it needs.
- System only allocates resources if the system will be in a *safe state* after the allocation.

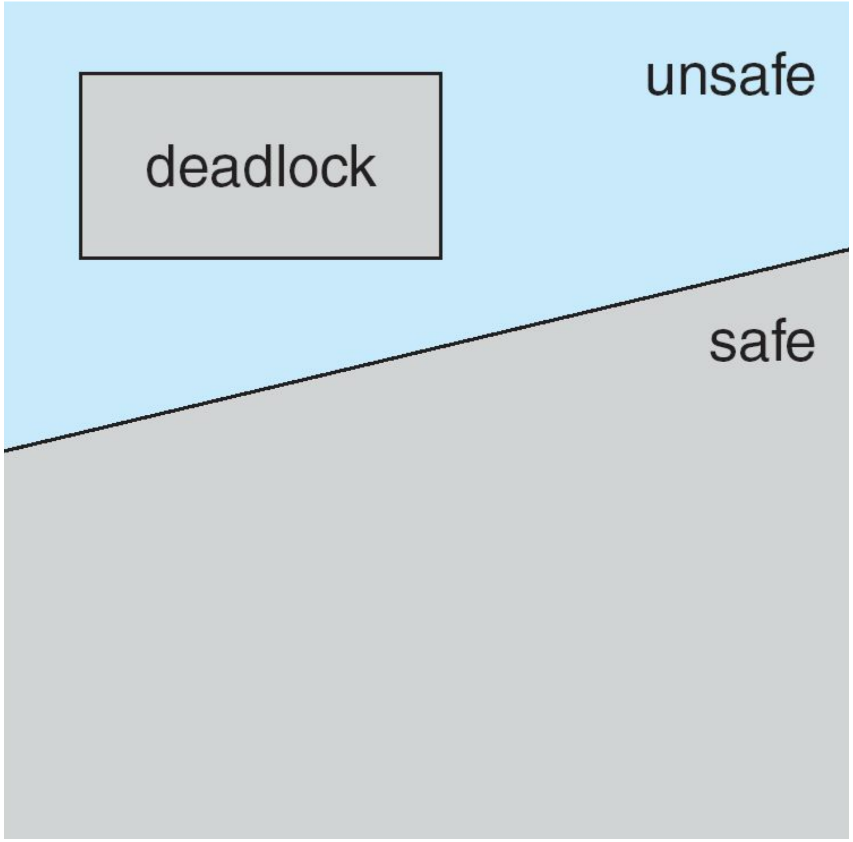
Safe state

- System is in **safe state** if there exists a sequence $<P_1, P_2, \dots, P_n>$ 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 $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished ($j < i$)
  - When all P_j ($j < i$) are 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 there is no such sequence, the system is in an unsafe state

Basic Facts

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

Safe, Unsafe, Deadlock State



Avoidance Algorithms

- Key idea
  - When there is a resource request, assume that the resource is allocated.
  - Then check the state of the system after the allocation.
  - Is it still safe? If yes, allocate the resource.
  - If not, reject the allocation request.

Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the banker’s algorithm

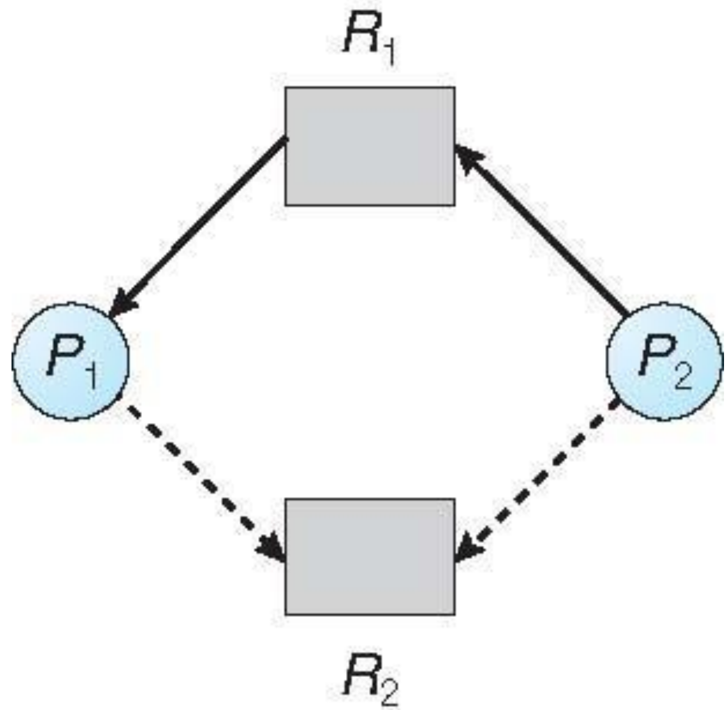
Resource Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- 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
- Resources needed by processes must be claimed *a priori* in the system

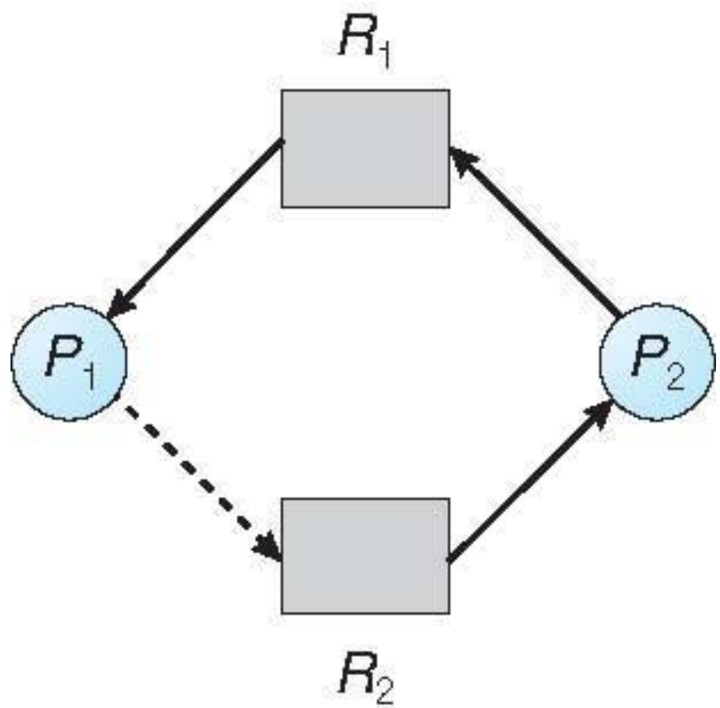
Resource Allocation Graph Algorithm

- 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

Resource Allocation Graph (aka Claim Graph)



Unsafe State in Resource Allocation Graph



Banker’s Algorithm

- Multiple instances of resource types

- Each process must a priori claim maximum use

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. 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) ***Need**$_i$* **≤ *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*] == *true*** for all ***i***, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

Request$_i$ = request vector for process $\boldsymbol{P_i}$. If ***Request***$_i$ **[*j*] = *k*** then process $\boldsymbol{P_i}$ wants ***k*** instances of resource type $\boldsymbol{R_j}$

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

$$\textit{Available} = \textit{Available} - \textit{Request}_i;$$

$$\textit{Allocation}_i = \textit{Allocation}_i + \textit{Request}_i;$$

$$\textit{Need}_i = \textit{Need}_i - \textit{Request}_i;$$

- If this is a safe state ⇒ the resources can be (and are) allocated to $\boldsymbol{P_i}$
- If this is an unsafe state ⇒ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 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:

	Allocation	*Max*	*Available*
	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	

Example (Cont.)

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

	Need
	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 < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2). Should it be granted?

- ***Check that Request$_1$ ≤ Need$_1$*** (that is, (1,0,2) ≤ (1,2,2) ⇒ true)
- Check that Request$_1$ ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true)
- Pretend the request is granted. Update the state:

	Allocation	*Need*	*Available*
	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 2	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 < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement

- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?

- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

Example: P_1 Request (1,0,2)

- ***Check that $Request_1 \le Need_1$*** (that is, $(1,0,2) \le (1,2,2) \Rightarrow$ true)
- Check that $Request_1 \le$ Available (that is, $(1,0,2) \le (3,3,2) \Rightarrow$ true)
- Pretend the request is granted. Update the state:

	Allocation	*Need*	*Available*
	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 2	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 < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> satisfies safety requirement

- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted? **No, not enough Available**

- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted? **No, the system will be unsafe**

Deadlock Detection & Recovery

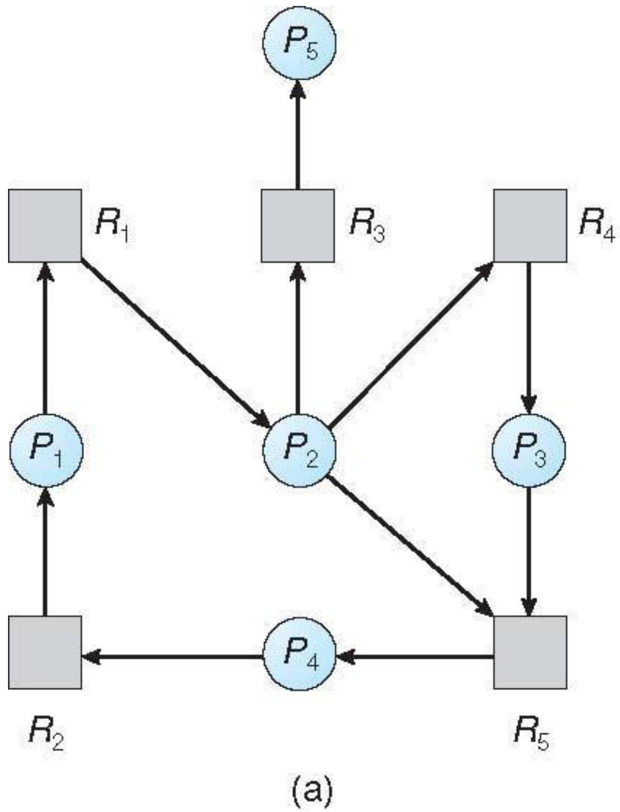
- Allow system to enter deadlock state
- Detect the deadlock
- Recover from it

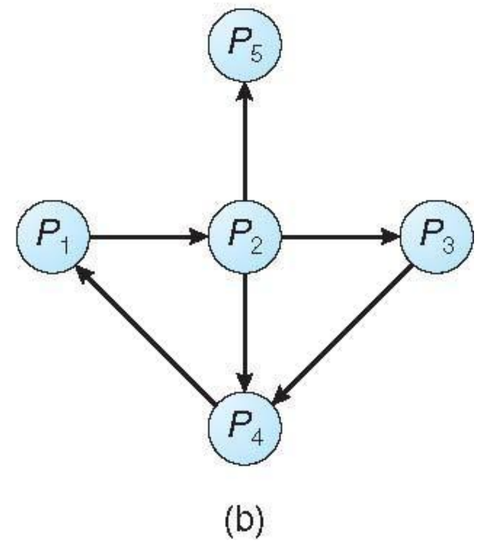
Deadlock detection: Single Instance of Each Resource Type

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

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

Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

- **Available***:* A vector of length ***m*** indicates the number of available resources of each type
- **Allocation***:* An ***n*** **x** ***m*** matrix defines the number of resources of each type currently allocated to each process
- **Request***:* An ***n*** **x** ***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.

Detection Algorithm

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

   (a) ***Work = Available***

   (b) For $\boldsymbol{i = 1, 2, \ldots, n}$, if $\boldsymbol{Allocation_i \neq 0}$, then ***Finish*****[i] =** ***false***; otherwise, ***Finish*****[i] =** ***true***

2. Find an index ***i*** such that both:

   (a) ***Finish*****[*i*] ==** ***false***

   (b) $\boldsymbol{Request_i \leq Work}$

   If no such ***i*** exists, go to step 4

Detection Algorithm (Cont.)

3. ***Work* = *Work* + *Allocation*$_i$**
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish[i] == true***, for all ***i***, 1 ≤ ***i*** ≤ ***n***, then the system is not deadlocked.

If there is no sequence of processes that results in ***Finish[i] == true***, for all ***i***, 1 ≤ ***i*** ≤ ***n***, *then the system is deadlocked.*

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	Request	Available
	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 $<P_0, P_2, P_3, P_1, P_4>$ will result in ***Finish[i] = true*** for all ***i***. Therefore, the system is not deadlocked.

Example (Cont.)

- $\boldsymbol{P_2}$ requests an additional instance of type ***C***

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?

Example (Cont.)

- $\boldsymbol{P_2}$ requests an additional instance of type ***C***

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

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

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes

- Abort one process at a time until the deadlock cycle is eliminated
  - In which order should we choose to abort?

1. Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process is holding on to
4. Resources process needs to complete
5. How many processes will need to be terminated
6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

Successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken

Considerations:

- **Selecting a victim** - Which process to select to preempt resources from?
- **Rollback** - How to preempt the resources from a process?
  - Return the process to some safe state
  - Restart the process
- **Starvation** - same process may always be picked as victim
  - include number of rollback in the decision