Deadlocks

Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

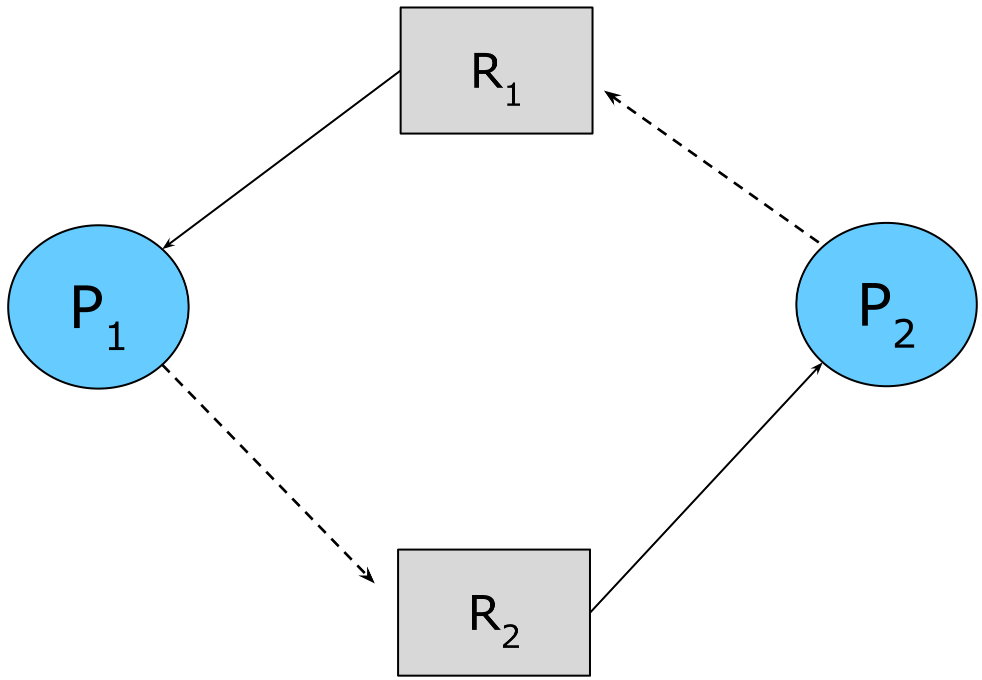
System Model

- System consists of resources
- 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!!!



A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

1. **Mutual exclusion**: only one process at a time can use a resource
2. **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
3. **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task

Deadlock Characterization

4. **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

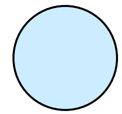
Deadlocks can be described in terms of a directed graph called a system resource-allocation graph.

It consists of a set of vertices *V* and a set of edges *E*.

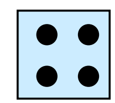
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource-Allocation Graph (Cont.)

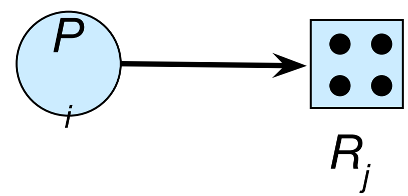
- Process



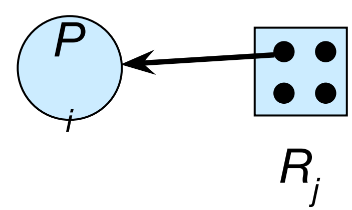
- Resource Type with 4 instances



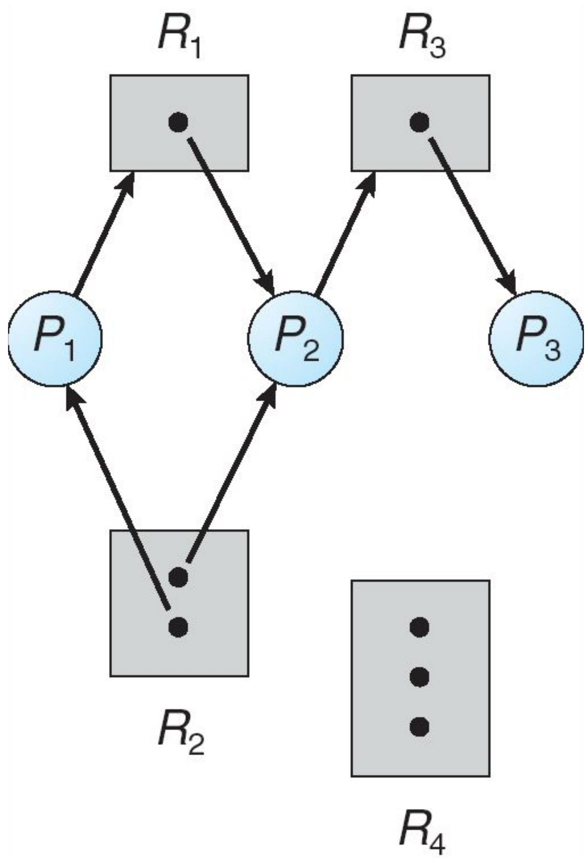
- P_i requests instance of R_j



- P_i is holding an instance of R_j



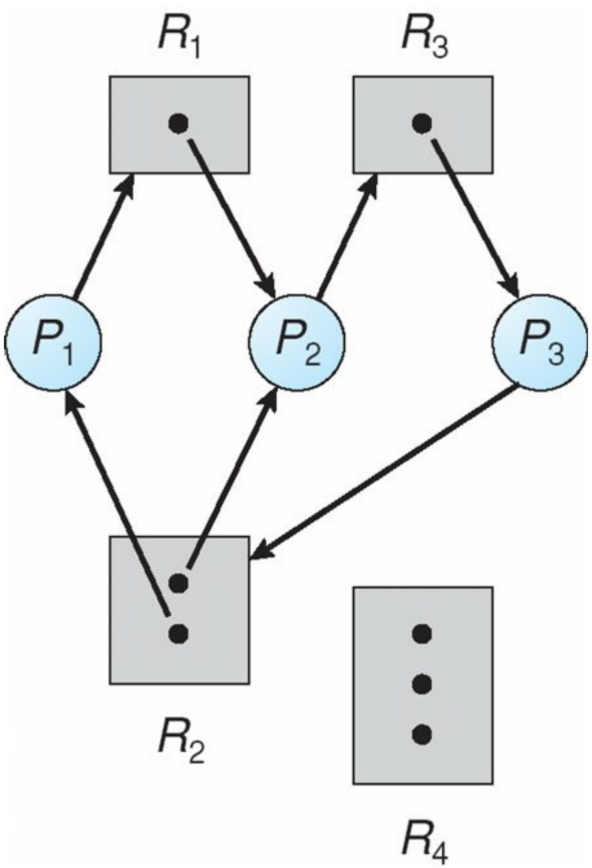
Example of a Resource Allocation Graph



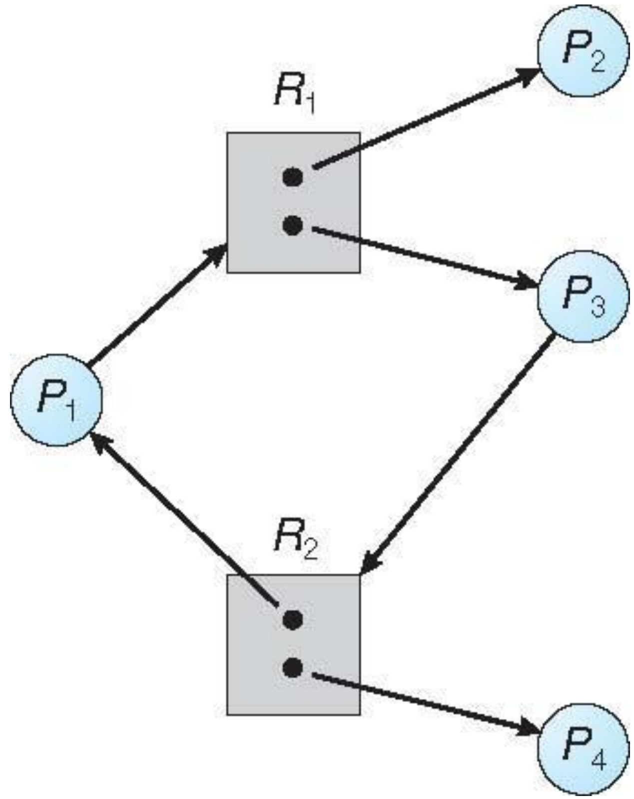
Basic Facts

- 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

Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Methods for Handling Deadlocks

1. Ensure that the system will ***never*** enter a deadlock state:
    - Deadlock Prevention
    - Deadlock Avoidance
2. Allow the system to enter a deadlock state, detect it and then recover
3. Ignore the problem and pretend that deadlocks never occur in the system;
    - used by most operating systems, including UNIX

Methods for Handling Deadlocks

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Deadlock Prevention

Deadlock Prevention

Restrain the ways request can be made

- **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** –
  - 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

Deadlock Avoidance

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

- 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

Safe State

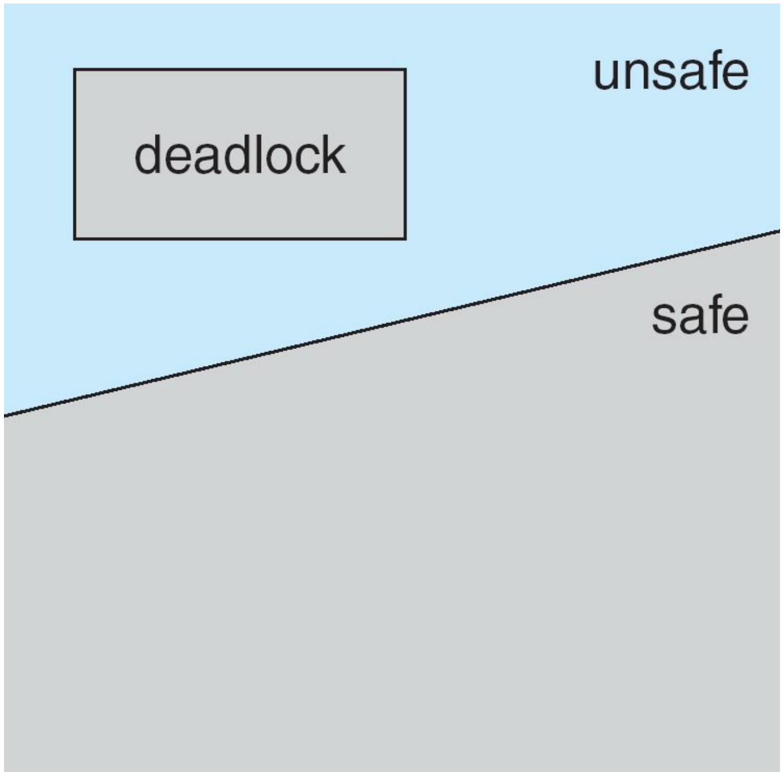
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- A state is ***safe*** *if the system can allocate resources to each process (up to its* maximum) in some order and still avoid a deadlock.

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 systems 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
  - 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, then system is said to be **unsafe**

Basic Facts: Safe, Unsafe, Deadlock State

- 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

- Consider a system with **12** disk drives and three process Po,P1 and P2

	Max Need	Allocation	Need
P_0	10	5	?
P_1	4	2	?
P_2	9	2	?

Available : ?

Safe, Unsafe, Deadlock State

- Consider a system with 12 disk drives and three process Po,P1 and P2

	Max Need	Allocation	Need	Available : 3
P_0	10	5	5	
P_1	4	2	2	
P_2	9	2	7	

- At Time t_0,the Sequence **<P_1,P_0,P_2> satisfies safety condition**

Safe, Unsafe, Deadlock State

- Now suppose, at Time t_1, P_2 requests and allotted 1 more disk drive

	Max Need	Allocation	Need	Available : 2
P_0	10	5	5	
P_1	4	2	2	
P_2	9	3	6	

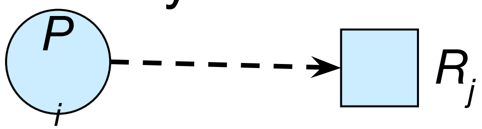
Unsafe state ≡ Deadlock!!!

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

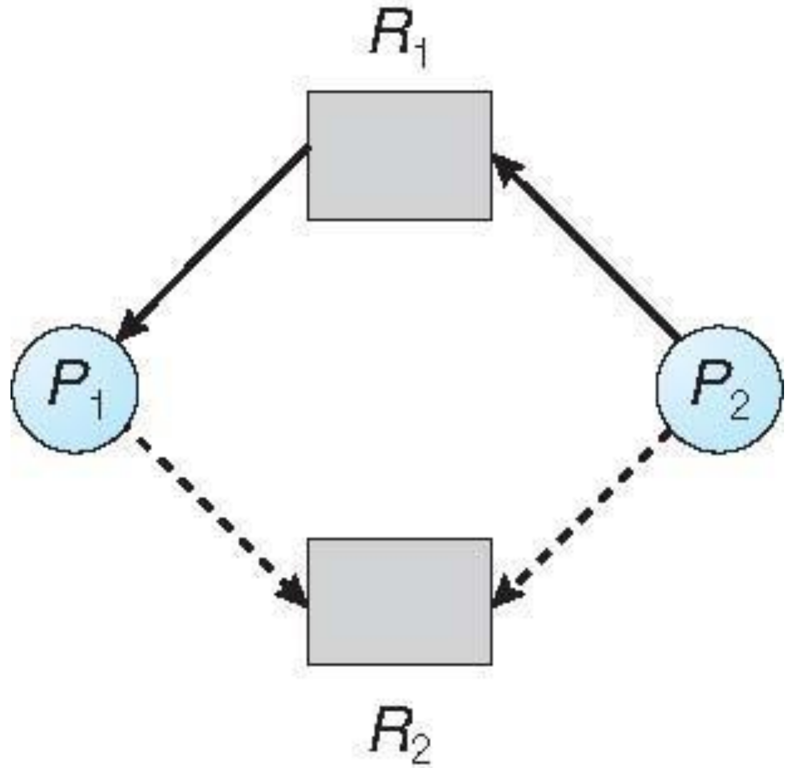


- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

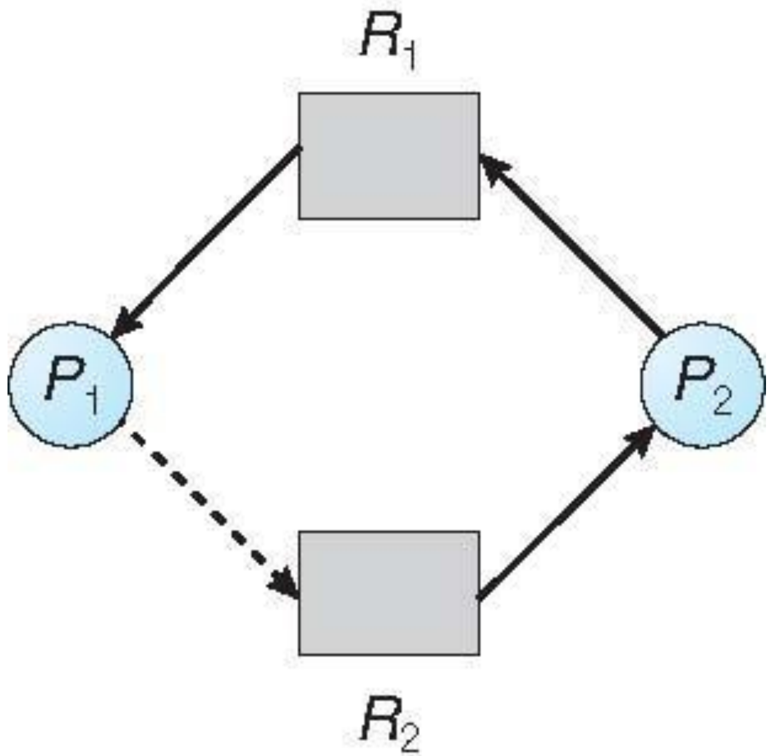
Resource-Allocation Graph Algorithm

- 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



Unsafe State In 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

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

Some notations

- Let X and Y be vectors of length n. We say that

$$X \leq Y$$

if and only if $X[i] \leq Y[i]$ for all $i = 1,2,\ldots n$

Safety Algorithm

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

   $$\textit{Work} = \textit{Available}$$

   $$\textit{Finish}[i] = \textit{false} \text{ for } i = 0, 1, \ldots, n-1$$

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

   (a) $\textit{Finish}[i] = \textit{false}$

   (b) $\textit{Need}_i \leq \textit{Work}$

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

3. $\textit{Work} = \textit{Work} + \textit{Allocation}_i$
   $\textit{Finish}[i] = \textit{true}$
   go to step 2

4. If $\textit{Finish}[i] == \textit{true}$ for all ***i***, then the system is in a safe state

Resource-Request Algorithm

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

1. If $\boldsymbol{Request_i \leq Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $\boldsymbol{Request_i \leq Available}$, 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:

$$\boldsymbol{Available = Available - Request_i;}$$

$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$

$$\boldsymbol{Need_i = Need_i - Request_i;}$$

Resource-Request Algorithm for Process P_i

The resulting resource-allocation :

- If safe ⇒ the resources are allocated to P_i
- If unsafe ⇒ 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 (5instances), 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)

Now, suppose P1 requests (1,0,2). To decide whether this request can be granted, we first check with Resource-Request Algorithm:

1. Check $Request_1 \le Need_1$. i.e. $(1,0,2) \le (1,2,2) \Rightarrow$ true
2. Check $Request_1 \le Available$ i.e. $(1,0,2) \le (3,3,2) \Rightarrow$ true
3. Have the system pretend to have allocated the requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

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

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

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

Example: P_1 Request (1,0,2)

3. Have the system pretend to have allocated the requested resources.

	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	

Now, check whether this new system is safe using Safety Algorithm

Example: P_1 Request (1,0,2)

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement

- Can request for (3,3,0) by P_4 be granted?

- Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

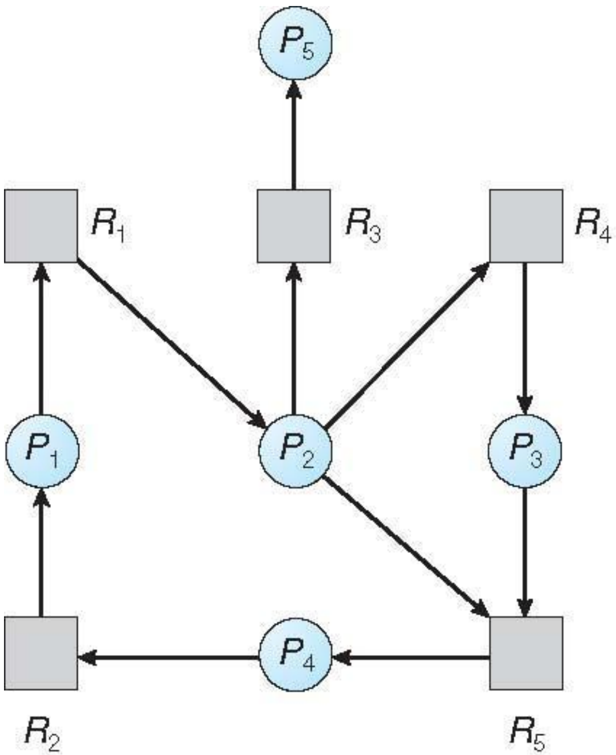
Deadlock Detection

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

Single Instance of Each Resource Type

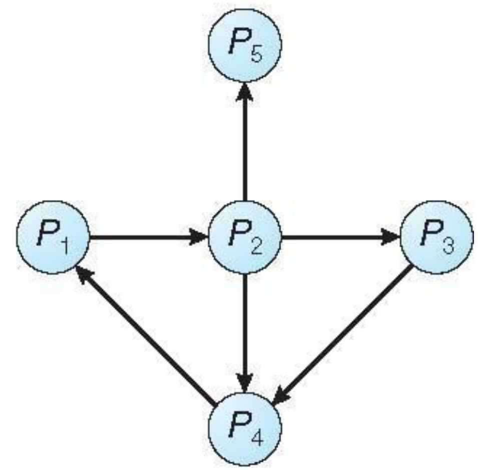
- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a)

Resource-Allocation Graph



(b)

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 $i = 1, 2, \ldots, n$, if $Allocation_i \neq 0$, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

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

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

   (b) $Request_i \leq Work$

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

3. $Work = Work + Allocation_i$

   Finish*[*i*] = *true

   go to step 2

Detection Algorithm (Cont.)

4. If ***Finish[i] == false***, for some ***i***, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if ***Finish[i] == false***, then P_i is deadlocked

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

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	

Example of Detection Algorithm

- 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***

Example (Cont.)

- ***P_2*** requests an additional instance of type ***C***

	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 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	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 $\boldsymbol{P_4}$

Detection-Algorithm Usage

- 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
- If invoked during every resource request, we can find the process "caused" deadlock. But it affects system performance.
- Better to invoke at fixed intervals or whenever CPU utilization drops below 40%
  - Can't identify the processes "caused" the deadlock.

Recovery from Deadlock

- Process Termination
- Resource preemption

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 has used
    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

- Preempt some resources from processes and give it to other processes until the deadlock cycle is broken
- **Issues to be addressed:-**

1. **Selecting a victim** – minimize cost
2. **Rollback** – return to some safe state, restart process for that state
3. **Starvation** – same process may always be picked as victim, include number of rollback in cost factor