Process Management

Deadlock

Introduction

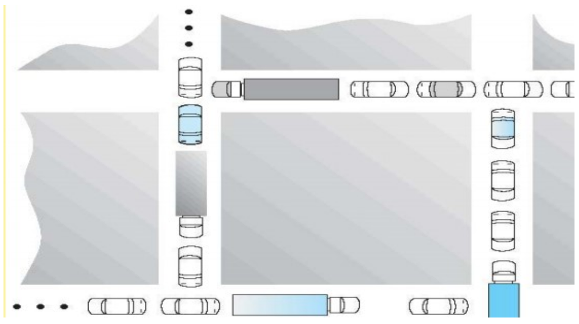
- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes, blocking the process.

Introduction

- Deadlock is a situation in which a set of blocked processes, each holding a resource, is waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape 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)*

Traffic Gridlock



- Deadlock can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs; Starvation is possible.

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

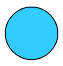
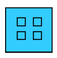
Deadlock Characterization

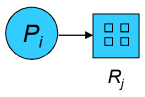
- Deadlock can arise if four conditions hold simultaneously.
  - **Mutual exclusion:** only one process at a time can use a resource.
  - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
  - **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  - **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ 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 $P_2, \ldots, P_{n-1}$ 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

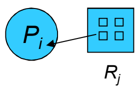
- A set of vertices *V* and a set of edges *E*.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- Request edge – directed edge $P_1 \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph

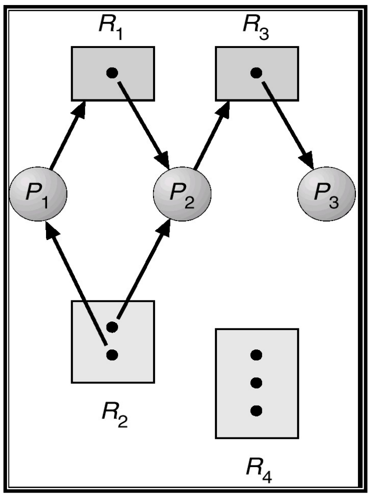
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

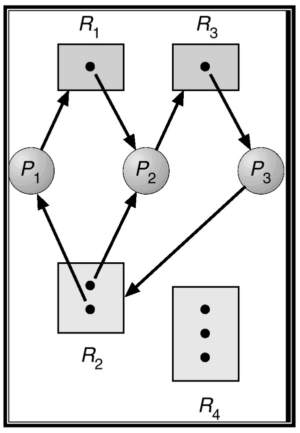


- Process P1 is holding an instance of resource R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Process P3 is holding an instance of R3

Resource-Allocation Graph

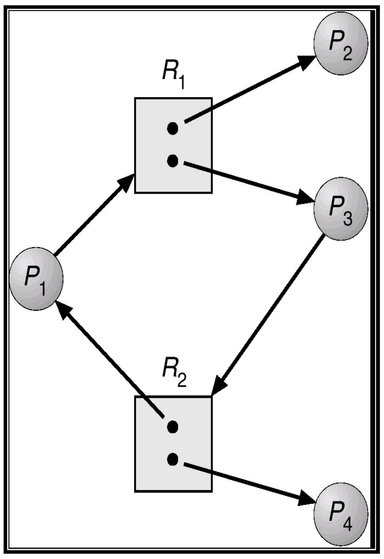
- If a graph contains no cycles, then no process is in deadlock. However, if the graph contains a cycle, then a deadlock **MAY** exists.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
- If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked.
- In this case, a cycle in the graph is a necessary and sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessary imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock



P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2



P1→ R1 → P3 → R2 → P1

The system is not in deadlock. P4 may release its instance of resource type R2 which can be allocated to P3, breaking the cycle.

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state.
  - Deadlock Prevention
  - Deadlock Avoidance
- Allow the system to enter a deadlock state and then recover.
  - Deadlock Detection and Recovery
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX/Linux.

Deadlock Prevention

- Ensure that one of the necessary conditions for deadlocks does not hold. Can be accomplished by restraining the ways requests can be made.
- **Mutual Exclusion:** Must hold for non-sharable resources that can be accessed simultaneously by various processes. For example, a printer cannot by simultaneously shared by several processes.
- Certain resources are intrinsically non sharable Therefore, Mutual exclusion cannot be used denied for deadlock prevention.

Deadlock Prevention

- **Hold and Wait:** Must guarantee that whenever a process request a resource, it does not hold any other resources.
  - One way is to require process to request and be allocated all its resources before its begin its execution. For example, to copy a file from drive to disk, sort it and print it, drive, disk and printer all three needs to be acquired, although printer is required only at the end of the process.
  - Another way is to acquire the resource, use it and then release it before requesting any other resource. Thus, acquire drive and disk, copy and release. Again request disk and printer, print the file and then release both the resources.
  - Both protocols have low resource utilization; starvation problem

Deadlock Prevention

- No Preemption: Allow Preemption
  - If a process, that is holding some resources, requests another resource that cannot be immediately allocated to it, then 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
  - Can be applied only to resources (CPU, memory space) whose state can be easily saved restored later.

Deadlock Prevention

- Circular Wait: Impose a total ordering of all resource types.
- A one to one function can be defined as F: R → N, where N is the set of natural numbers. F(R1) = n1, F(R2) = n2, ...
- Each process can requests resources in an increasing order of enumeration i.e. each process can initially request any number of instances of a resource type, say Ri. After that, the process can request instances of Rj, if and only if, F(Rj) > F(Ri)

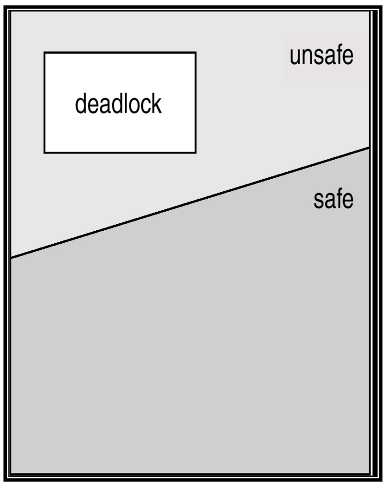
Deadlock Avoidance

- 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 of the System

- 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 safe sequence of all processes. <P_1, P_2, …, P_n> such that for each P_i, the resources that Pi 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.

Deadlock Avoidance



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.

Deadlock Avoidance Algorithms

- Single Instance of each resource type
  - Use variant of resource allocation graph
- Multiple instances of each resource type
  - Use Banker's algorithm

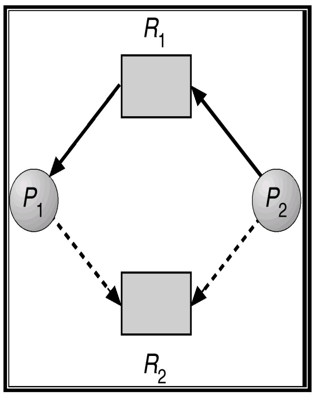
Resource Allocation graph with claim edge

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- Request edge converts 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.
- A cycle in the graph implies that the system is in unsafe state.

Resource Allocation Graph Algorithm

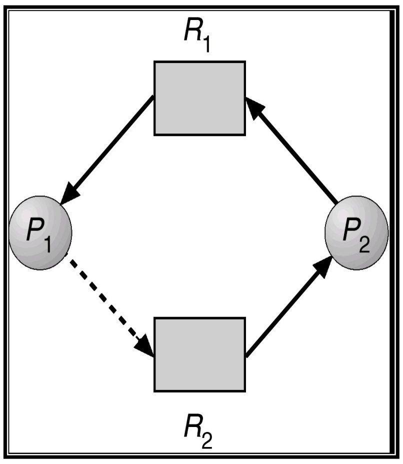
- Suppose Process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the RAG.
- Otherwise, the process must wait till system is safe.

Example



- P1 is holding resource R1 and has a claim on R2
- P2 is requesting R1 and has claim on R2
- Currently, there is no cycle and hence system is in a safe state.

Example



Suppose the request from P2 to R2 is granted and converted to an assignment edge.
This will now result in a cycle taking the system to an unsafe state.

Banker's Algorithm

- Let n = number of processes, and m = number of resources types

Data Structures

- *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].

Banker's Algorithm

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$_i$;

   Allocation$_i$ = Allocation$_i$ + Request$_i$;

   Need$_i$ = Need$_i$ – Request$_i$;

   Call Saftey algorithm.
   - *If safe $\Rightarrow$ the resources are allocated to P_i.*
   - *If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored*

Safety Algorithm

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

   Work = Available

   Finish [i] = *false* for $i = 1,2, \ldots, n$.
2. Find an i such that both:

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

   (b) $Need_i \leq Work$

   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.

Example

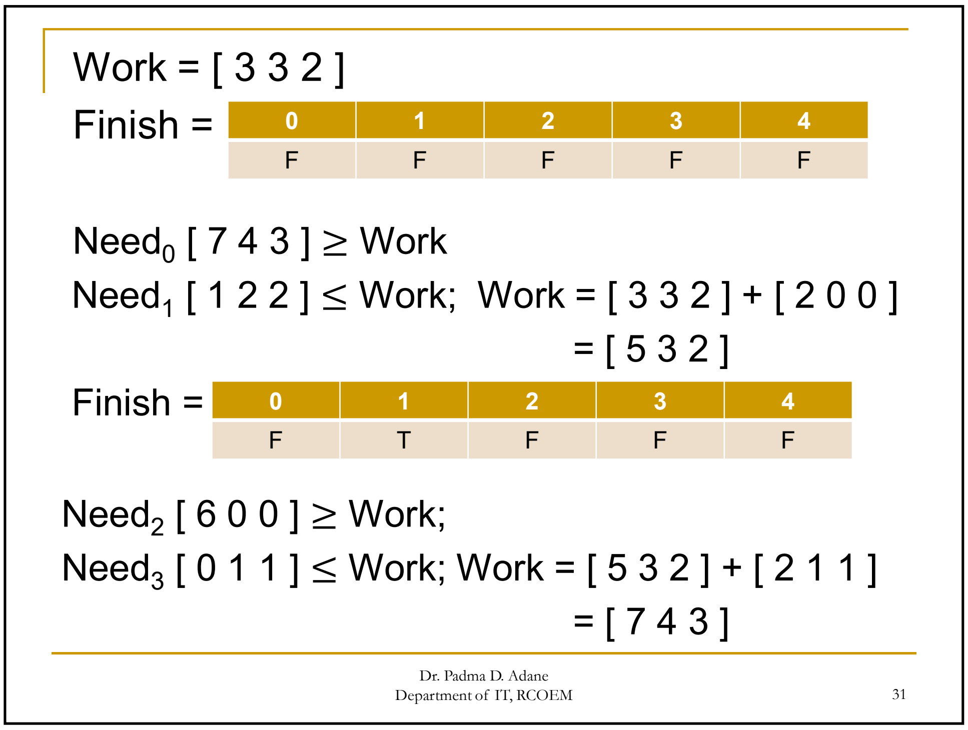
- Let there be 5 processes P_0 through P_4 in the system and 3 resource types *A* with 10 instances, *B* with 5 instances and *C* with 7 instances
- At time t_0 the system state is as follows:

	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	

- From the given data, calculate the Need matrix as Need[i] = Max[i] – Allocation[i]

	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

Is the system in a safe state?



Work = [3 3 2]

Finish =

0	1	2	3	4
F	F	F	F	F

$Need_0$ [7 4 3] ≥ Work

$Need_1$ [1 2 2] ≤ Work; Work = [3 3 2] + [2 0 0]

= [5 3 2]

Finish =

0	1	2	3	4
F	T	F	F	F

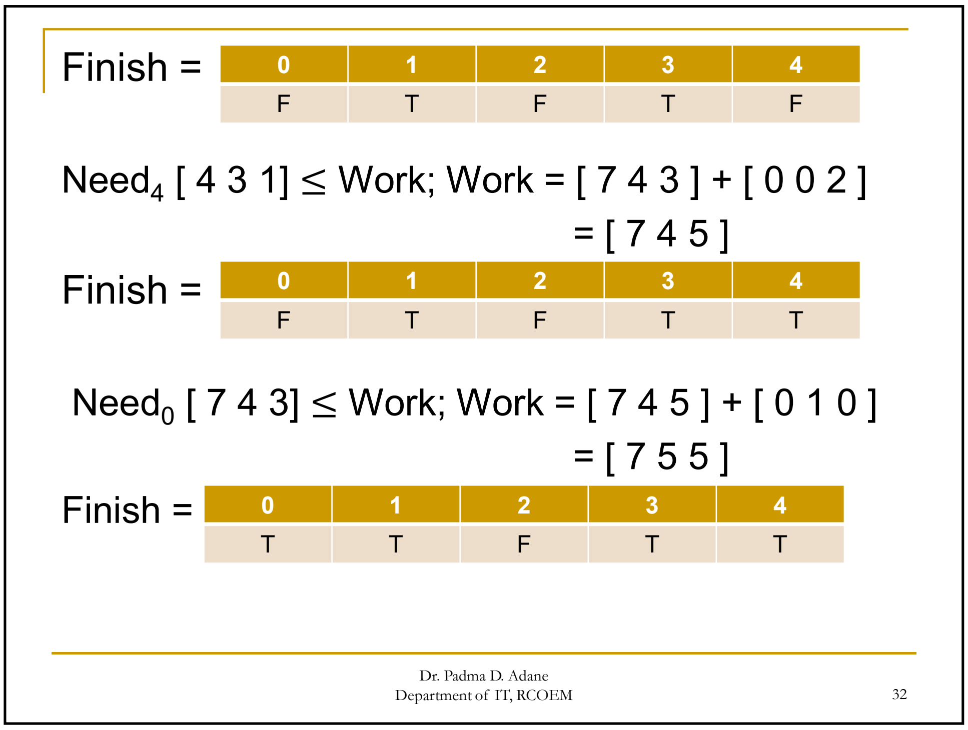
$Need_2$ [6 0 0] ≥ Work;

$Need_3$ [0 1 1] ≤ Work; Work = [5 3 2] + [2 1 1]

= [7 4 3]

Dr. Padma D. Adane
Department of IT, RCOEM

31



Finish =

0	1	2	3	4
F	T	F	T	F

$Need_4$ [4 3 1] ≤ Work; Work = [7 4 3] + [0 0 2]

= [7 4 5]

Finish =

0	1	2	3	4
F	T	F	T	T

$Need_0$ [7 4 3] ≤ Work; Work = [7 4 5] + [0 1 0]

= [7 5 5]

Finish =

0	1	2	3	4
T	T	F	T	T

Dr. Padma D. Adane
Department of IT, RCOEM

32

$Need_2$ [6 0 0] $\leq$ Work; Work = [7 5 5] + [3 0 2]
= [10 5 7]

Finish =

0	1	2	3	4
T	T	T	T	T

The system is safe

Safe Sequence < P_1, P_3, P_4, P_0, P_2>

There can be more than one safe sequence

Example

- Consider again the safe state of the system

	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	

- Can the request from Process P_1 for [1 0 2] be granted?

	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

- Is $\text{Request}_1 \leq \text{Need}_1$ $[1\ 2\ 2] \leq [1\ 0\ 2]$
- Is $\text{request}_1 \leq$ Available $[1\ 2\ 2] \leq [3\ 3\ 2]$
- *Available = Available – $\textit{Request}_1$*
  - *Available = [3 3 2] – [1 0 2] = [2 3 0]*
- *$\textit{Allocation}_1 = \textit{Allocation}_1 + \textit{Request}_1$*
  - *$\textit{Allocation}_1$* = [2 0 0] + [1 0 2] = [3 0 2]
- *$\textit{Need}_1 = \textit{Need}_1 – \textit{Request}_1$*
  - *$\textit{Need}_1$* = [1 2 2] - [1 0 2] = [0 2 0]

	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 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Check if the system is safe.

Sequence <P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.

Additional Requests

- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

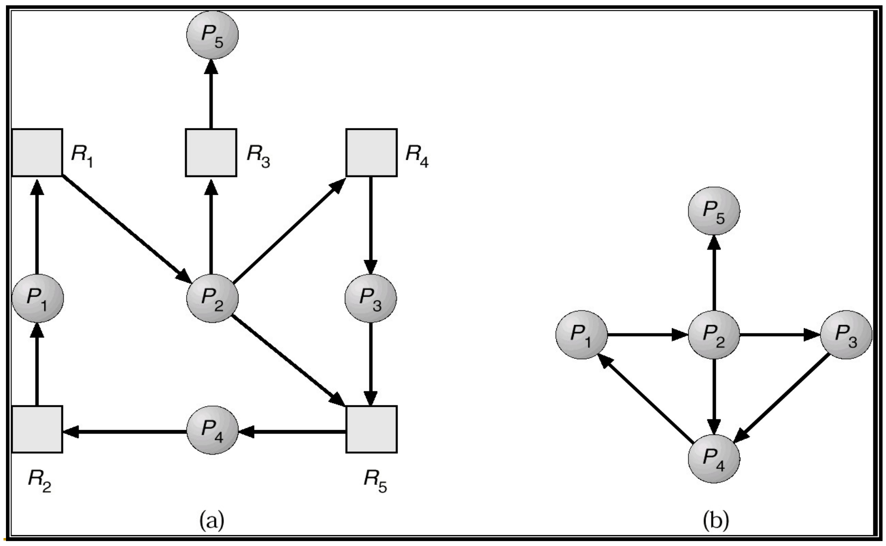
Deadlock Detection

- Certain systems do not take precautionary steps and let deadlock happen; it runs deadlock detection algorithm periodically to detect it and then recover from it.
- For all resources having only a single instance, detection algorithm uses a variant of resource allocation graph, called as wait-for graph.
- For resources having several instances, the algoithm uses several time varying data structures similar to the banker's algorithm.

Single Instance of Each Resource Type

- In a wait-for graph, an edge from Pi to Pj implies that process Pi is waiting for a resource held by process Pj.
- An edge Pi → Pj exists in a wait-for graph if and only if the corresponding Resource-allocation graph has two edges Pi → R_q and R_q → Pj for some resource R_q.
- A deadlock exists in the system if and only if the wait-for graph contains a cycle.
- 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 equivalent Wait-for graph



Several Instances of a Resource Type

- Data structures used
  - *Available:* A vector of length *m* indicating the number of available resources of each type.
  - *Allocation:* An *n* x *m* matrix defining the number of resources of each type currently allocated to each process.
  - *Request:* An *n* x *m* matrix indicating 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.

Deadlock 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.

Deadlock Detection Algorithm

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

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($m \times n^2$) operations to detect whether the system is in deadlocked state.

Example

- 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*.

Example

- *Let* 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	

Is the system in deadlock?

- Deadlock exists and processes P_1, P_2, P_3, and P_4 are in deadlock

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 detection algorithm is invoked after every resource request that cannot be granted immediately, it is possible to find the process leading to deadlock along with the processes involved in the deadlock; however, this results in considerable overhead in computation time.
- If detection algorithm is invoked arbitrarily, say, once every hour or whenever CPU utilization falls below 40%, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

System Recovery from Deadlock

- Two Alternatives
  - Process Termination
  - Resource Preemption
- Process Termination
  - Abort all deadlocked processes: An expensive option as the processes may have computed for a long time.
  - Abort one process at a time until the deadlock cycle is eliminated: Incurs lot of overhead as after aborting each process detection algorithm needs to be run to check if the deadlock is over.

System Recovery from Deadlock

- 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?

System Recovery from Deadlock

- Resource Preemption: Three issues needs to be addressed while dealing with deadlock using resource preemption
  - Selecting a victim: Which Process and which resource to preempt? One with minimum cost; in terms of how many resources the process is holding? how much work has been done? how mush work is left?
  - Rollback: Restart process; a costly affair, return to some safe state; requires system to record lot of information about each process.
  - Starvation: Same process may always be picked as victim; include number of rollback in cost factor.

Combined Approach to Deadlock Handling

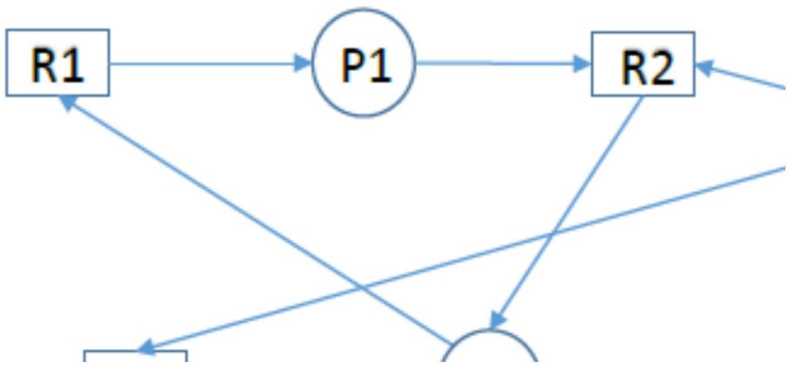
- Combine the three basic approaches
  - prevention
  - avoidance
  - detection

allowing the use of the optimal approach for each of resources in the system.

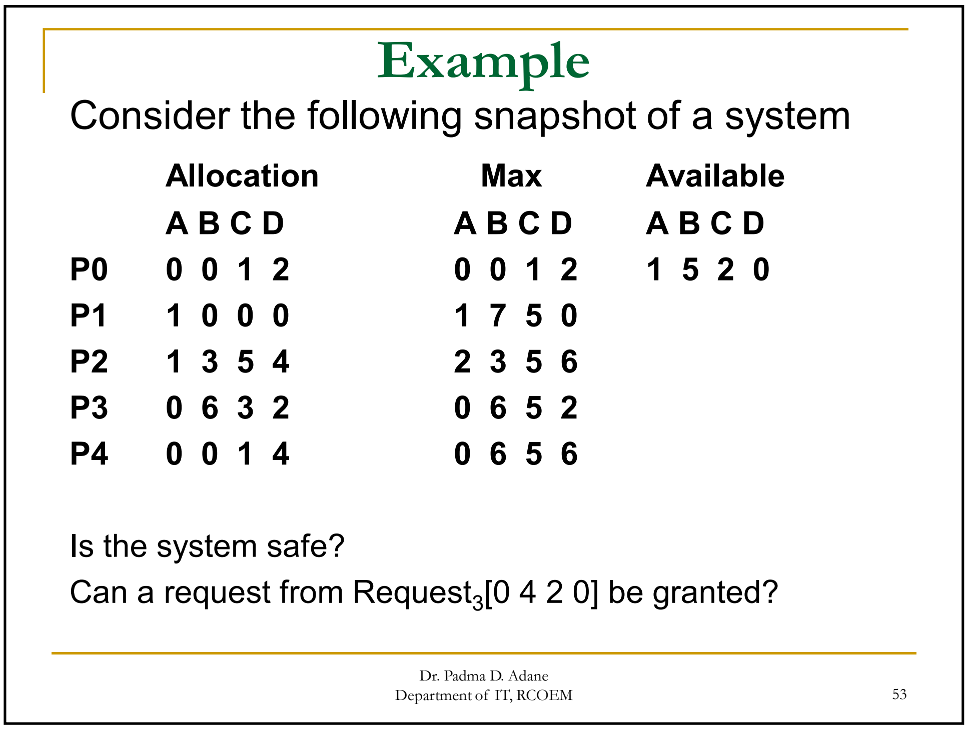
- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.

Example

- Is the system, represented by the RAG given below, in deadlock?



- P1 and P3 are deadlocked because a cycle exists between them. P2 is also deadlocked as all resources required by it and held by P3 which will not release them since it is deadlocked.



Example

Consider the following snapshot of a system

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

Is the system safe?

Can a request from Request_3[0 4 2 0] be granted?