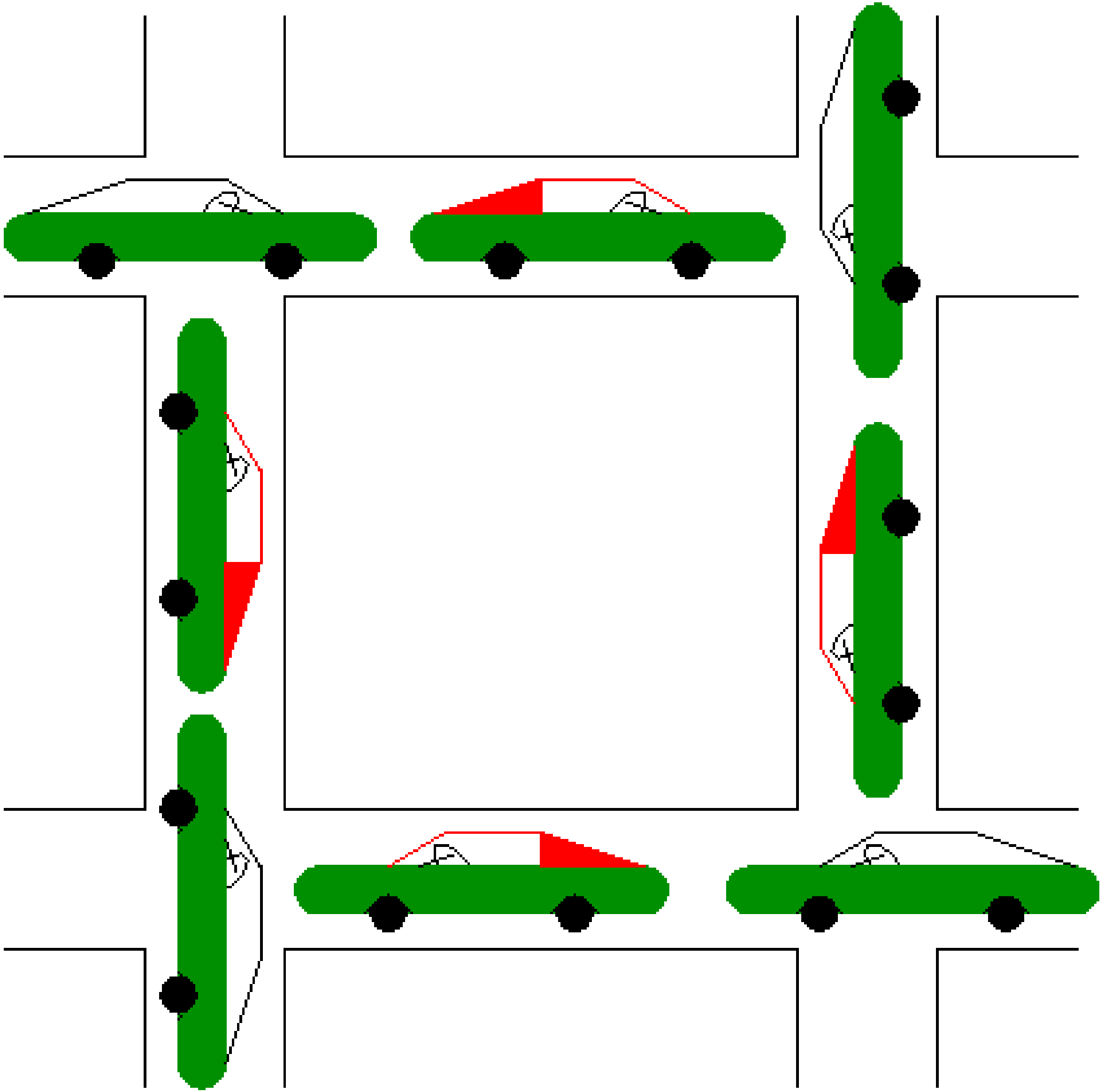
DEADLOCK

Operating System (CSC1036 & INF1036)

Dr. Hasin A. Ahmed
Assistant Professor
Department of Computer Science
Gauhati University



Deadlock

- Computer systems are full of resources that can only be used by one process at a time.
- Common examples include printers and tape drives
- Having two processes simultaneously writing to the printer leads to nonsense.
- Consequently, all operating systems have the ability to (temporarily) grant a process exclusive access to certain resources, both hardware and software.
- For many applications, a process needs exclusive access to not one resource, but several

Deadlock

- Suppose, for example, two processes each want to record a scanned document on a CD.
- Process A requests permission to use the scanner and is granted.
- Process B is programmed differently and requests the CD recorder first and is also granted.
- Now A asks for the CD recorder, but the request is denied until B releases it.
- Unfortunately, instead of releasing the CD recorder B asks for the scanner.
- At this point both processes are blocked and will remain so forever.
- This situation is called a deadlock.

Deadlock: Definition

- *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*

Deadlock

- *Deadlocks can occur in a variety of situations besides requesting dedicated I/O devices.*
- *In a database system, for example, a program may have to lock several records it is using, to avoid race conditions.*
- *If process A locks record R1 and process B locks record R2, and then each process tries to lock the other one’s record, we also have a deadlock.*
- *A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a database)*
- *A resource is anything that can be used by only a single process at any instant of time.*

Deadlock: necessary conditions

- *Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock. All conditions must be hold.*
- *Mutual exclusion condition: Each resource is either currently assigned to exactly one process or is available.*
- *Hold and wait condition: Processes currently holding resources that were granted earlier can request new resources.*
- *No preemption condition: Resources previously granted cannot be forcibly taken away from a process.*
- *Circular wait condition: There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.*

Resource Alloation Graph

- *The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares*
- *An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process*
- *An arc from a process to a resource means that the process is currently blocked waiting for that resource.*
- *A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind)*

Resource Allocation Graph

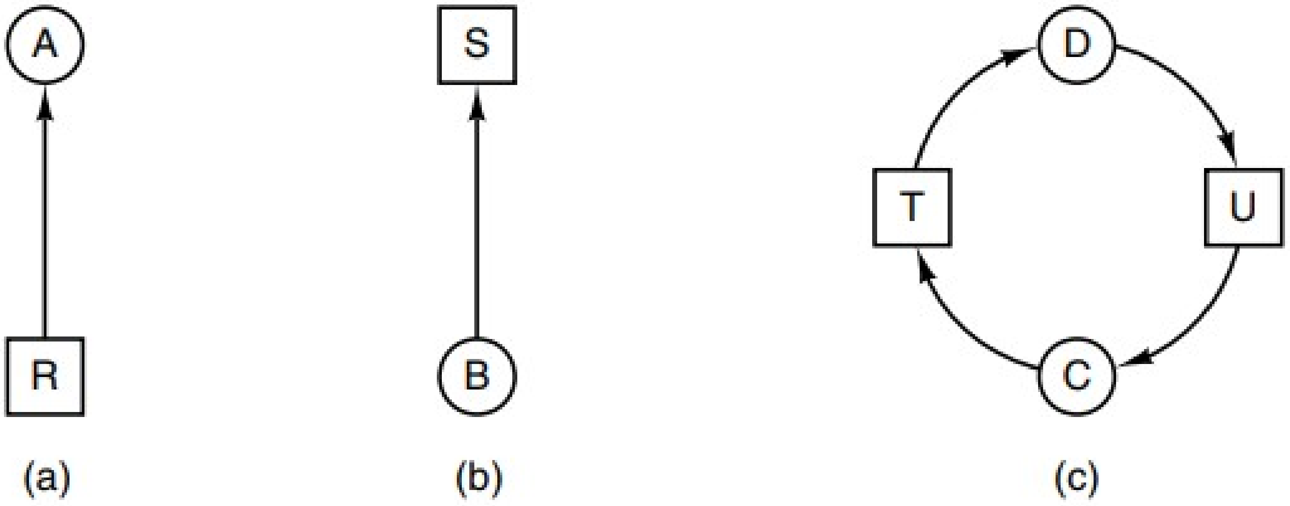


Figure 3-9. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

Dealing with Deadlock

1) *Just ignore the problem altogether.*

2) *Detection and recovery. Let deadlocks occur, detect them, and take action.*

3) *Dynamic avoidance by careful resource allocation.*

4) *Prevention, by structurally negating one of the four conditions necessary to cause a deadlock.*

The Austrich Algorithm

- *Stick your head in the sand and pretend there is no problem at all.*
- *(Ostriches can run at 60 km/hour and their kick is powerful enough to kill any lion with visions of a big chicken dinner.)*
- *If deadlocks occur on the average once every five years, but system crashes due to hardware failures, compiler errors, and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks.*

Deadlock Prevention

- This technique tries to ensure that at least one of the necessary conditions stated by Coffman et al. is not fulfilled.
- **Addressing Mutual exclusion:** Mutual exclusion condition will be fulfilled for non-sharable resources. A sharable resource can never fulfil the mutual exclusion condition, hence will not be part of dealock.
- But we cannot prevent deadlock by denying mutual exclusion because some resources are intrinsically non-sharable.
- **Addressing Hold and Wait:** If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks.

Deadlock Prevention

- One way to achieve this goal is to require all processes to request all their resources before starting execution.
- If everything is available, the process will be allocated whatever it needs and can run to completion.
- If one or more resources are busy, nothing will be allocated and the process would just wait.
- An immediate problem with this approach is that many processes do not know how many resources they will need until after they have started running.
- Processes will keep resources engaged for a long time, reducing efficiency

Deadlock Prevention

- A slightly different way to break the hold-and-wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds.
- Then it tries to get everything it needs all at once.
- Addressing no-preemption: If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted.
- In other words, these resources are implicitly released.
- The preempted resources are added to the list of resources for which the process is waiting.
- The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Deadlock Prevention

- Alternatively, if a process requests some resources, we first check whether they are available.
- If they are, we allocate them.
- If they are not, we check whether they are allocated to some other process that is waiting for additional resources.
- If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.
- If the resources are neither available nor held by a waiting process, the requesting process must wait.
- While it is waiting, some of its resources may be preempted, but only if another process requests them.

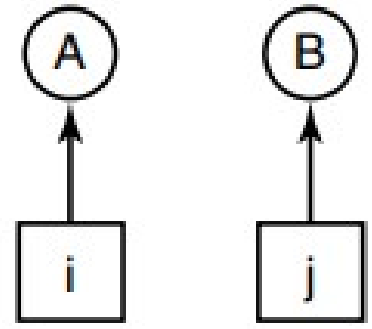
Deadlock Prevention

- **Addressing Circular wait**: One way to avoid the circular wait is to provide a global numbering of all the resources
- Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order.
- If we go by the following numbering, a process may request first a scanner and then a tape drive, but it may not request first a plotter and then a scanner.
- 1. Imagesetter
- 2. Scanner
- 3. Plotter
- 4. Tape drive
- 5. CD Rom drive

Deadlock Prevention

- With this rule, the resource allocation graph can never have cycles
- Let us see why this is true for the case of two processes

- We can get a deadlock only if A requests resource j and B requests resource i.
- If i > j, then A is not allowed to request j because that is lower than what it already has.
- If i < j, then B is not allowed to request i because that is lower than what it already has.
- Either way, deadlock is impossible.



Deadlock Prevention

- With multiple processes, the same logic holds.
- At every instant, one of the assigned resources will be highest.
- The process holding that resource will never ask for a resource already assigned.
- It will either finish, or at worst, request even higher numbered resources, all of which are available.
- Eventually, it will finish and free its resources.
- At this point, some other process will hold the highest resource and can also finish.
- In short, there exists a scenario in which all processes finish, so no deadlock is present.

Deadlock Prevention

- Although numerically ordering the resources eliminates the problem of deadlocks, it may be impossible to find an ordering that satisfies everyone.
- Moreover, a perfectly good and available copy of a resource could be inaccessible with such a rule

Deadlock Avoidance

- Deadlock can avoided not by imposing arbitrary rules on processes but by carefully analyzing each resource request to see if it could be safely granted.
- The question arises: is there an algorithm that can always avoid deadlock by making the right choice all the time?
- The answer is a qualified yes, but only if certain information is available in advance.
- We will discuss about two deadlock avoidance algorithms that can handle two different situations

Deadlock Avoidance

- 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.
- More formally! a system is in a safe state only if there exists a safe sequence.
- A safe state is not a deadlocked state.
- Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however.
- An unsafe state may lead to a deadlock.

Deadlock Avoidance

- **Let us try to understand the concept of 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.***

Deadlock Avoidance

- To illustrate, let us consider a system with 12 magnetic tape drives and three processes: P0, P1, and P2 .
- Process P0 requires 10 tape drives, process P1 may need as many as 4 tape drives, and process P2 may need up to 9 tape drives.
- Suppose that, at time t_0, process P0 is holding 5 tape drives, process P1 is holding 2 tape drives, and process P2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

	Maximum Needs	Currently Holds
P0	10	5
P1	4	2
P2	9	2

Deadlock Avoidance

- ***Is the system safe at t_0***

Deadlock Avoidance

- ***The answer is yes, the system is in safe state***

- The sequence < P1, P0, P2 > satisfies the safety condition
- Initial tape drives: 3
- After execution of P1: 5 (+2)
- After execution of P0: 10 (+5)
- After execution of P2: 12 (+2)

	Maximum Needs	Currently Holds
P0	10	5
P1	4	2
P2	9	2

Deadlock Avoidance

- ***Now suppose t_1 process P2 requests and is allocated one more tape drive***

- At this point, only process P1 can be allocated all its tape drives.
- When it returns them, the system will have only 4 available tape drives, not sufficient for rest
- So it is an unsafe state

	Maximum Needs	Currently Holds
P0	10	5
P1	4	2
P2	9	3

Deadlock Avoidance

- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system "will never deadlock.
- The idea is simply to ensure that the system will always remain in a safe state.
- Initially, the system is in a safe state.
- Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
- The request is granted only if the allocation leaves the system in a safe state.

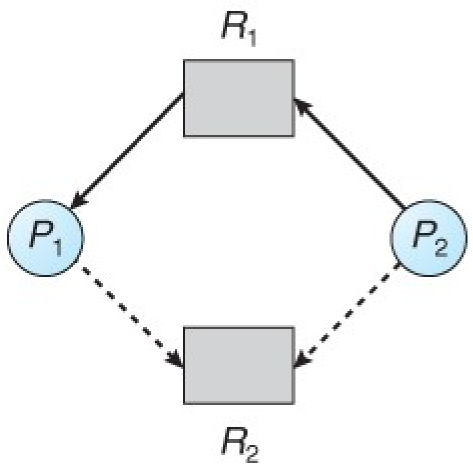
Deadlock Avoidance

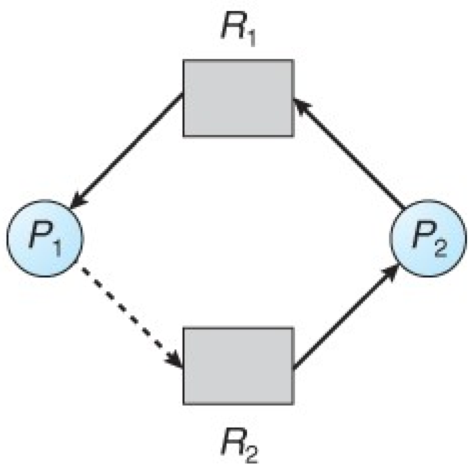
- In this scheme, if a process requests a resource that is currently available, it may still have to wait.
- Thus, resource utilization may be lower than it would otherwise be.
- We are going to discuss about two algorithms for deadlock avoidance
- First one is Resource allocation graph based algorithm which is applied on a situation where all resources have single instance
- Second one is Banker’s algorithm whic is used for a system with multi instance resources

Resource Allocation Graph based Algorithm

- If we have a resource-allocation system with only one instance of each resource type, then this algorithm can be used.
- It uses a variant of the resource-allocation graph for deadlock avoidance.
- In addition to the request and assignment edges of traditional resource allocation graph, a new type of edge called a claim edge is introduced in this variant.
- A claim edge Pi -> R; indicates that process Pi may request resource Rj at some time in the future.
- This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- When process Pi requests resource Rj the claim edge Pi -> R j is converted to a request edge.

Resource Allocation Graph based Algorithm



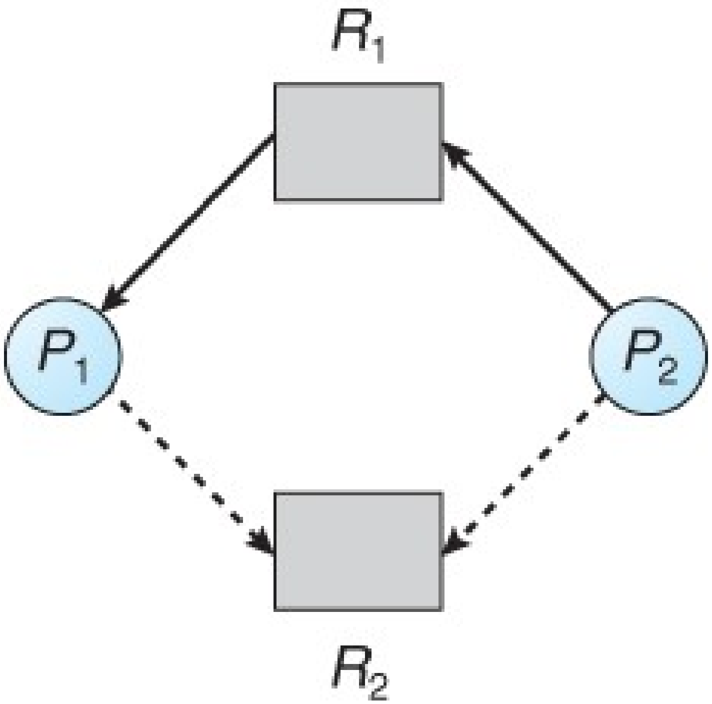


Resource Allocation Graph based Algorithm

- Suppose that process Pi requests resource R;. The request can be granted only if converting the request edge Pi -> Rj to an assignment edge Rj -> Pi does not result in the formation of a cycle in the resource-allocation graph.
- Note that we check for safety by using a cycle-detection algorithm.
- An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.
- Presence of a cycle will indicate that the system is in unsafe state, and hence the allocation will be delayed.

Resource Allocation Graph based Algorithm

- Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph



Banker's Algorithm

- The resource-allocation-graph based algorithm is not applicable to a resource allocation system with multiple instances of each resource type.
- In such situations an algorithm named Banker's algorithm is used.
- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Banker’s Algorithm

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
- This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Banker's Algorithm

- Assuming n number of processes and m number of resources, following data structures are used to encode the resource allocation system
- Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, there are k instances of resource type R_j available.
- Max: An nxm matrix defines the maximum demand of each process. If Max[i][j] equals k, then process P_i may request at most k instances of resource type R_j.

Banker's Algorithm

- Allocation: An nxm matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals k, then process P_i is currently allocated k instances of resource type R_j.
- Need: An nxm matrix indicates the remaining resource need of each process. If Need[i][j] equals k then process P_i may need k more instances of resource type R_i to complete its task. Note that Need[i][j] equals Max[i][j]- Allocation[i][j].

Banker's Safety Algorithm

1) 1) Let *Work* and *Finish* be vectors of length *m* mid *n*, respectively. Initialize *Work* = *Available* and *Finish[i]* =*false* for *i* = 0, 1, ..., n – 1.

2) 2) Find an i such that both

- *Finish[i]* == *false*
- $Need_i$ <= *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, otherwise it is unsafe.

Banker's Safety Algorithm

1) This algorithm may require an order of $m \times n^2$ operations to deternline whether a state is safe.

Banker's Resource-Request Algorithm

1) If $Request_i$ is <= $Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i$ <= Available, go to step 3. Otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows

- Available = Available - $Request_i$;
- $Allocation_i$ = $Allocation_i$ + $Request_i$;
- $Need_i$ =$Need_i$ – $Request_i$;

4) If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources.

5) However, if the new state is unsafe, then P_i must wait for $Request_i$, and the old resource-allocation state is restored.

Banker's Algorithm

Process	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	

Banker's Algorithm

Process	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			

Process	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

Banker's Algorithm

m=3, n=5

Step 1 of Safety Algo

Work = Available

Work =

3	3	2

	0	1	2	3	4
Finish =	false	false	false	false	false

Process	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

Banker's Algorithm

For i = 0 Step 2

$Need_0$ = 7, 4, 3

7,4,3 3,3,2

Finish [0] is false and $Need_0$ > Work

So P_0 must wait

But Need ≤ Work

For i = 1 Step 2

$Need_1$ = 1, 2, 2

1,2,2 3,3,2

Finish [1] is false and $Need_1$ < Work

So P_1 must be kept in safe sequence

Step 3

3, 3, 2 2, 0, 0

Work = Work + $Allocation_1$

	A	B	C
Work =	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

Process	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

Banker's Algorithm

For i = 2 Step 2

6, 0, 0 5,3, 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2$ > Work

So P_2 must wait

For i=3 Step 2

0, 1, 1 5, 3, 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3$ < Work

So P_3 must be kept in safe sequence

Step 3

5, 3, 2 2, 1, 1

Work = Work + $Allocation_3$

Work =

A	B	C
7	4	3

Finish =

0	1	2	3	4
false	true	false	true	false

Process	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

Banker's Algorithm

For i = 4 Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $\underbrace{Need_4}_{4,3,1} < \underbrace{Work}_{7,4,3}$

So P_4 must be kept in safe sequence

Step 3

$Work = \underbrace{Work}_{7,4,3} + \underbrace{Allocation_4}_{0,0,2}$

Work =

A	B	C
7	4	5

Finish =

0	1	2	3	4
false	true	false	true	true

Process	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

Banker's Algorithm

For i = 0
$Need_0$ = 7, 4, 3
Finish [0] is false and Need < Work (7, 4, 3 < 7, 4, 5)
So P_0 must be kept in safe sequence

Step 2

Work = Work + $Allocation_0$ (7, 4, 5 + 0, 1, 0)

Work =

A	B	C
7	5	5

Finish =

0	1	2	3	4
true	true	false	true	true

Step 3

Process	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

Banker's Algorithm

For i = 2 Step 2

$Need_2 = 6, 0, 0$

Finish [2] is false and $\overset{6,\,0,\,0}{Need_2} < \overset{7,\,5,\,5}{Work}$

So P_2 must be kept in safe sequence

Step 3

$Work = \overset{7,\,5,\,5}{Work} + \overset{3,\,0,\,2}{Allocation_2}$

Work =

A	B	C
10	5	7

Finish =

0	1	2	3	4
true	true	true	true	true

Finish [i] = true for $0 \le i \le n$ Step 4

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Process	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

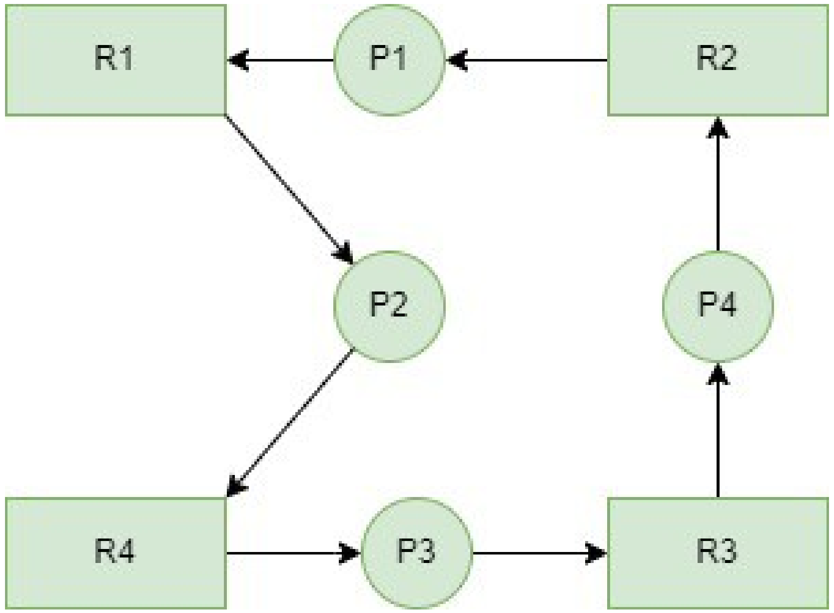
Detection and Recovery

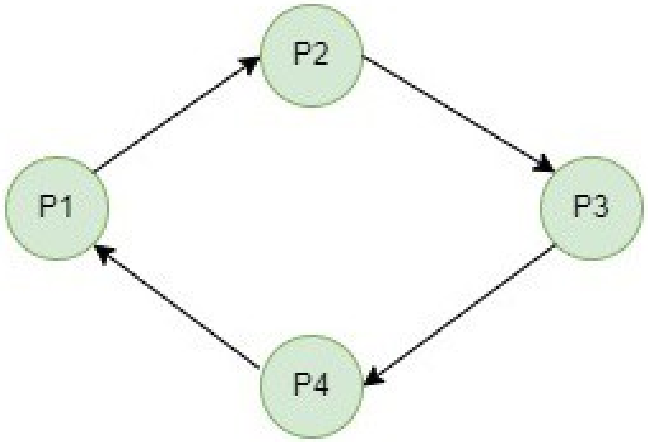
- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm then a deadlock situation may occur. In this environment, the system must provide:
- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Detection and Recovery

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.
- We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.
- An edge Pi -> Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi -> R and R-> Pj for some resource R

Detection and Recovery





Detection and Recovery

- As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle.
- To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.
- 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.
- On detection of deadlock processes are killed to recover the system.

Detection and Recovery

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
- We turn now to a deadlock detection algorithm that is applicable to such a system.
- The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm
  - Available
  - Allocation
  - Request

Detection and Recovery

- 1) Let Work and Finish be vectors of length m and n, respectively. Initialize Work=Available. For i=0, 1, ..., n-1, if $Request_i$!= 0, then Finish[i]=false; otherwise, Finish[i]=true
- 2) Find an index i such that both
- a. Finish[i]== false
- b. $Request_i$<= 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]=false, for some i, 0<= i < n, then the system is in a deadlocked state. Moreover, if Finish[i]=false, then process Pi is deadlocked.

Detection and Recovery

- This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.
- When should we invoke the detection algorithm?
- Of coursc, if the deadlock-detection algorithm is invoked for every resource request, this will incur a considerable overhead in computation time.
- A less expensive alternative is simply to invoke the algorithm at less frequent intervals for example, once per hour or whenever CPU utilization drops below 40 percent.

thank you

???