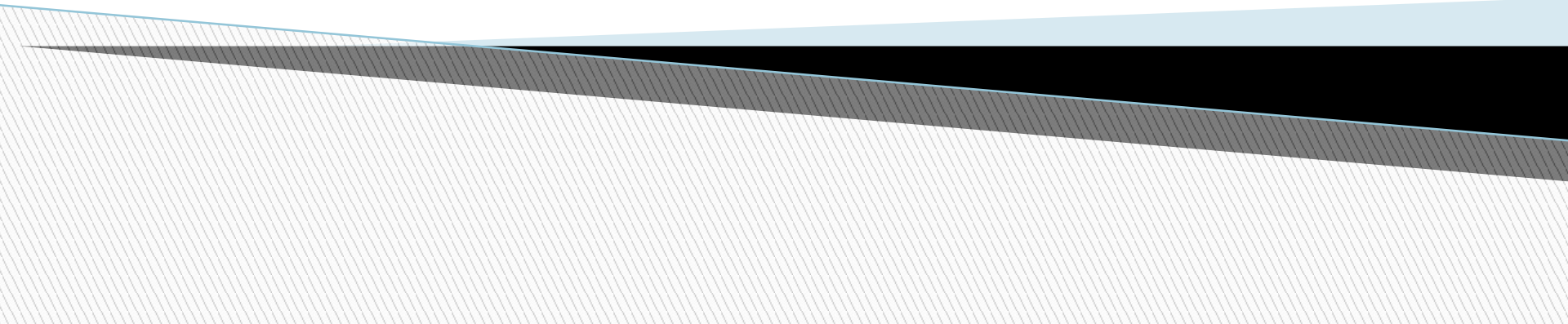
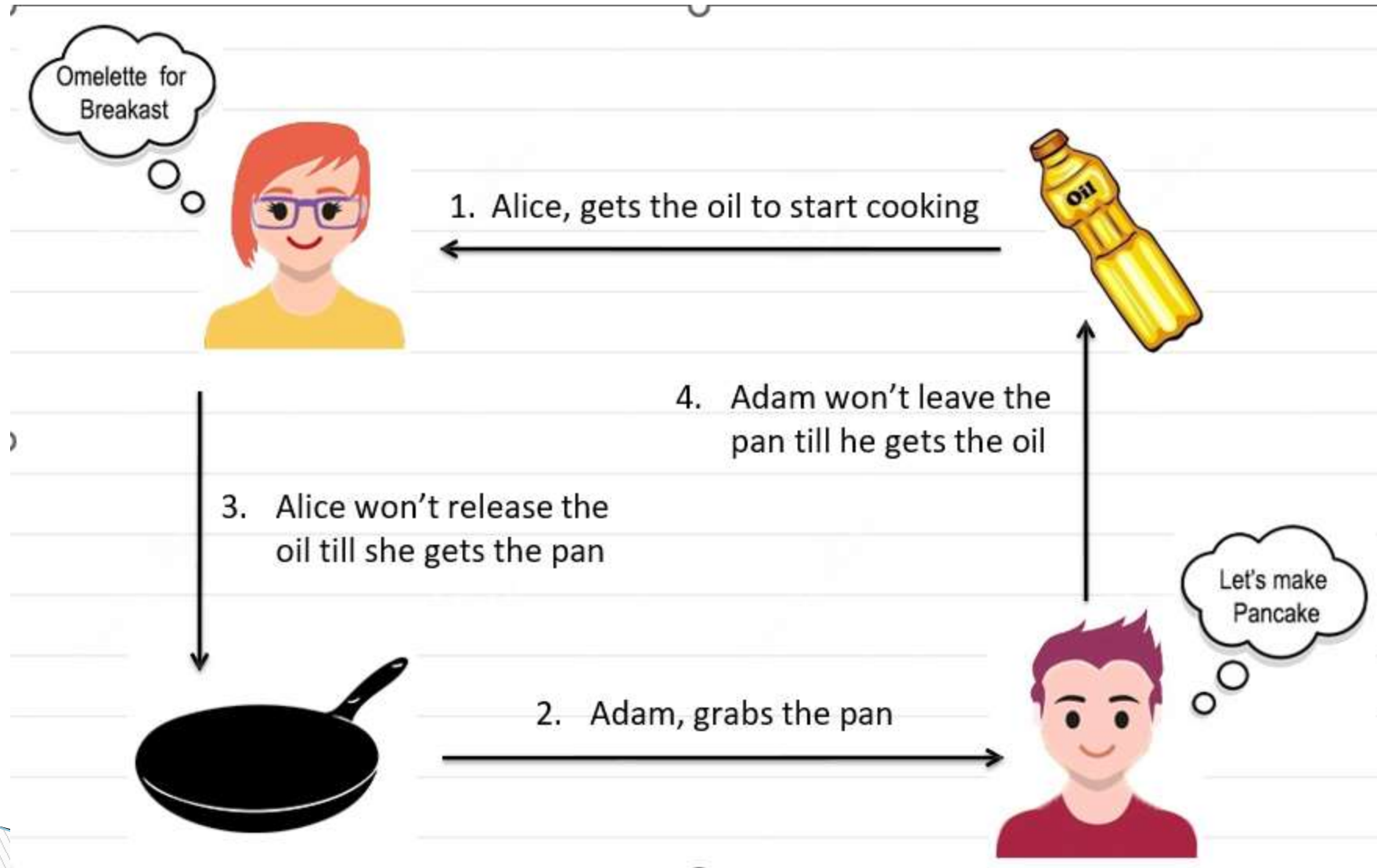


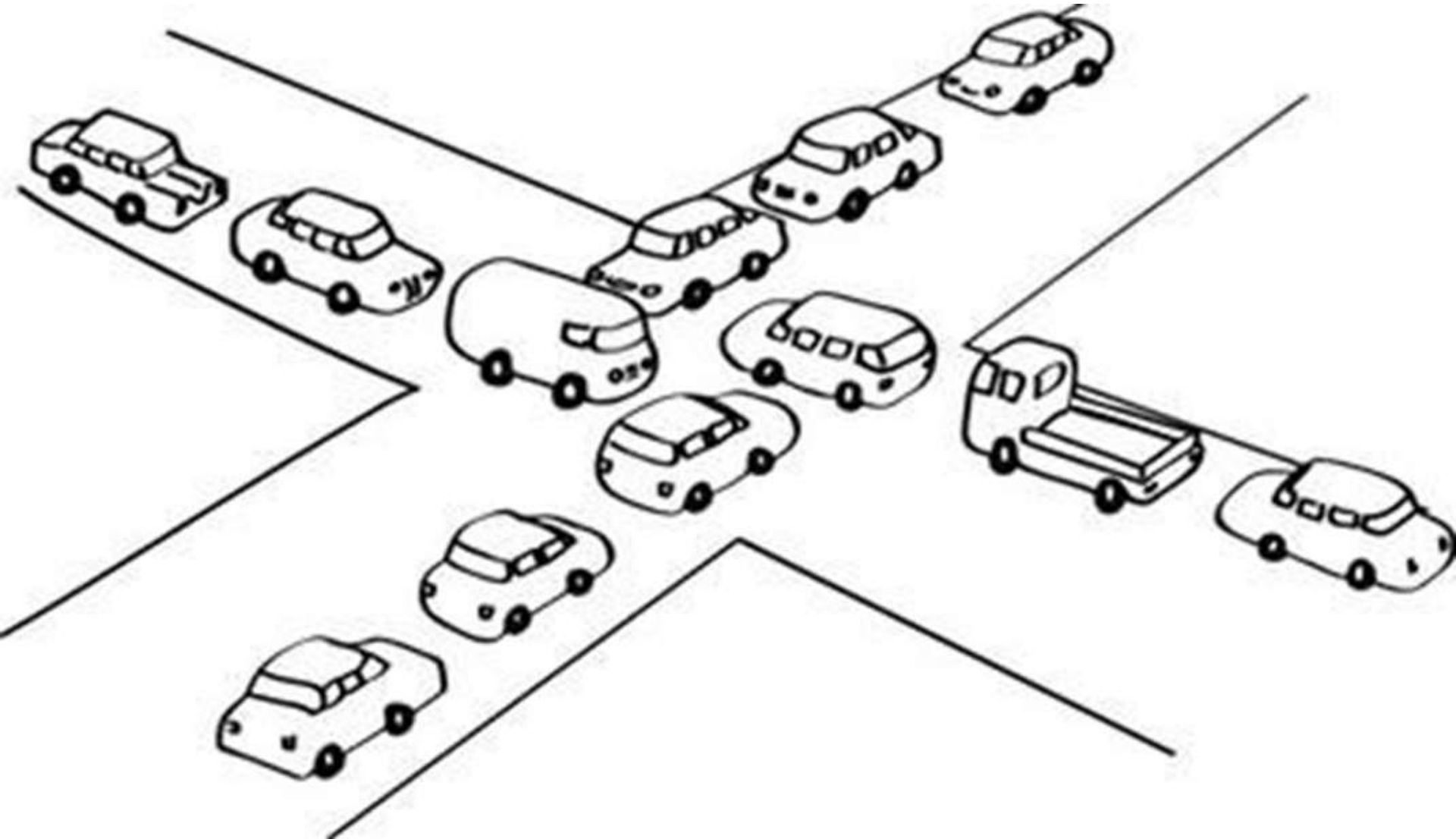
Deadlocks



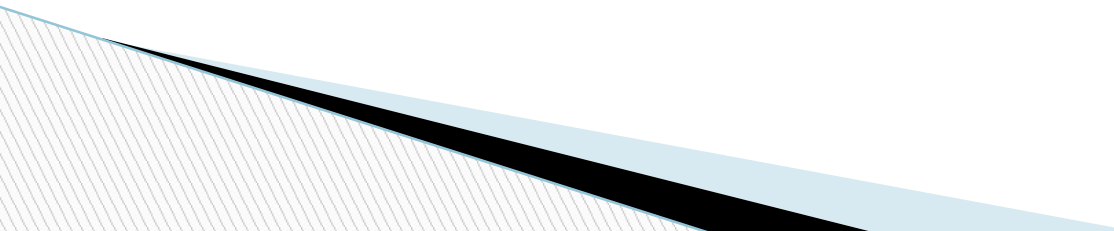
Deadlock in Real Life



Another Example



Deadlock Introduction

- In a multiprogramming environment, several processes may compete for a finite number of resources.
 - A process requests resources, and 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.
 - This situation is called a **DEADLOCK**
- 

Resources

Two types of resources:

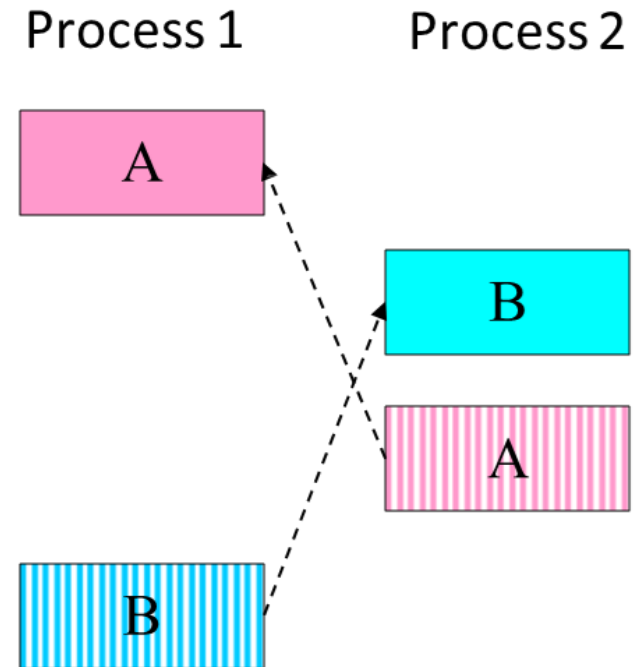
- Preemptable resources**: can be taken away from a process with no ill effects.
- Non-preemptable resources**: will cause the process to fail if taken away.

What is Deadlock?

- **Definition:** A set of processes is *deadlocked* when they are sharing the same set of resources and preventing each other from accessing the resource, resulting in both programs ceasing to function.
- Process Deadlock
 - A process is deadlocked when it is waiting on an event that will never happen.
- System Deadlock
 - A system is deadlocked when one or more processes are deadlocked

When Do Deadlocks Happen?

- Suppose
 - Process 1 holds resource A and requests resource B
 - Process 2 holds B and requests A
 - Both can be blocked, with neither able to proceed
- Deadlocks occur when ...
 - Processes are granted **exclusive access** to devices or software constructs (resources)
 - Each deadlocked process needs a resource held by another deadlocked process

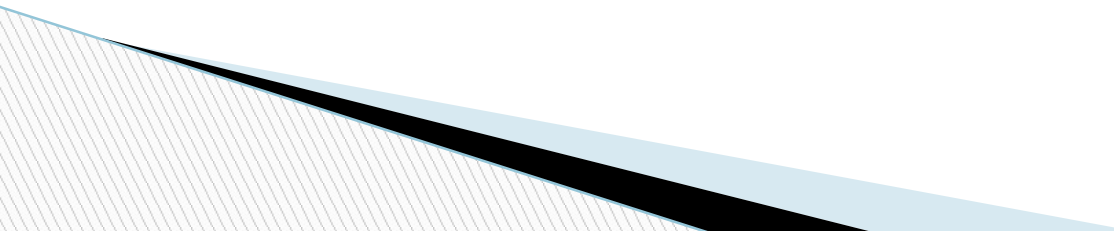


DEADLOCK!

Using Resources

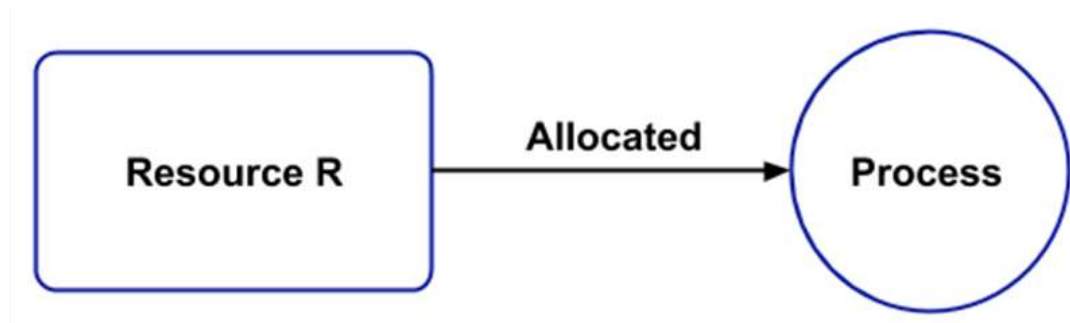
- Sequence of events required to use a resource:
 - Request the resource
 - Use the resource
 - Release the resource
- Can't use the resource if request is denied.
 - Requesting process has options:
 1. Block and wait for resource
 2. Continue (if possible) without it: may be able to use an alternate resource
 3. Process fails with error code
 - Some of these may be able to **prevent deadlock...**

Necessary Conditions for a Deadlock

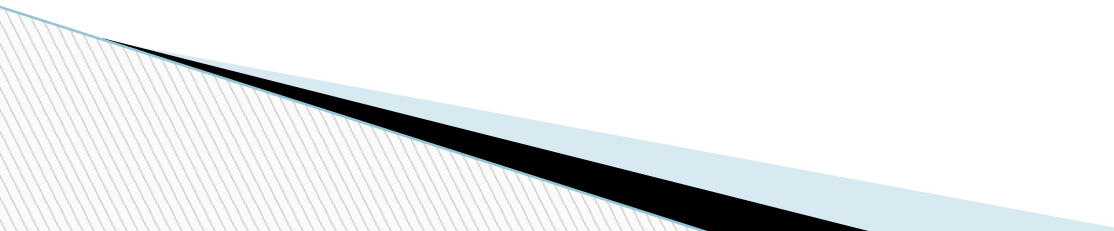
- Mutual Exclusion
 - Hold & Wait
 - No Pre-emption
 - Circular Wait
- 

Mutual Exclusion

- A resource can be held by only one process at a time.
- In other words, if a process P1 is using some resource R at a particular instant of time, then some other process P2 can't hold or use the same resource R at that particular instant of time.

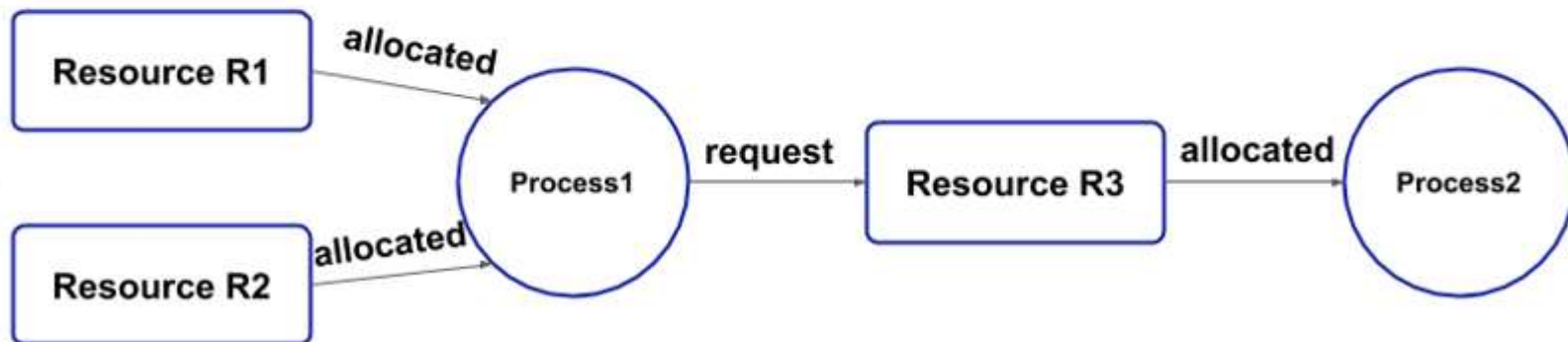


Mutual Exclusion cont..

- At least one resource must be held in a **non-sharable mode**; that is, only one process at a time can use the resource.
 - If another process requests that resource, the requesting process must be delayed until the resource has been released.
- 

Hold & Wait

- A process can hold a number of resources at a time and at the same time, it can request for other resources that are being held by some other process.
- For example, a process P1 can hold two resources R1 and R2 and at the same time, it can request some resource R3 that is currently held by process P2.

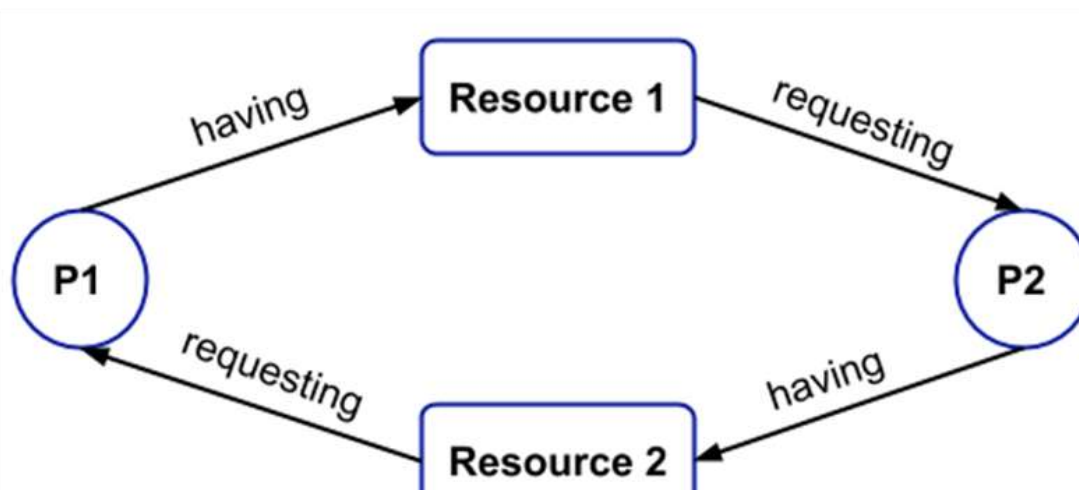


No Pre-emption

- Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it after that process has completed its task. i.e., a resource can't be taken back from the process by another process, forcefully.
- For example, if process P1 is using some resource R, then some other process P2 can't forcefully take that resource.

Circular Wait

- Circular wait is a condition when the first process is waiting for the resource held by the second process, the second process is waiting for the resource held by the third process, and so on.
- At last, the last process is waiting for the resource held by the first process.
- So, every process is waiting for each other to release the resource and no one is releasing their own resource.



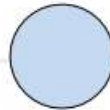
Resource-Allocation Graph

Deadlock can be recognized using a tool named “resource allocation and request graph” or RARG

- Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E .
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- Edges are of two types:
 - **request edge**: directed edge $P_i \longrightarrow R_j$
 - **assignment edge**: directed edge $R_j \longrightarrow P_i$

Resource-Allocation Graph Notations

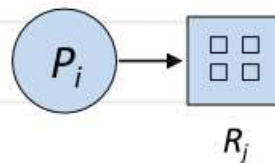
- Process



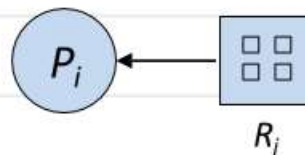
- Resource Type with 4 instances



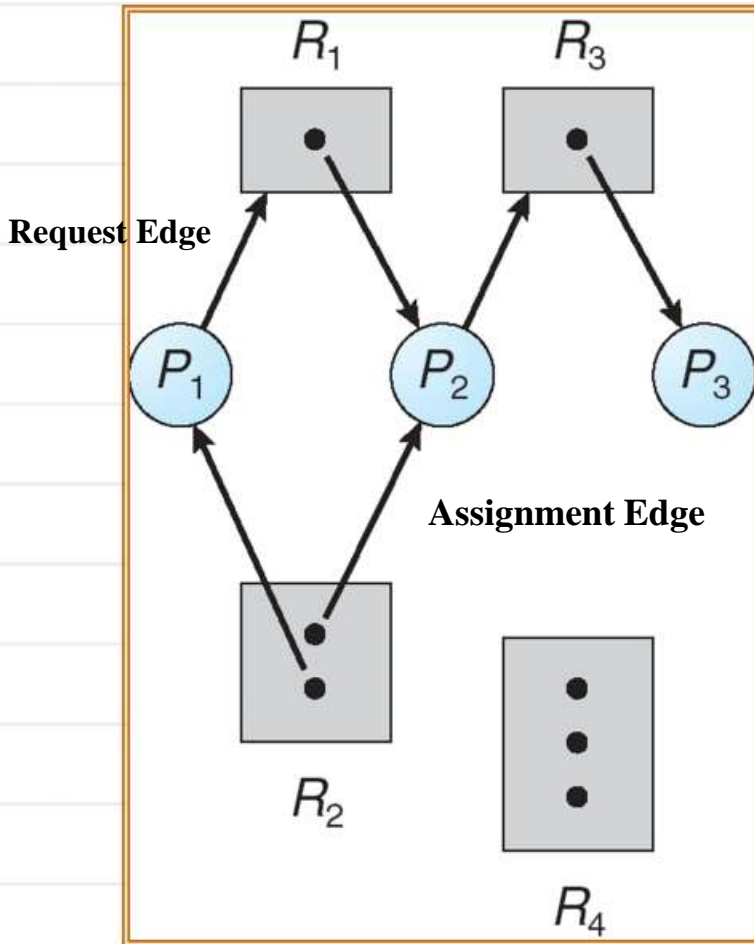
- P_i requests instance of R_j



- P_i is holding an instance of R_j

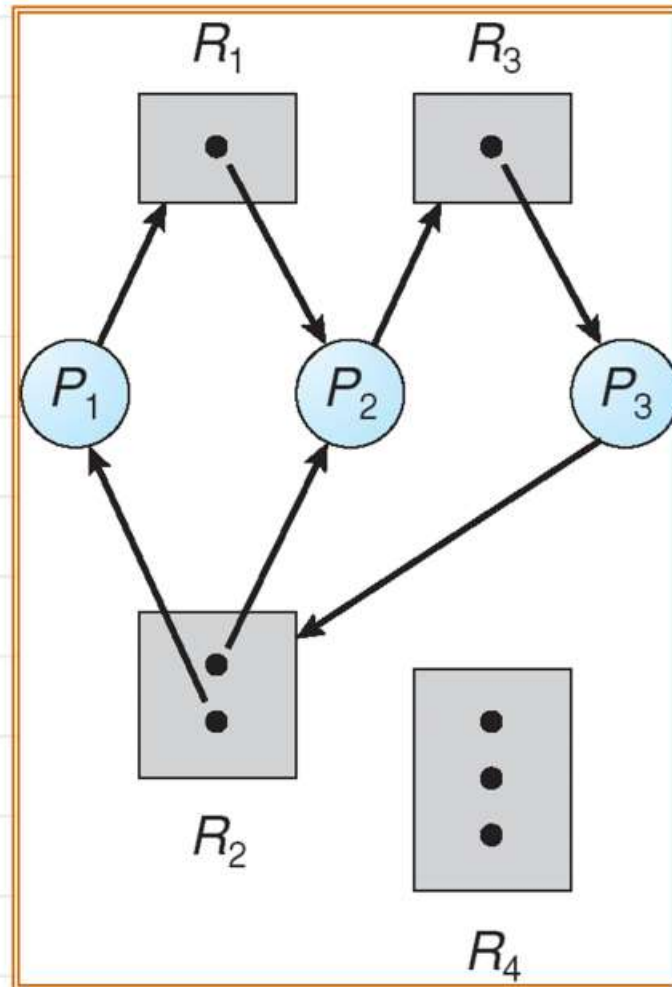


Example of a Resource Allocation Graph



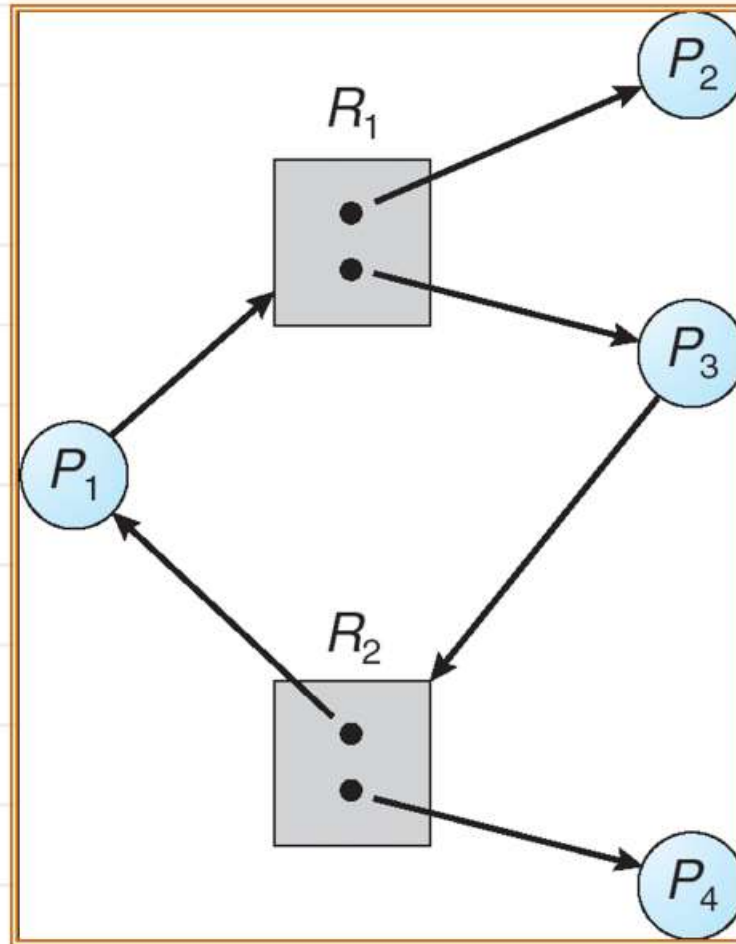
- The presence or absence of cycles within this graph can indicate the potential for deadlock.
- No cycle \Rightarrow No deadlock
- Cycle with single instance per resource \Rightarrow **Deadlock**
- Cycle with multiple instances per resource \Rightarrow Possibility of deadlock

Resource Allocation Graph with a Deadlock




$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
 $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Resource Allocation Graph with a Cycle but no Deadlock



$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

- Given the definition of a resource-allocation graph, it can be shown that if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.
 - 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 both a necessary and a sufficient condition for the existence of deadlock.
- 

- If each resource type has several instances, then a cycle does not necessarily 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.

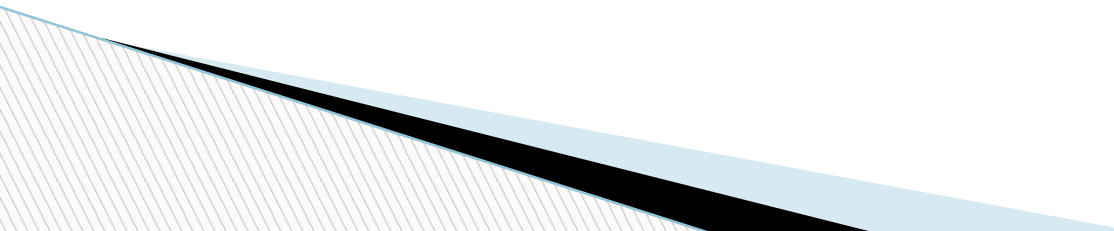
Methods for Handling Deadlocks

1. Ensure that the system will *never* enter a deadlock state.
 - **Deadlock prevention** - prevent one of necessary conditions occurring.
 - **Deadlock avoidance** - monitor resource use and deny requests that would lead to deadlock.
2. Allow the system to enter a deadlock state and then recover.
 - **Deadlock detection and recovery**
3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

Deadlock Prevention

Remove the possibility of deadlock occurring by denying one of the four necessary conditions:

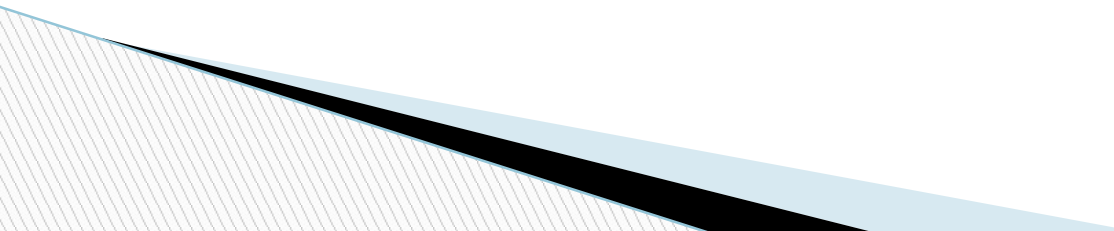
- *Mutual Exclusion*
 - *Hold & Wait*
 - *No pre-emption*
 - *Circular Wait*
- 

Deadlock Prevention

Disallow at least one condition that is necessary for deadlock to occur.

- **Disallow Mutual Exclusion:** not required for sharable resources; The mutual exclusion condition **must hold for non-sharable resources** - For example, a printer cannot be simultaneously shared by several processes.**so this would not work.**
- Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a shareable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- In general, however, we **cannot prevent deadlocks by denying the mutual-exclusion** condition because some resources are intrinsically non-sharable.

- **Disallow Hold and Wait:** must guarantee that whenever a process requests a resource, it does not hold any other resources.
 - A process is given its resources on an "ALL or NONE" basis:
 - 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.
 - Either a process gets ALL its required resources and proceeds or it gets NONE of them and waits until it gets ALL
 - Low resource utilization; starvation possible.

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
 - **One protocol** that can be used requires each process to request and be allocated all its resources before it begins execution.
 - **An alternative protocol** allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
- 

- To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.
- If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer.
- It will hold the printer for its entire execution, even though it needs the printer only at the end.

- The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

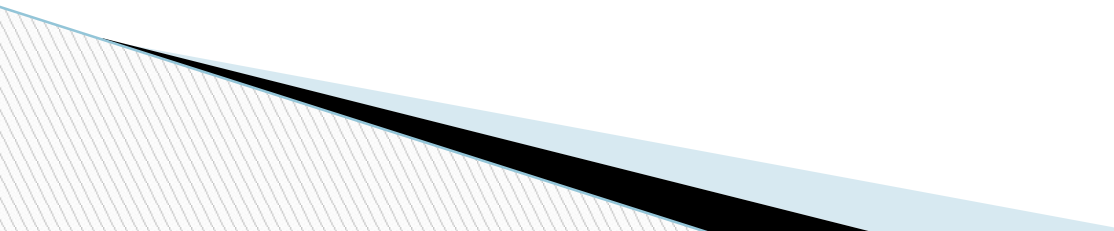
Disadvantages

- First, **resource utilization** may be low, since resources may be allocated but unused for a long period. In the example given, for instance, we can release the DVD drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.
- Second, **starvation** is possible. A process that needs several popular resources may have to wait indefinitely because at least one of the resources that it needs is always allocated to some other process.

Denying “No preemption”

- Implementation

- Resources can be removed from a process before it is finished with them.

- To ensure that this condition does not hold, we can use the following protocol.
 - 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.
- 

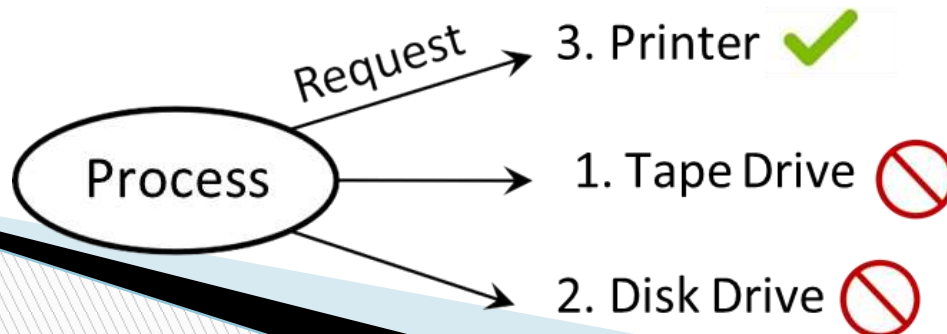
Denying “Circular Wait”

- Implementation

- Resources are uniquely numbered: impose a total ordering of all resource types and require that each process request resources in an increasing order of enumeration.

1. Tape Drive
2. Disk Drive
3. Printer
4. CPU

- Thus preventing the circular wait from occurring





Example

Consider three resources:

- **R1 (Printer)** → Assigned number 1
- **R2 (Scanner)** → Assigned number 2
- **R3 (Hard Drive)** → Assigned number 3

Now, assume two processes:

- **P1 requests R1 (Printer) first**, then requests R2 (Scanner). 
Allowed
- **P2 requests R2 (Scanner) first**, then requests R3 (Hard Drive). 
Allowed

However, **P1 cannot request R1 after holding R2**, because that would mean requesting resources in a decreasing order (which is not allowed). This prevents a circular chain of dependencies, thus **breaking the circular wait condition** and preventing deadlock.

This method is called the **Resource Ordering Strategy**

P1 → R1 → and then R2 But

R2 → P2 → R1 (P2 can't request for R1 so denying CW)

Deadlock Avoidance

Deadlock Avoidance

- Possible side effects of preventing deadlock are low device utilization and reduced system throughput.
- A method for avoiding deadlocks is to require additional information about how resources are to be requested.

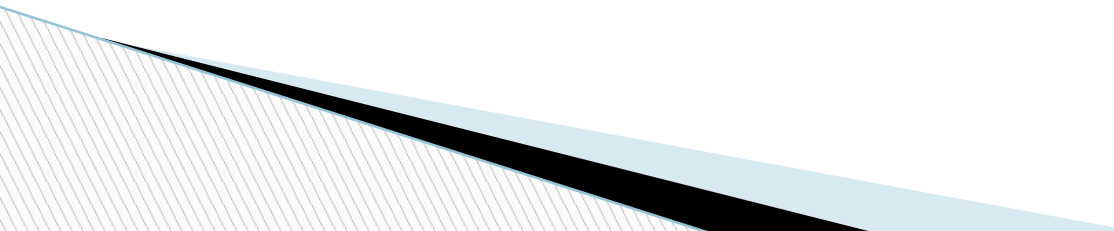
Deadlock Avoidance

- With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid possible future deadlock.
- A deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that a **circular wait** condition can never happen.

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



Safe State

- A state is **safe** if the system can allocate resources to each processes in some order (**safe sequence**) and still avoid a deadlock.
 - If no such sequence exists, then the system state is said to be **unsafe**.
- 

Safe State

- To illustrate, we consider a system has 12 magnetic tape drives and 3 processes. P0 needs 10 tapes, P1 needs 4 and P2 needs 9.
- Currently, P0 has 5, P1 has 2 and P2 has 2

	max need	allocated	<u>Need</u> (max need- allocated)
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7

- 3 drives remaining
- Safe sequence = $\langle P_1, P_0, P_2 \rangle$

What is the minimum number of resources required to ensure that deadlock will never occur, if there are currently three processes P_1 , P_2 , and P_3 running in a system whose maximum demand for the resources of the same type are 3, 4, and 5 respectively?

Data:

Maximum resource requirement of Process $P_1 = 3$

Maximum resource requirement of Process $P_2 = 4$

Maximum resource requirement of Process $P_3 = 5$

Concept:

Deadlock can occur If any process gets available resource $<$ demanded resource

Max resource for process P_1 to be in deadlock = needed - 1 = $3 - 1 = 2$

Max resource for process P_2 to be in deadlock = needed - 1 = $4 - 1 = 3$

Max resource for process P_3 to be in deadlock = needed - 1 = $5 - 1 = 4$

If one resource is added, any of the n processes can take it and finish its execution leaving behind the allocated resource, and hence the system will be deadlock-free.

Calculation:

The minimum value of m that ensures that deadlock will never occur = $(2 + 3 + 4) + 1 = 10$

What is the minimum number of resources required to ensure that deadlock will never occur, if there are currently four processes P1, P2, P3, P4 running in a system whose maximum demand for the resources of same type are 3, 4, 2 and 4 resp?

- a. 9
- b. 8
- c. 10
- d. 15

a. 10



What is the maximum number of resources required to ensure that deadlock will never occur, if there are currently five processes P1, P2, P3, P4, P5 running in a system whose maximum demand for the resources of same type are 2, 3, 4, 4 and 5 resp?

- a. 19
- b. 8
- c. 10
- d. 14

a. 19

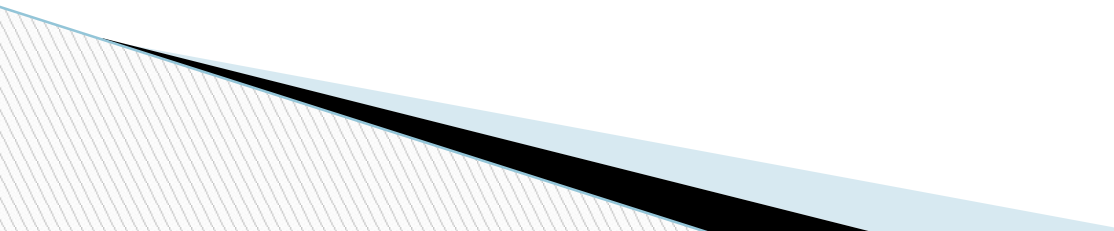


There are 4 process and each require 2 resources to complete the task. So, what would be the Minimum number of resource which guarantee the dead lock free ?

The formula often used in this context is: $n * k - (n - 1)$ where 'n' is the number of processes and 'k' is the maximum resources a process needs. In this case, it's $4 * 2 - (4 - 1) = 8 - 3 = 5$

A system is having 4 processes each requiring 3 units of resource R, the minimum number of units of R such that no deadlock will occur?

A system is having 4 processes each requiring 4 units of resource R, the minimum number of units of R such that no deadlock will occur?



A system is having 7 processes each requiring 2 units of resource R, the minimum number of units of R such that no deadlock will occur?

- a. 7
- b. 8
- c. 5
- d. 10

8



A system is having 5 processes each requiring 3 units of resource R, the maximum number of units of R such that no deadlock will occur?

- a. 11
- b. 10
- c. 14
- d. 9

14



A system is having 4 processes each requiring 3 units of resource R, the maximum number of units of R such that no deadlock will occur?

- a. 11
- b. 10
- c. 14
- d. 9

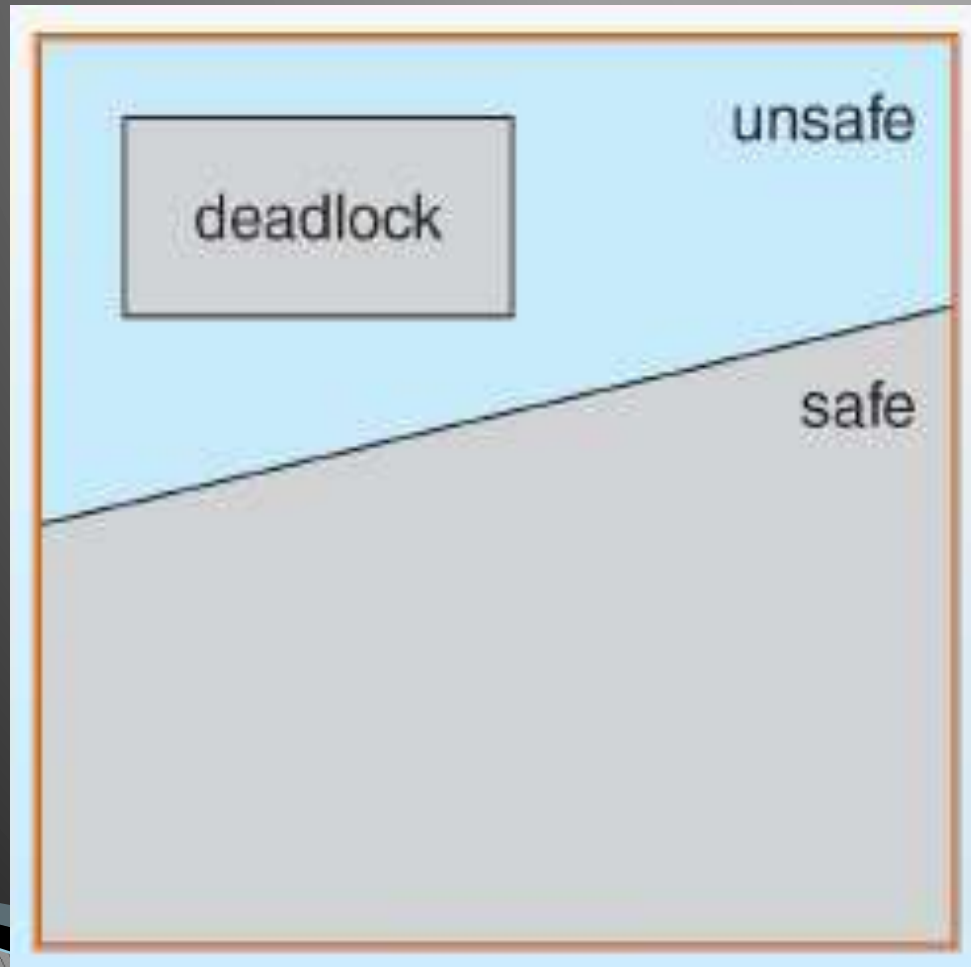
14



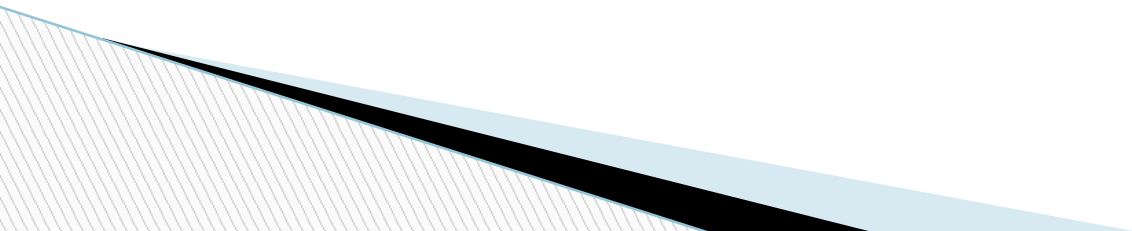
Basic Facts

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

Safe, Unsafe, Deadlock




Deadlock Avoidance

- The algorithm is simply to ensure that the system will always remain in a safe state.
 - Therefore, if a process requests a resource that is currently available, it may still have to wait.
 - Thus, resource utilization may be lower.
- 

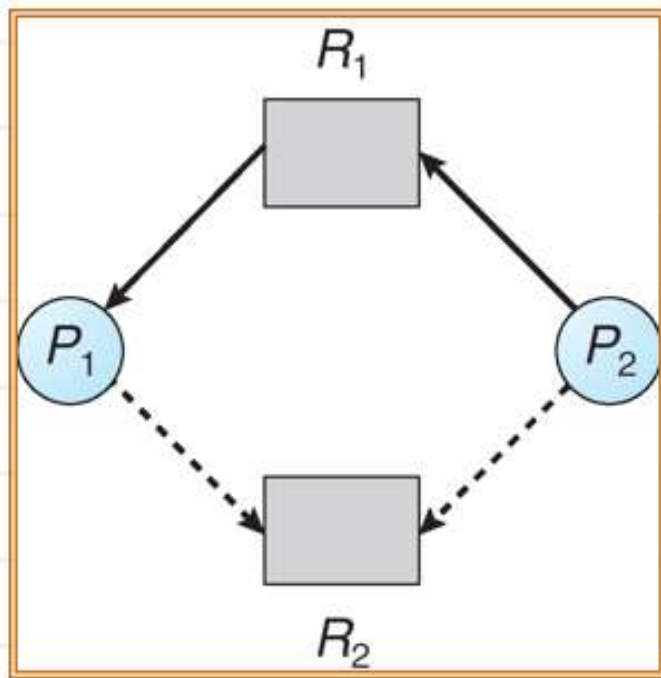
Avoidance algorithms

- ? **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

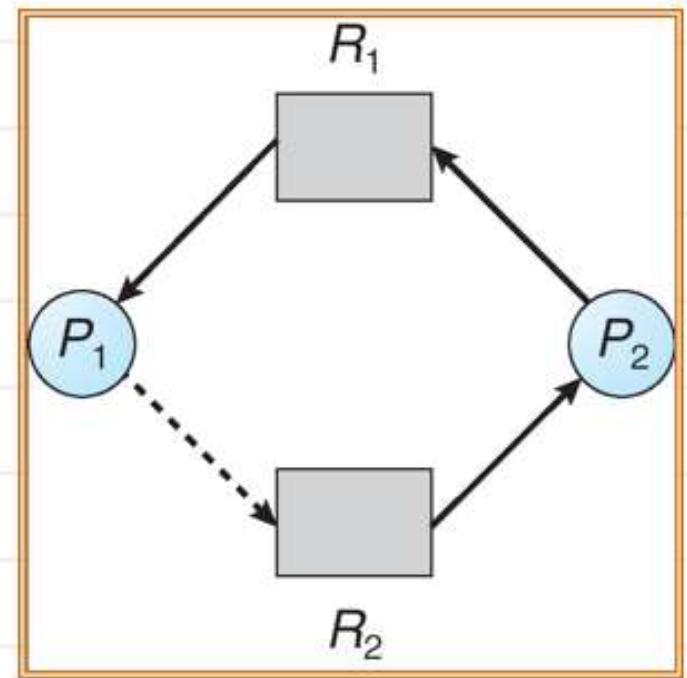

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_i **may request** resource R_j *in the future*; represented by a dashed line. 
- **Claim edge converts to request edge** when a process **requests** a resource.
- **Request edge converted to an assignment edge** when the resource is **allocated** to the process.
- When a resource is released by a process, assignment edge reconverts to a claim edge.

- 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.



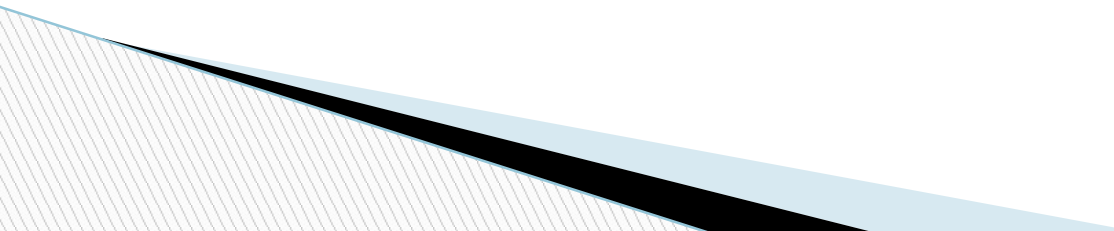
Safe State

Allocate one
instance of
 R_2 to P_2



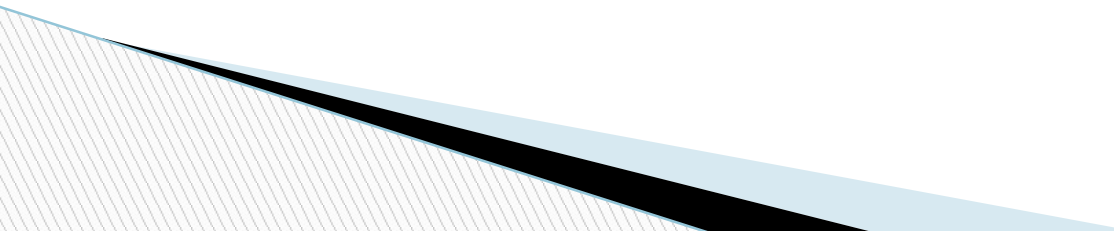
Unsafe State

Banker's Algorithm

- Multiple instances.
 - Each process must claim maximum use in advance.
 - 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.
- 

Banker's Algorithm

Two algorithms need to be discussed:

1. Safety state check algorithm
 2. Resource request algorithm
- 

Banker's Algorithm

- The banker's algorithm is a resource allocation and deadlock avoidance algorithm.
- It tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources.
- Makes an “safe-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

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**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j .
- **Allocation**: $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j .
- **Need**: $n \times m$ 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 $i = 0, 1, \dots, n-1$.

2. Find and 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.

Resource-Request Algorithm for Process P_i

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:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- 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

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 :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	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 of Banker's Algorithm

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

	<u>Need</u>
	<i>A B C</i>
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 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

Example of Banker's Algorithm

- Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true.

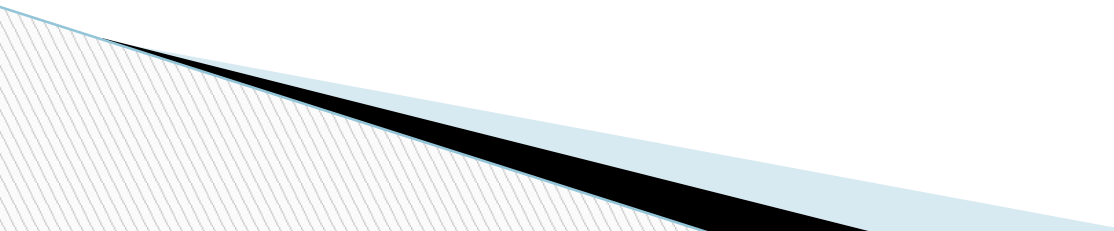
	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ 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?

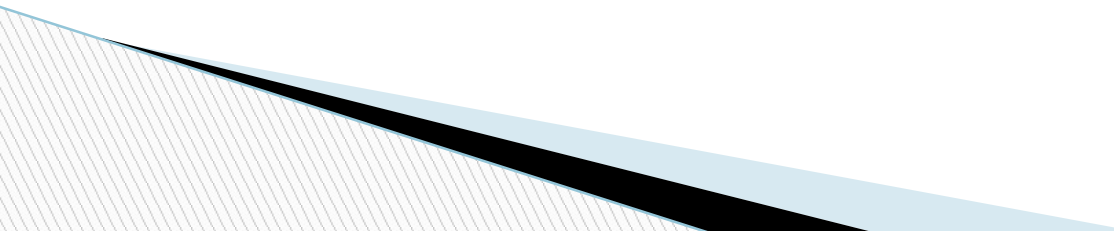
Deadlock Avoidance

- Allow the chance of deadlock occur
- But avoid it happening.
- Check whether the next state (change in system) may end up in a deadlock situation

Deadlock Detection

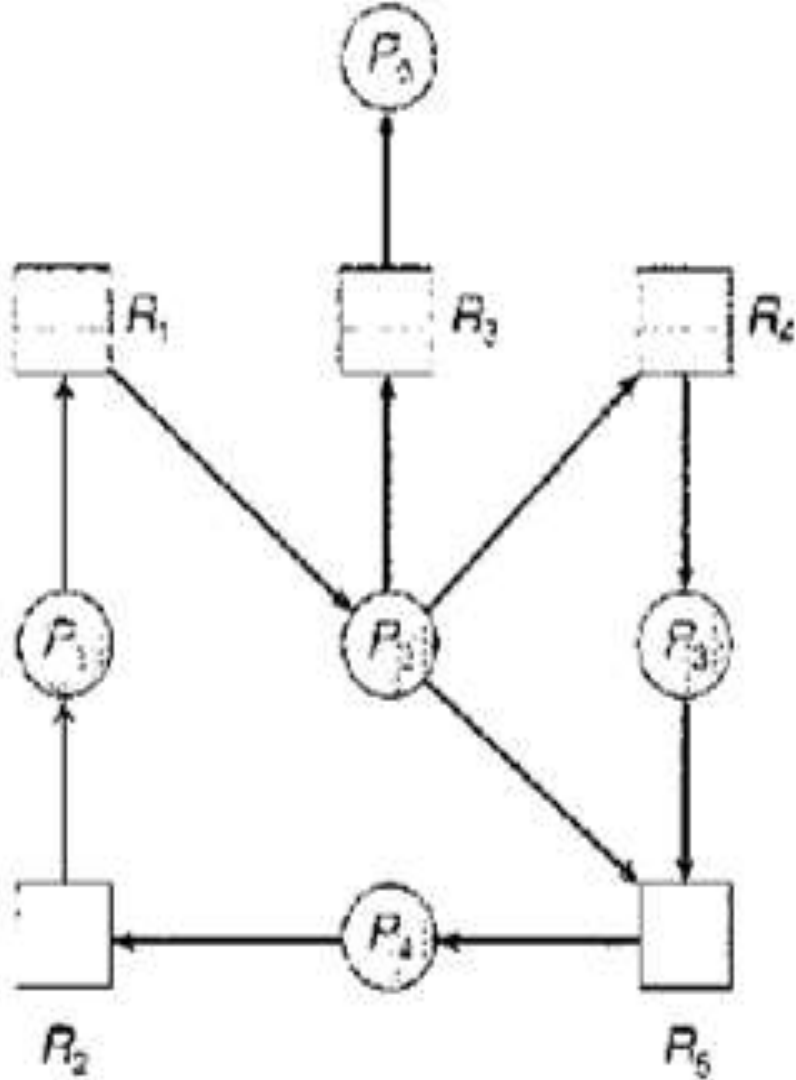
- Methods by which the occurrence of deadlock, the processes and resources involved are detected.
 - Generally work by detecting a circular wait
 - The cost of detection must be considered
 - One method is resource allocation graphs
- 

Single Instance of Each Resource Type

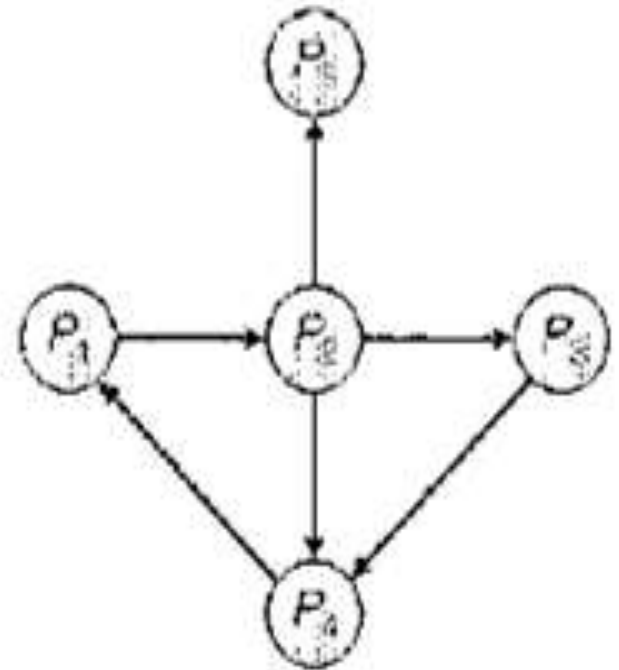
- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.
 - We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- 

Transformation from Resource Graph to wait-for graph

- ? An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- ? An edge $P_i \text{ ----} > P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i, \text{ ---} > R_q$ and $R_q \text{ ---} > P_j$ for some resource R_q



(a)



(b)

Resource Allocation Graph and wait Graph

Several Instances of a Resource Type

- The algorithm employs several time-varying data structures
- **Available:** A vector of length m indicates the number of available resources of each type.
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An $n \times m$ matrix indicates the current request of each process. If $\text{Request}[i][j]$ equals 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 $Work = Available$. For $i = 0, 1, \dots, n-1$, 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.
4. If $Finish[i] == false$, for some i , $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.

- ? To illustrate this algorithm, we consider a system with five processes P1 through P4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time T_0 , we have the following resource-allocation state:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	ABC	ABC
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	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ results in $Finish[i] = \text{true}$ for all i .

Suppose now that process P_2 makes one additional request for an instance of type C. The *Request* matrix is modified as follows:

	<i>Request</i>		
	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

We claim that the system is now deadlocked. Although we can reclaim the resources held by process P_0 , the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P_1 , P_2 , P_3 , and P_4 .

Recovery From Deadlock

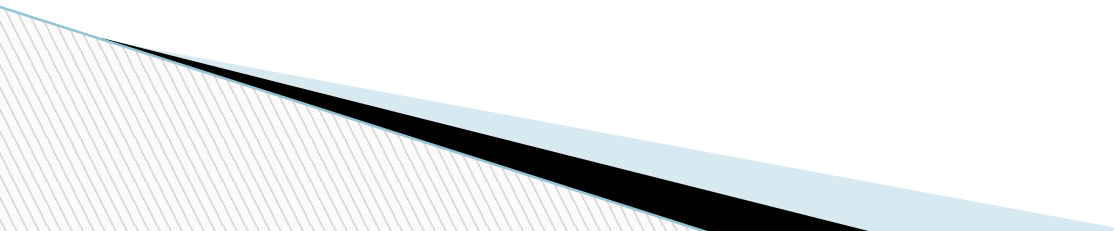
When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.

- Another possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock.
- One is simply to **abort one or more processes** to break the circular wait. The other is to **preempt some resources** from one or more of the deadlocked processes.

Process Termination

- ? To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.


Abort all deadlocked processes

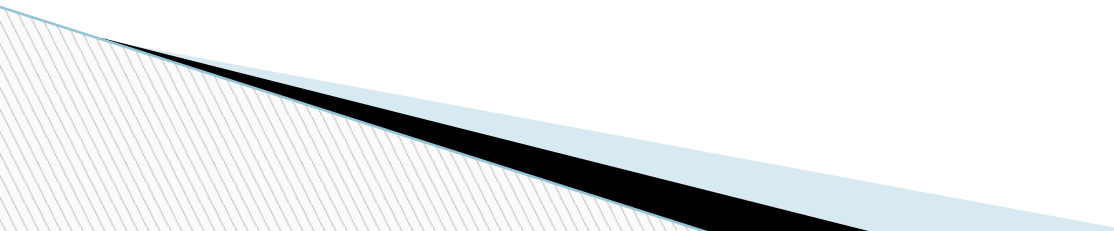
- ? This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- 

Abort one process at a time until the deadlock cycle is eliminated

- ? This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

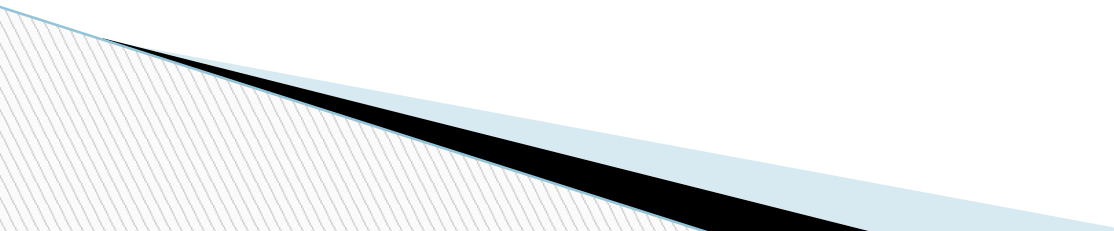
- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job

- if the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions.
 - The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. Many factors may affect which process is chosen, including:
- 

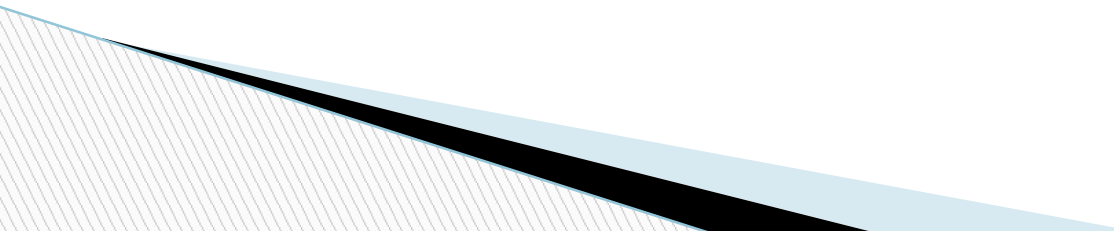
1. What the priority of the process is
 2. How long the process has computed and how much longer the process will compute before completing its designated task
 3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)
- 

4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated

Resource Preemption


- To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
 - If 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? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution
- 

Rollback

- If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

- Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it.
 - Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
- 

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?
- In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that must be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Question : What is the simplest and most used method to recover from a deadlock?

RE-BOOT

