UNIT-III

DEADLOCKS

DEADLOCK IN OPERATING SYSTEM

Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

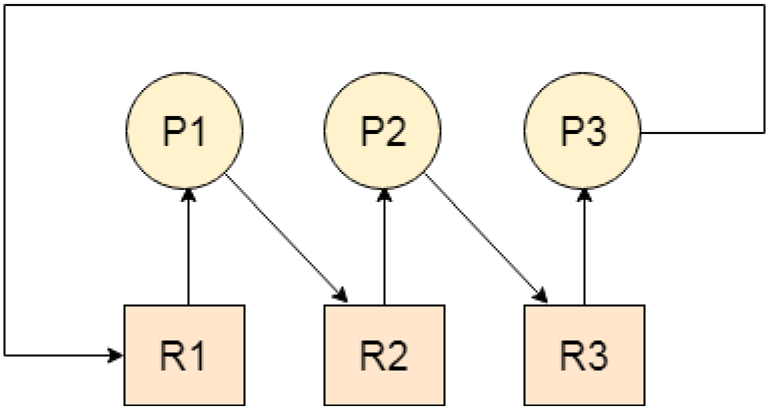
1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

Let us assume that there are three processes P1, P2 and P3. There are three different resources R1, R2 and R3. R1 is assigned to P1, R2 is assigned to P2 and R3 is assigned to P3.

After some time, P1 demands for R1 which is being used by P2. P1 halts its execution since it can't complete without R2. P2 also demands for R3 which is being used by P3. P2 also stops its execution because it can't continue without R3. P3 also demands for R1 which is being used by P1 therefore P3 also stops its execution.

In this scenario, a cycle is being formed among the three processes. None of the process is progressing and they are all waiting. The computer becomes unresponsive since all the processes got blocked.



Difference between Starvation and Deadlock

Sr.	Deadlock	Starvation
1	Deadlock is a situation where no process got blocked and no process proceeds	Starvation is a situation where the low priority process got blocked and the high priority processes proceed.
2	Deadlock is an infinite waiting.	Starvation is a long waiting but not infinite.
3	Every Deadlock is always a starvation.	Every starvation need not be deadlock.
4	The requested resource is blocked by the other process.	The requested resource is continuously be used by the higher priority processes.
5	Deadlock happens when Mutual exclusion, hold and wait, No preemption and circular wait occurs simultaneously.	It occurs due to the uncontrolled priority and resource management.

Necessary conditions for Deadlocks

1. **Mutual Exclusion**

   A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.

2. **Hold and Wait**

   A process waits for some resources while holding another resource at the same time.

3. **No preemption**

   The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.

4. **Circular Wait**

   All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

SYSTEM MODEL

A deadlock occurs when a set of processes is stalled because each process is holding a resource and waiting for another process to acquire another resource. In the diagram below, for example, Process 1 is holding Resource 1 while Process 2 acquires Resource 2, and Process 2 is waiting for Resource 1.

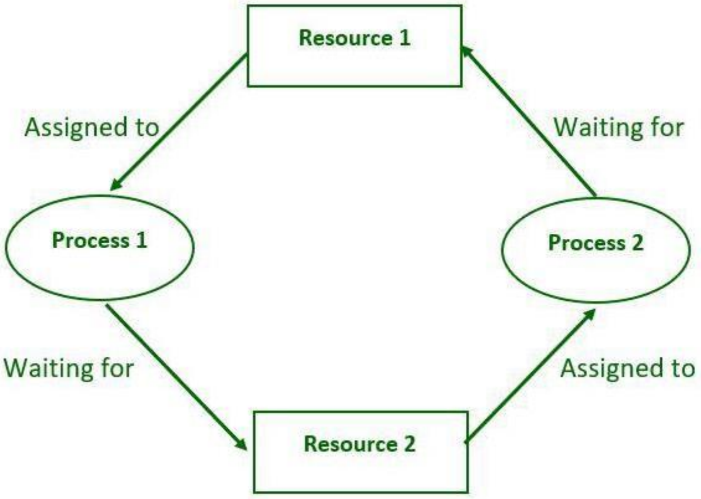


Figure: Deadlock in Operating system

System Model :

- For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources that can be divided into different categories and allocated to a variety of processes, each with different requirements.
- Memory, printers, CPUs, open files, tape drives, CD-ROMs, and other resources are examples of resource categories.
- By definition, all resources within a category are equivalent, and any of the resources within that category can equally satisfy a request from that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category must be subdivided further. For example, the term “printers” may need to be subdivided into “laser printers” and “color inkjet printers.”
- Some categories may only have one resource.
- The kernel keeps track of which resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available for all kernel-managed resources. Mutexes or wait() and signal() calls can be used to control application-managed resources (i.e. binary or counting semaphores.)

- When every process in a set is waiting for a resource that is currently assigned to another process in the set, the set is said to be deadlocked.

Operations:

In normal operation, a process must request a resource before using it and release it when finished, as shown below.

1. **Request–**
   If the request cannot be granted immediately, the process must wait until the resource(s) required to become available. The system, for example, uses the functions open(), malloc(), new(), and request ().
2. **Use–**
   The process makes use of the resource, such as printing to a printer or reading from a file.
3. **Release–**
   The process relinquishes the resource, allowing it to be used by other processes.

Necessary Conditions

There are four conditions that must be met in order to achieve deadlock as follows.

1. **Mutual Exclusion** –
   At least one resource must be kept in a non-shareable state; if another process requests it, it must wait for it to be released.

2. **Hold and Wait** –
   A process must hold at least one resource while also waiting for at least one resource that another process is currently holding.

3. **No preemption** –
   Once a process holds a resource (i.e. after its request is granted), that resource cannot be taken away from that process until the process voluntarily releases it.

4. **Circular Wait** –
   There must be a set of processes P0, P1, P2,…, PN such that every P[I] is waiting for P[(I + 1) percent (N + 1)]. (It is important to note that this condition implies the hold-and-wait condition, but dealing with the four conditions is easier if they are considered separately).

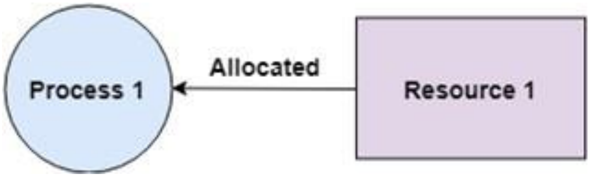
DEADLOCK CHARACTERIZATION

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.

A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive. They are given as follows −

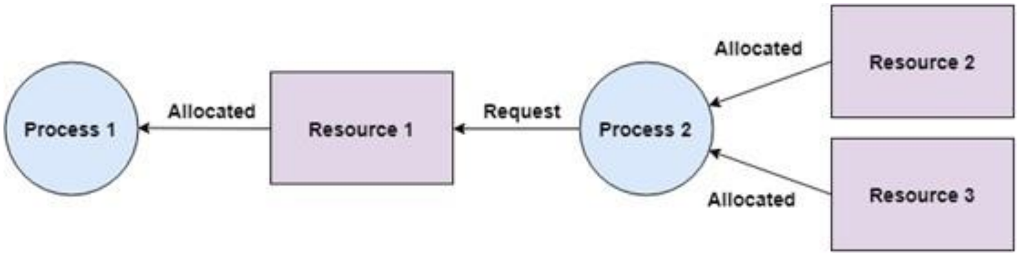
Mutual Exclusion

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



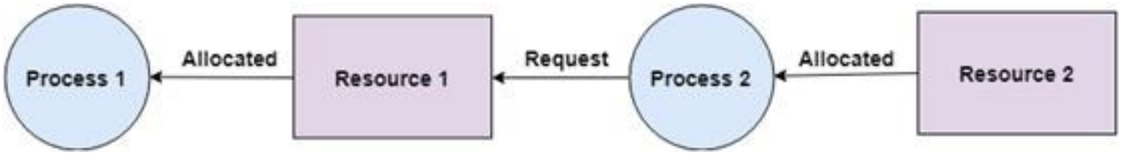
Hold and Wait

A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



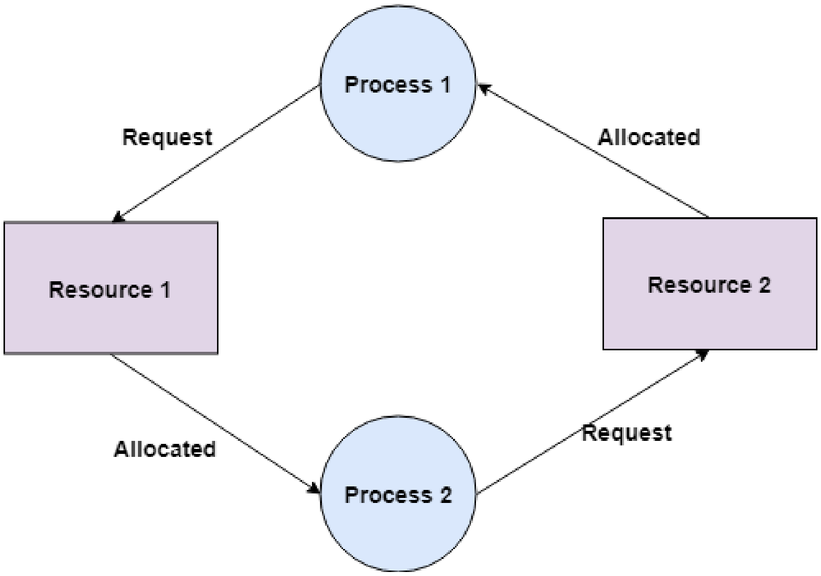
No Preemption

A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



Circular Wait

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



METHODS FOR HANDLING DEADLOCKS

Deadlock is a situation where a process or a set of processes is blocked, waiting for some other resource that is held by some other waiting process. It is an undesirable state of the system. The following are the four conditions that must hold simultaneously for a deadlock to occur.

1. **Mutual Exclusion** – A resource can be used by only one process at a time. If another process requests for that resource then the requesting process must be delayed until the resource has been released.

2. **Hold and wait** – Some processes must be holding some resources in the non-shareable mode and at the same time must be waiting to acquire some more resources, which are currently held by other processes in the non-shareable mode.
3. **No pre-emption** – Resources granted to a process can be released back to the system only as a result of voluntary action of that process after the process has completed its task.
4. **Circular wait** – Deadlocked processes are involved in a circular chain such that each process holds one or more resources being requested by the next process in the chain.

Methods of handling deadlocks: There are four approaches to dealing with deadlocks.

1. Deadlock Prevention

2. Deadlock avoidance (Banker's Algorithm)

3. Deadlock detection & recovery

4. Deadlock Ignorance (Ostrich Method)

1. Deadlock Prevention: The strategy of deadlock prevention is to design the system in such a way that the possibility of deadlock is excluded. The indirect methods prevent the occurrence of one of three necessary conditions of deadlock i.e., mutual exclusion, no pre-emption, and hold and wait. The direct method prevents the occurrence of circular wait. **Prevention techniques – Mutual exclusion** – are supported by the OS. **Hold and Wait** – the condition can be prevented by requiring that a process requests all its required resources at one time and blocking the process until all of its requests can be granted at the same time simultaneously. But this prevention does not yield good results because:

- long waiting time required
- inefficient use of allocated resource
- A process may not know all the required resources in advance

No pre-emption – techniques for ‘no pre-emption are’

- If a process that is holding some resource, requests another resource that can not be immediately allocated to it, all resources currently being held are released and if necessary, request again together with the additional resource.
- If a process requests a resource that is currently held by another process, the OS may pre-empt the second process and require it to release its resources. This works only if both processes do not have the same priority.

Circular wait One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in increasing order of enumeration, i.e., if a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in ordering.

2. Deadlock Avoidance: The deadlock avoidance Algorithm works by proactively looking for potential deadlock situations before they occur. It does this by tracking the resource usage of each process and identifying conflicts that could potentially lead to a deadlock. If a potential deadlock is identified, the algorithm will take steps to resolve the conflict, such as rolling back one of the processes or pre-emptively allocating resources to other processes. The Deadlock Avoidance Algorithm is designed to minimize the chances of a deadlock occurring, although it cannot guarantee that a deadlock will never occur. This approach allows the three necessary conditions of deadlock but makes judicious choices to assure that the deadlock point is never reached. It allows more concurrency than avoidance detection A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to deadlock. It requires knowledge of future process requests. Two techniques to avoid deadlock :

1. Process initiation denial
2. Resource allocation denial

Advantages of deadlock avoidance techniques:

- Not necessary to pre-empt and rollback processes
- Less restrictive than deadlock prevention

Disadvantages :

- Future resource requirements must be known in advance
- Processes can be blocked for long periods
- Exists a fixed number of resources for allocation

Banker’s Algorithm:

The Banker’s Algorithm is based on the concept of resource allocation graphs. A resource allocation graph is a directed graph where each node represents a process, and each edge represents a resource. The state of the system is represented by the current allocation of resources between processes. For example, if the system has three processes, each of which is using two resources, the resource allocation graph would look like this:

Processes A, B, and C would be the nodes, and the resources they are using would be the edges connecting them. The Banker’s Algorithm works by analyzing the state of the system and determining if it is in a safe state or at risk of entering a deadlock.

To determine if a system is in a safe state, the Banker’s Algorithm uses two matrices: the available matrix and the need matrix. The available matrix contains the amount of each resource currently available. The need matrix contains the amount of each resource required by each process.

The Banker’s Algorithm then checks to see if a process can be completed without overloading the system. It does this by subtracting the amount of each resource used by the process from the available matrix and adding it to the need matrix. If the result is in a safe state, the process is allowed to proceed, otherwise, it is blocked until more resources become available.

The Banker’s Algorithm is an effective way to prevent deadlocks in multiprogramming systems. It is used in many operating systems, including Windows and Linux. In addition, it is used in many other types of systems, such as manufacturing systems and banking systems.

The Banker’s Algorithm is a powerful tool for resource allocation problems, but it is not foolproof. It can be fooled by processes that consume more resources than they need, or by processes that produce more resources than they need. Also, it can be fooled by processes that consume resources in an unpredictable manner. To prevent these types of problems, it is important to carefully monitor the system to ensure that it is in a safe state.

3. Deadlock Detection: Deadlock detection is used by employing an algorithm that tracks the circular waiting and kills one or more processes so that the deadlock is removed. The system state is examined periodically to determine if a set of processes is deadlocked. A deadlock is resolved by aborting and restarting a process, relinquishing all the resources that the process held.

- This technique does not limit resource access or restrict process action.
- Requested resources are granted to processes whenever possible.
- It never delays the process initiation and facilitates online handling.
- The disadvantage is the inherent pre-emption losses.

4. Deadlock Ignorance: In the Deadlock ignorance method the OS acts like the deadlock never occurs and completely ignores it even if the deadlock occurs. This method only applies if the deadlock occurs very rarely. The algorithm is very simple. It says ” if the deadlock occurs, simply reboot the system and act like the deadlock never occurred." That's why the algorithm is called the **Ostrich Algorithm**.

Advantages:

- Ostrich Algorithm is relatively easy to implement and is effective in most cases.
- It helps in avoiding the deadlock situation by ignoring the presence of deadlocks.

Disadvantages:

- Ostrich Algorithm does not provide any information about the deadlock situation.
- It can lead to reduced performance of the system as the system may be blocked for a long time.
- It can lead to a resource leak, as resources are not released when the system is blocked due to deadlock.

DEADLOCK PREVENTION

If we simulate deadlock with a table which is standing on its four legs then we can also simulate four legs with the four conditions which when occurs simultaneously, cause the deadlock.

However, if we break one of the legs of the table then the table will fall definitely. The same happens with deadlock, if we can be able to violate one of the four necessary conditions and don't let them occur together then we can prevent the deadlock.

Let's see how we can prevent each of the conditions.

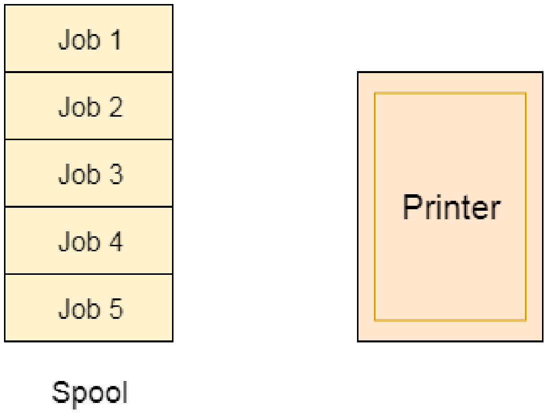
1. Mutual Exclusion

Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.

However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.

Spooling

For a device like printer, spooling can work. There is a memory associated with the printer which stores jobs from each of the process into it. Later, Printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing. Later, it collects the output when it is produced.



Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.

1. This cannot be applied to every resource.
2. After some point of time, there may arise a race condition between the processes to get space in that spool.

We cannot force a resource to be used by more than one process at the same time since it will not be fair enough and some serious problems may arise in the performance. Therefore, we cannot violate mutual exclusion for a process practically.

2. Hold and Wait

Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.

However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.

!(Hold and wait) = !hold or !wait (negation of hold and wait is, either you don't hold or you don't wait)

This can be implemented practically if a process declares all the resources initially. However, this sounds very practical but can't be done in the computer system because a process can't determine necessary resources initially.

Process is the set of instructions which are executed by the CPU. Each of the instruction may demand multiple resources at the multiple times. The need cannot be fixed by the OS.

The problem with the approach is:

1. Practically not possible.
2. Possibility of getting starved will be increases due to the fact that some process may hold a resource for a very long time.

3. No Preemption

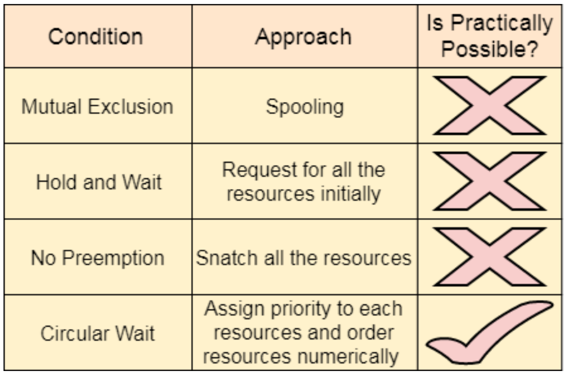
Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.

This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now can become inconsistent.

Consider a printer is being used by any process. If we take the printer away from that process and assign it to some other process then all the data which has been printed can become inconsistent and ineffective and also the fact that the process can't start printing again from where it has left which causes performance inefficiency.

4. Circular Wait

To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.



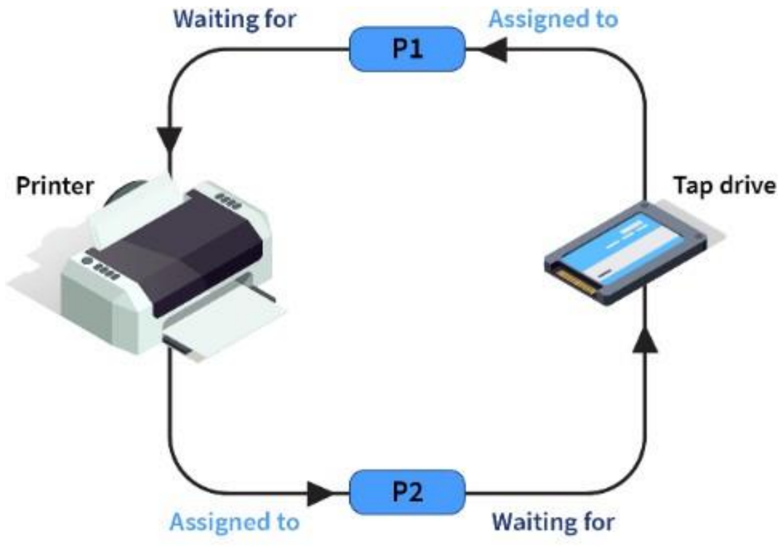
Condition	Approach	Is Practically Possible?
Mutual Exclusion	Spooling	✗
Hold and Wait	Request for all the resources initially	✗
No Preemption	Snatch all the resources	✗
Circular Wait	Assign priority to each resources and order resources numerically	✓

DEADLOCK AVOIDANCE

Deadlock Avoidance is a process used by the Operating System to avoid Deadlock. Let's first understand what is Deadlock in Operating System. Deadlock is a situation that occurs in Operating System when any Process enters a waiting state because another waiting process is holding the demanded resource. Deadlock is a common problem in multi-processing where several processes share a specific type of mutually exclusive resource known as a soft lock or software.

But how can Operating System avoid Deadlock?

Operating System avoids Deadlock by knowing the maximum resources requirements of the processes initially, and also Operating System knows the free resources available at that time. Operating System tries to allocate the resources according to the process requirements and checks if the allocation can lead to a safe state or an unsafe state. If the resource allocation leads to an unsafe state, then Operating System does not proceed further with the allocation sequence.



Deadlock

How does Deadlock avoidance work?

Let's understand the working of Deadlock Avoidance with the help of an intuitive example.

Process	Maximum Required	current Available	Need
P1	9	5	4
P2	5	2	3

Process	Maximum Required	current Available	Need
P3	3	1	2

Let's consider three processes P1, P2, P3. Some more information on which the processes tells the Operating System are :

- P1 process needs a maximum of 9 resources (Resources can be any software or hardware Resource like tape drive or printer etc..) to complete its execution. P1 is currently allocated with 5 Resources and needs 4 more to complete its execution.
- P2 process needs a maximum of 5 resources and is currently allocated with 2 resources. So it needs 3 more resources to complete its execution.
- P3 process needs a maximum of 3 resources and is currently allocated with 1 resource. So it needs 2 more resources to complete its execution.
- Operating System knows that only 2 resources out of the total available resources are currently free.

But only 2 resources are free now. Can P1, P2, and P3 satisfy their requirements? Let's try to find out.

As only 2 resources are free for now, then only P3 can satisfy its need for 2 resources. If P3 takes 2 resources and completes its execution, then P3 can release its 3 (1+2) resources. Now the three free resources which P3 released can satisfy the need of P2. Now, P2 after taking the three free resources, can complete its execution and then release 5 (2+3) resources. Now five resources are free. P1 can now take 4 out of the 5 free resources and complete its execution. So, with 2 free resources available initially, all the processes were able to complete their execution leading to Safe State. The order of execution of the processes was <P3, P2, P1>.

What if initially there was only 1 free resource available? None of the processes would be able to complete its execution. Thus leading to an unsafe state.

We use two words, safe and unsafe states. What are those states? Let's understand these concepts.

Safe State and Unsafe State

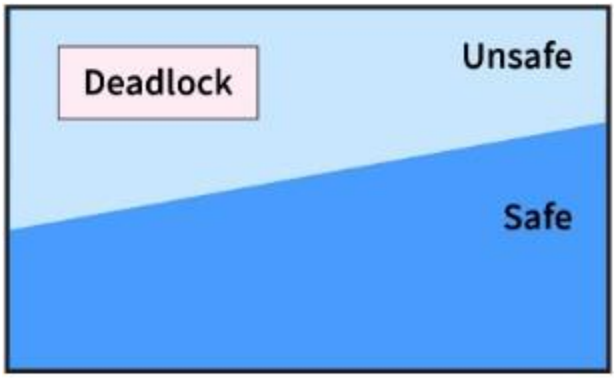
Safe State - In the above example, we saw that Operating System was able to satisfy the need of all three processes, P1, P2, and P3, with their resource requirements. So all the processes were able to complete their execution in a certain order like P3->P2->P1.

So, **If Operating System is able to allocate or satisfy the maximum resource requirements of all the processes in any order then the system is said to be in Safe State.**

So safe state does not lead to Deadlock.

Unsafe State - If Operating System is not able to prevent Processes from requesting resources which can also lead to Deadlock, then the System is said to be in an Unsafe State.

Unsafe State does not necessarialy cause deadlock it may or maynot causes deadlock.



So, in the above diagram shows the three states of the System. An unsafe state does not always cause Deadlock. Some unsafe states can lead to Deadlock, as shown in the diagram.

Deadlock Avoidance Example

Let's take an example that has multiple resources requirement for every Process. Let there be three Processes P1, P2, P3, and 4 resources R1, R2, R3, R4. The maximum resources requirements of the Processes are shown in the below table.

Process	**R1**	**R2**	**R3**	**R4**
P1	3	2	3	2
P2	2	3	1	4
P3	3	1	5	0

A number of currently allocated resources to the processes are:

Process	**R1**	**R2**	**R3**	**R4**
P1	1	2	3	1
P2	2	1	0	2
P3	2	0	1	0

The total number of resources in the System are :

R1	**R2**	**R3**	**R4**
7	4	5	4

We can find out the no of available resources for each of P1, P2, P3, P4 by subtracting the currently allocated resources from total resources.

Available Resources are :

R1	**R2**	**R3**	**R4**
2	1	1	1

Now, The need for the resources for the processes can be calculated by :

Need = Maximum Resources Requirement - Currently Allocated Resources.

The need for the Resources is shown below:

Process	**R1**	**R2**	**R3**	**R4**
P1	2	1	0	1
P2	0	2	1	2
P3	1	1	4	0

The available free resources are <2,1,1,1> of resources of R1, R2, R3, and R4 respectively, which can be used to satisfy only the requirements of process P1 only initially as process P2 requires 2 R2 resources which are not available. The same is the case with Process P3, which requires 4 R3 resources which is not available initially.

The Steps for resources allotment is explained below:

1. Firstly, Process P1 will take the available resources and satisfy its resource need, complete its execution and then release all its allocated resources. Process P1 is initially allocated <1,2,3,1> resources of R1, R2, R3, and R4 respectively. Process P1 needs <2,1,0,1> resources of R1, R2, R3 and R4 respectively to complete its execution. So, process P1 takes the available free resources <2,1,1,1> resources of R1, R2, R3, R4 respectively and can complete its execution and then release its current allocated resources and also the free resources it used to complete its execution. Thus P1 releases <1+2,2+1,3+1,1+1> = <3,3,4,2> resources of R1, R2, R3, and R4 respectively.
2. After step 1 now, available resources are now <3,3,4,2>, which can satisfy the need of Process P2 as well as process P3. After process P2 uses the available Resources and completes its execution, the available resources are now <5,4,4,4>.

3. Now, the available resources are <5,4,4,4>, and the only Process left for execution is Process P3, which requires <1,1,4,0> resources each of R1, R2, R3, and R4. So it can easily use the available resources and complete its execution. After P3 is executed, the resources available are <7,4,5,4>, which is equal to the maximum resources or total resources available in the System.

So, **the process execution sequence in the above example was <P1, P2, P3>**. But it could also have been <P1, P3, P2> if process P3 would have been executed before process P2, which was possible as there were sufficient resources available to satisfy the need of both Process P2 and P3 after step 1 above.

Deadlock Avoidance

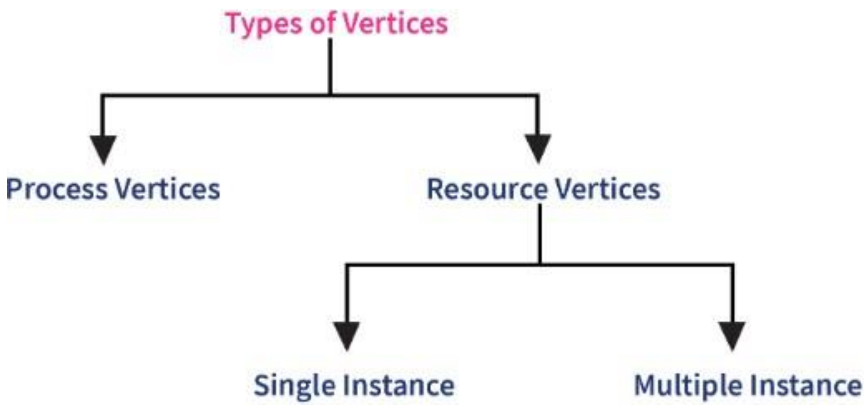
Deadlock Avoidance can be solved by two different algorithms:

- **Resource allocation Graph**
- **Banker's Algorithm**

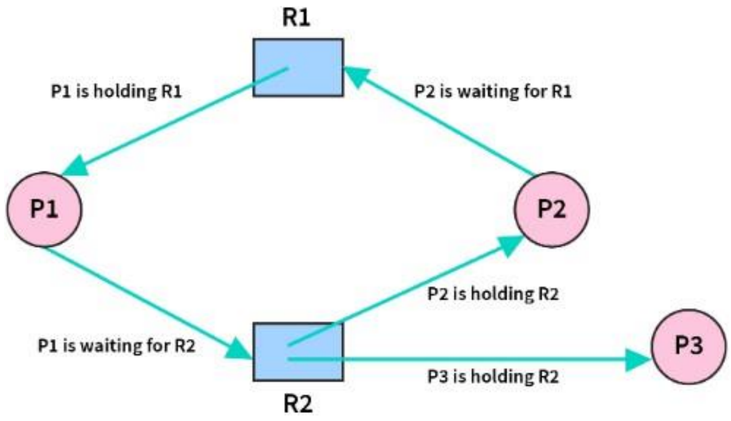
We will discuss both algorithms in detail in their separate article.

Resource Allocation Graph

Resource Allocation Graph (RAG) is used to represent the state of the System in the form of a Graph. The Graph contains all processes and resources which are allocated to them and also the requesting resources of every Process. Sometimes if the number of processes is less, We can easily identify a deadlock in the System just by observing the Graph, which can not be done easily by using tables that we use in Banker's algorithm.



Resource Allocation Graph has a process vertex represented by a circle and a resource vertex represented by a box. The instance of the resources is represented by a dot inside the box. The instance can be single or multiple instances of the resource. An example of RAG is shown below.



Multi Instances Without Deadlock

Banker's Algorithm

Banker's algorithm does the same as we explained the Deadlock avoidance with the help of an example. The algorithm predetermines whether the System will be in a safe state or not by simulating the allocation of the resources to the processes according to the maximum available resources. **It makes an "s-state" check before actually allocating the resources to the Processes**.

When there are more number of Processes and many Resources, then Banker's Algorithm is useful.

DEADLOCK DETECTION

The algorithm employs several times varying data structures:

- **Available –**

  A vector of length m indicates the number of available resources of each type.
- **Allocation –**

  An n*m matrix defines the number of resources of each type currently allocated to a process. The column represents resource and rows represent a process.
- **Request –**

  An n*m matrix indicates the current request of each process. If request[i][j] equals k then process P_i is requesting k more instances of resource type R_j.

This algorithm has already been discussed here

Now, the Bankers algorithm includes a **Safety Algorithm / Deadlock Detection Algorithm** The algorithm for finding out whether a system is in a safe state can be described as follows:

Steps of Algorithm:

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]* = true; otherwise, *Finish[i]*= false.
2. Find an index i such that both

   a) *Finish[i] == false*

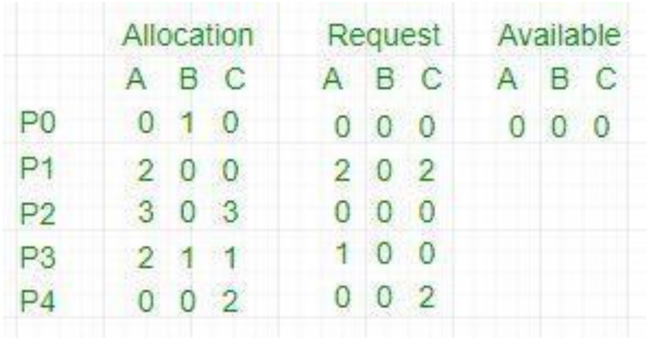
   b) *Request*$_i$ <= *Work*

   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* the process P_i is deadlocked.

For example,



	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

1. In this, Work = [0, 0, 0] &

   Finish = [false, false, false, false, false]

2. *i=0* is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].

3. Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] &
   Finish = [true, false, false, false, false].

4. *i=2* is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].
5. Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3] &
   Finish = [true, false, true, false, false].

6. *i=1* is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].
7. Work =[3, 1, 3]+[2, 0, 0] =>[5, 1, 3] &
   Finish = [true, true, true, false, false].

8. *i=3* is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 3].
9. Work =[5, 1, 3]+[2, 1, 1] =>[7, 2, 4] &
   Finish = [true, true, true, true, false].

10. *i=4* is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 4].

11. Work =[7, 2, 4]+[0, 0, 2] =>[7, 2, 6] &
    Finish = [true, true, true, true, true].

12. Since Finish is a vector of all true it means **there is no deadlock** in this example.

There are several algorithms for detecting deadlocks in an operating system, including:

1. **Wait-For Graph:** A graphical representation of the system's processes and resources. A directed edge is created from a process to a resource if the process is waiting for that resource. A cycle in the graph indicates a deadlock.
2. **Banker's Algorithm:** A resource allocation algorithm that ensures that the system is always in a safe state, where deadlocks cannot occur.
3. **Resource Allocation Graph**: A graphical representation of processes and resources, where a directed edge from a process to a resource means that the process is currently holding that resource. Deadlocks can be detected by looking for cycles in the graph.

4. **Detection by System Modeling:** A mathematical model of the system is created, and deadlocks can be detected by finding a state in the model where no process can continue to make progress.
5. **Timestamping**: Each process is assigned a timestamp, and the system checks to see if any process is waiting for a resource that is held by a process with a lower timestamp.

These algorithms are used in different operating systems and systems with different resource allocation and synchronization requirements. The choice of algorithm depends on the specific requirements of the system and the trade-offs between performance, complexity, and accuracy. If a system does not employ either a deadlock prevention or deadlock avoidance algorithm then a deadlock situation may occur. In this case-

- Apply an algorithm to examine the system’s state to determine whether deadlock has occurred.
- Apply an algorithm to recover from the deadlock. For more refer- Deadlock Recovery

Advantages of Deadlock Detection Algorithms in Operating Systems:

1. Improved System Stability: Deadlocks are a major concern in operating systems, and detecting and resolving deadlocks can help to improve the stability of the system.
2. Better Resource Utilization: By detecting deadlocks and freeing resources, the operating system can ensure that resources are efficiently utilized and that the system remains responsive to user requests.
3. Easy Implementation: Some deadlock detection algorithms, such as the Wait-For Graph, are relatively simple to implement and can be used in a wide range of operating systems and systems with different resource allocation and synchronization requirements.

Disadvantages of Deadlock Detection Algorithms in Operating Systems:

1. Performance Overhead: Deadlock detection algorithms can introduce a significant overhead in terms of performance, as the system must regularly check for deadlocks and take appropriate action.
2. Complexity: Some deadlock detection algorithms, such as the Resource Allocation Graph or Timestamping, are more complex to implement and require a deeper understanding of the system and its behavior.

3. False Positives and Negatives: Deadlock detection algorithms are not perfect and may produce false positives or negatives, indicating the presence of deadlocks when they do not exist or failing to detect deadlocks that do exist.
4. Overall, the choice of deadlock detection algorithm depends on the specific requirements of the system, the trade-offs between performance, complexity, and accuracy, and the risk tolerance of the system. The operating system must balance these factors to ensure that deadlocks are detected and resolved effectively and efficiently.

RECOVERY FROM DEADLOCK

When a Deadlock Detection Algorithm determines that a deadlock has occurred in the system, the system must recover from that deadlock. There are two approaches of breaking a Deadlock:

1. Process Termination:

To eliminate the deadlock, we can simply kill one or more processes. For this, we use two methods:

- **(a). Abort all the Deadlocked Processes:**
  Aborting all the processes will certainly break the deadlock, but with a great expense. The deadlocked processes may have computed for a long time and the result of those partial computations must be discarded and there is a probability to recalculate them later.
- **(b). Abort one process at a time until deadlock is eliminated:**
  Abort one deadlocked process at a time, until deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead, because after aborting each process, we have to run deadlock detection algorithm to check whether any processes are still deadlocked.

2. Resource Preemption:

To eliminate deadlocks using resource preemption, we preempt some resources from processes and give those resources to other processes. This method will raise three issues –

- **(a). Selecting a victim:**
  We must determine which resources and which processes are to be preempted and also the order to minimize the cost.

- **(b). Rollback:**

  We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means abort the process and restart it.

- **(c). Starvation:**

  In a system, it may happen that same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called **Starvation** and must be avoided. One solution is that a process must be picked as a victim only a finite number of times.