UNIT – 2

Process Synchronization:

In multi-processing systems, when multiple concurrent processes execute and update shared resources, the operating system needs to preserve the order of execution to achieve correct results. This demands that the interacting processes need to execute in a co-ordinated manner. Process synchronization is the procedure that achieves the desired coordination.

Understanding Process Synchronization

Definition and Purpose

Process synchronization involves the coordination and control of concurrent processes to ensure correct and predictable outcomes. Its primary purpose is to prevent race conditions, data inconsistencies, and resource conflicts that may arise when multiple processes access shared resources simultaneously.

Challenges in Concurrent Execution

Concurrent execution introduces several challenges, including −

- **Race Conditions** − Concurrent processes accessing shared resources may result in unexpected and erroneous outcomes. For example, if two processes simultaneously write to the same variable, the final value may be unpredictable or incorrect.
- **Deadlocks** − Processes may become stuck in a state of waiting indefinitely due to resource dependencies. Deadlocks occur when processes are unable to proceed because each process is waiting for a resource held by another process, creating a circular dependency.
- **Starvation** − A process may be denied access to a shared resource indefinitely, leading to its inability to make progress. This situation arises when certain processes consistently receive priority over others, causing some processes to wait indefinitely for resource access.
- **Data Inconsistencies** − Inconsistent or incorrect data may occur when processes manipulate shared data concurrently. For example, if multiple processes simultaneously update a database record, the final state of the record may be inconsistent or corrupted.

Synchronization Mechanisms

Mutual Exclusion

In order to avoid conflicts when processes need to use a shared resource mutual exclusion plays a crucial role in synchronizing them. Locks, semaphores and similar synchronization primitives are often utilized for ensuring exclusive access. By allowing only one process to access a shared resource at any given time, mutual exclusion prevents data races and ensures data consistency.

Semaphores

Semaphores are synchronization objects that maintain a count and allow or restrict access to resources based on the count value. Successful management of shared resources and coordinated process execution often require reliable tools such as semaphores. These flexible mechanisms allow for control over various resource requirements - whether binary (0 or 1) or non-binary (greater than 1). Ultimately. This approach ensures efficient use of available resources without compromising other vital processes.

Monitors

Monitors are higher-level synchronization constructs that encapsulate shared data and the procedures that operate on them. They ensure that only one process can execute a procedure within the monitor at any given time, preventing concurrent access to shared data. Monitors provide a structured and controlled way to synchronize concurrent processes, often using condition variables to manage process coordination.

Condition Variables

Condition variables are synchronization primitives used in conjunction with locks or monitors to enable processes to wait for specific conditions to be satisfied before proceeding. They provide a means for processes to communicate and coordinate their actions. Processes can wait on a condition variable until another process signals or broadcasts that the condition has been met.

Significance of Process Synchronization

Process synchronization is crucial for the following reasons −

- **Correctness** − Synchronization mechanisms prevent race conditions and ensure the correctness of shared data. By allowing only one process to access a shared resource at a time, synchronization mechanisms maintain data integrity and consistency.
- **Resource Management** − Synchronization allows for orderly access and efficient utilization of shared resources. It ensures that processes acquire resources in a controlled manner, preventing conflicts and optimizing resource utilization.
- **Deadlock Avoidance** − Synchronization techniques help prevent and resolve deadlocks, ensuring processes can make progress. By employing deadlock prevention or handling strategies, such as resource allocation graphs or deadlock detection algorithms, deadlocks can be avoided or resolved in a timely manner.
- **Coordination** − Synchronization enables processes to coordinate their actions and communicate effectively. It provides mechanisms for processes to wait for specific conditions, signal events, and synchronize their execution, facilitating cooperation and synchronization among concurrent processes.

Synchronization in Operating Systems

Examples of Synchronization in Operating Systems

Different operating systems provide mechanisms for process synchronization. Some common examples include −

- **POSIX Threads** − Provides thread synchronization primitives such as mutexes, condition variables, and barriers. These primitives allow threads to synchronize their actions and coordinate access to shared resources.
- **Java** − Offers built-in synchronization features, including synchronized blocks and the java. util.concurrent package. These features enable developers to synchronize access to shared resources in Java programs.
- **Windows** − Supports synchronization through primitives like critical sections, events, and semaphores. These synchronization mechanisms enable processes and threads in Windows-based systems to coordinate their actions and access shared resources safely.

Synchronization Policies and Algorithms

Operating systems employ various synchronization policies and algorithms to ensure efficient and fair process execution. These policies determine the order in which processes are granted access to resources and influence system performance and responsiveness. For example, scheduling algorithms determine how the operating system schedules processes and assigns them CPU time, taking into account factors like process priorities and resource availability.

Challenges and Considerations

Deadlock Prevention and Handling

Deadlocks can pose significant challenges in process synchronization. Employing deadlock prevention or handling strategies is essential to maintain system stability. Techniques such as

resource allocation graphs, deadlock detection algorithms, and deadlock avoidance strategies can be employed to prevent and resolve deadlocks effectively.

Performance Trade-offs

Synchronization mechanisms introduce overhead in terms of computation and system resources. Ensuring seamless coordination between multiple processes within the whole network architecture requires an inherent understanding of how this interconnection relates back not only from both a programming perspective but also from a high-performance computing perspective. Failure in obtaining this knowledge often leads to over synchronization or under synchronization resulting in sub optimal parallelization or worse yet deadlocks within concurrent software systems inside networked clusters. The best way forward then would be through judicious design and thorough analysis with optimization being at forefront whilst still keeping tabs on other variables that could affect overall results.

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.

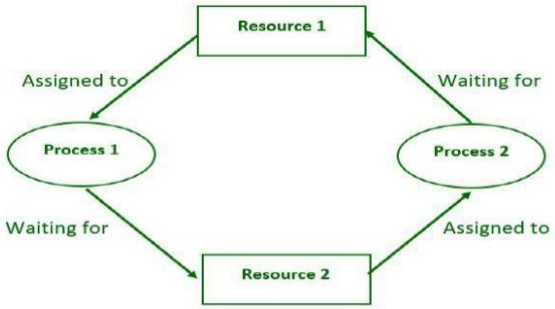


Figure: Deadlock in Operating system

System Model:

- For the purposes of deadlock discussion, a system can be modelled 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.

What is Deadlock in Operating System (OS)?

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

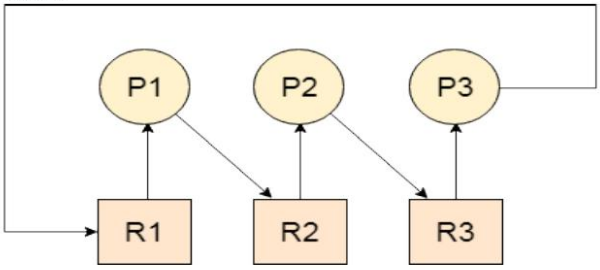
1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

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 pre-emption 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 pre-emption**

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.

Strategies for handling Deadlock

1. Deadlock Ignorance

Deadlock Ignorance is the most widely used approach among all the mechanism. This is being used by many operating systems mainly for end user uses. In this approach, the Operating system assumes that deadlock never occurs. It simply ignores deadlock. This approach is best suitable for a single end user system where User uses the system only for browsing and all other normal stuff.

There is always a trade-off between Correctness and performance. The operating systems like Windows and Linux mainly focus upon performance. However, the performance of the system decreases if it uses deadlock handling mechanism all the time if deadlock happens 1 out of 100 times, then it is completely unnecessary to use the deadlock handling mechanism all the time.

In these types of systems, the user has to simply restart the computer in the case of deadlock. Windows and Linux are mainly using this approach.

2. Deadlock prevention

Deadlock happens only when Mutual Exclusion, hold and wait, No pre-emption and circular wait holds simultaneously. If it is possible to violate one of the four conditions at any time then the deadlock can never occur in the system.

3. Deadlock avoidance

In deadlock avoidance, the operating system checks whether the system is in safe state or in unsafe state at every step which the operating system performs. The process continues until the system is in safe state. Once the system moves to unsafe state, the OS has to backtrack one step.

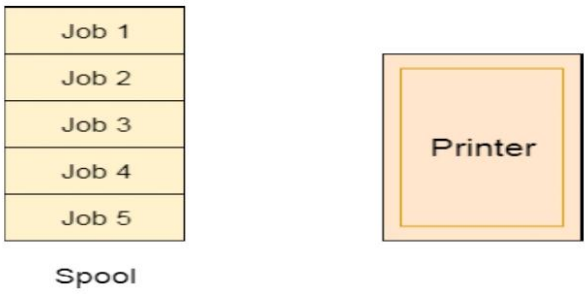
In simple words, The OS reviews each allocation so that the allocation doesn't cause the deadlock in the system.

4. Deadlock detection and recovery
This approach let the processes fall in deadlock and then periodically check whether deadlock occur in the system or not. If it occurs then it applies some of the recovery methods to the system to get rid of deadlock.
We will discuss deadlock detection and recovery later in more detail since it is a matter of discussion.

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 Pre-emption

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.

Deadlock avoidance

In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states.

In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.

The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need. The Deadlock avoidance algorithm examines the resource allocations so that there can never be a circular wait condition.

Safe and Unsafe States

The resource allocation state of a system can be defined by the instances of available and allocated resources, and the maximum instance of the resources demanded by the processes. A state of a system recorded at some random time is shown below.

Resources Assigned

Process	Type 1	Type 2	Type 3	Type 4
A	3	0	2	2
B	0	0	1	1
C	1	1	1	0
D	2	1	4	0

Resources still needed

Process	Type 1	Type 2	Type 3	Type 4
A	1	1	0	0
B	0	1	1	2
C	1	2	1	0
D	2	1	1	2

1. E = (7 6 8 4)
2. P = (6 2 8 3)
3. A = (1 4 0 1)

Above tables and vector E, P and A describes the resource allocation state of a system. There are 4 processes and 4 types of the resources in a system. Table 1 shows the instances of each resource assigned to each process.

Table 2 shows the instances of the resources; each process still needs. Vector E is the representation of total instances of each resource in the system.

Vector P represents the instances of resources that have been assigned to processes. Vector A represents the number of resources that are not in use.

A state of the system is called safe if the system can allocate all the resources requested by all the processes without entering into deadlock.

If the system cannot fulfil the request of all processes, then the state of the system is called unsafe.

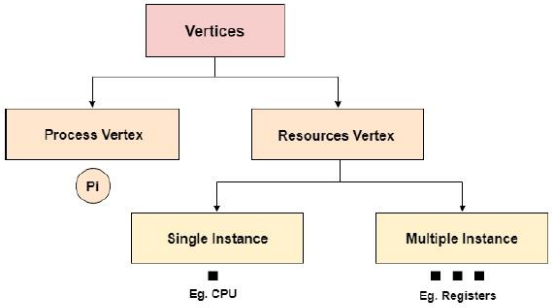
The key of Deadlock avoidance approach is when the request is made for resources then the request must only be approved in the case if the resulting state is also a safe state.

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	✓

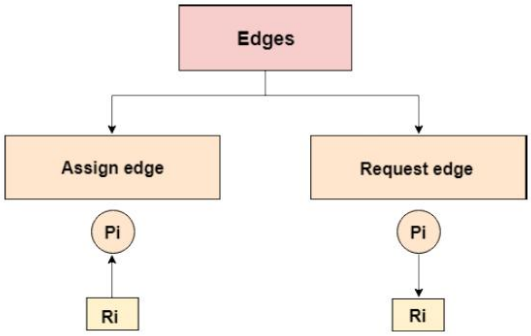
Among all the methods, violating Circular wait is the only approach that can be implemented practically.

Resource Allocation Graph

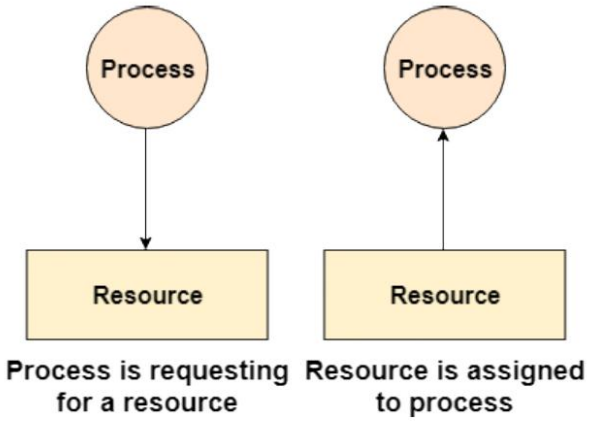
The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.
It also contains the information about all the instances of all the resources whether they are available or being used by the processes.
In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle. Let's see the types of vertices and edges in detail.



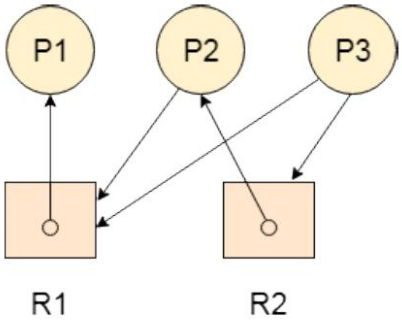
Vertices are mainly of two types, Resource and process. Each of them will be represented by a different shape. Circle represents process while rectangle represents resource.
A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.



Edges in RAG are also of two types; one represents assignment and other represents the wait of a process for a resource. The above image shows each of them.
A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process.
A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.



Example
Let's consider 3 processes P1, P2 and P3, and two types of resources R1 and R2. The resources are having 1 instance each.
According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.
The graph is deadlock free since no cycle is being formed in the graph.



Banker's Algorithm in Operating System (OS)

There is an algorithm called Banker's Algorithm used in removing deadlocks while dealing with the safe allocation of resources to processes in a computer system. It gives all the resources to each process and avoids resource allocation conflicts.

Overview of Banker's Algorithm

- The 'S-State' checks all the possible tests or activities before determining whether the resource should be allocated to any process or not.
- It enables the operating system to share resources among all the processes without creating deadlock or any critical situation.
- The algorithm is so named based on banking operations because it mimics the process through which a bank determines whether or not it can safely make loan approvals.

Real Life Example

Imagine having a bank with T amount of money and n account holders. At some point, whenever one of the account holders requests a loan:

- The bank withdraws the requested amount of cash from the total amount available for any further withdrawals.
- The bank checks if the cash that is available for withdrawal will be enough to cater to all future requests/withdraws.
- If there is enough money available (that is to say, the available cash is greater than T), he lends the loan.
- This ensures that the bank will not suffer operational problems when it receives subsequent applications.

Banker's Algorithm in Operating Systems

Likewise, in an operating system:

- When a new process is created, it needs to provide all the vital information, such as which processes are scheduled to run shortly, resource requests, and potential delays.
- This knowledge helps the OS decide which sequence of process executions needs to proceed to avoid any deadlock.
- Since the order of executions that should occur in order to prevent deadlocks is defined, the Banker's Algorithm is usually considered a deadlock avoidance or a deadlock detection algorithm in OS

Advantages

Following are the essential characteristics of the Banker's algorithm:

1. It contains various resources that meet the requirements of each process.
2. Each process should provide information to the operating system for upcoming resource requests, the number of resources, and how long the resources will be held.
3. It helps the operating system manage and control process requests for each type of resource in the computer system.
4. The algorithm has a Max resource attribute that represents indicates each process can hold the maximum number of resources in a system.
5. It means that in the Banker's Algorithm, the resources are granted only if there is no possibility of a deadlock when those resources are to be assigned. Thus, it ensures that the system runs at optimal performance.
6. The algorithm allows one to avoid the pointless holding of resources by any process since the algorithm actually checks whether the granting of resources is feasible or not.
7. Since the Banker's Algorithm analyzes all potential problems before assigning the resources, its implementation renders the OS more stable and reliable.

Disadvantages

1. It requires a fixed number of processes, and no additional processes can be started in the system while executing the process.
2. The algorithm does no longer allows the processes to exchange its maximum needs while processing its tasks.
3. Each process has to know and state their maximum resource requirement in advance for the system.
4. The number of resource requests can be granted in a finite time, but the time limit for allocating the resources is one year.
5. It could be pretty intricate to manage the algorithm, which is especially known in the case of systems with a vast quantity of processes and resources. This, consequently, translates to increased overhead.
6. In dynamic environments, often with frequently changing processes and resource needs, the Banker's Algorithm proves to be too inflexibly rigid to handle dynamic resource allocation well.
7. Since, at each step, the algorithm is checking for safe states before allocating resources, it tends to be very memory and processing-intensive and thus not very efficient for large systems.

When working with a banker's algorithm, it requests to know about three things:

1. How much each process can request for each resource in the system. It is denoted by the [**MAX**] request.
2. How much each process is currently holding each resource in a system. It is denoted by the [**ALLOCATED**] resource.
3. It represents the number of each resource currently available in the system. It is denoted by the [**AVAILABLE**] resource.

Important Data Structures in the Banker's Algorithm

Suppose n is the number of processes, and m is the number of each type of resource used in a computer system.

1. **Available**: It is an array of length 'm' that defines each type of resource available in the system. When Available[j] = K, means that 'K' instances of Resources type R[j] are available in the system.
2. **Max:** It is a [n x m] matrix that indicates each process P[i] can store the maximum number of resources R[j] (each type) in a system.

3. **Allocation:** It is a matrix of m x n orders that indicates the type of resources currently allocated to each process in the system. When Allocation [i, j] = K, it means that process P[i] is currently allocated K instances of Resources type R[j] in the system.
4. **Need:** It is an M x N matrix sequence representing the number of remaining resources for each process. When the Need[i] [j] = k, then process P[i] may require K more instances of resources type Rj to complete the assigned work. Nedd[i][j] = Max[i][j] - Allocation[i][j].
5. **Finish**: It is the vector of the order **m**. It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

The Banker's Algorithm is essentially a combination of two components:

- **Safety Algorithm:** Determines if the system is in a safe state, which means all the processes will eventually be allowed to execute without any deadlocks.
- **Resource Request Algorithm:** It decides whether it is safe to grant the request made by the other process for allocating more resources to itself without violating the safety.

Safety Algorithm

A safe state is said to be a state from which the system can execute some sequence of processes without deadlock. A system is said to be safe if all its processes can be executed safely in some sequence.

It is a safety algorithm used to check whether or not a system is in a safe state or follows the safe sequence in a banker's algorithm:

1. There are two vectors, Wok and Finish, of length m and n in a safety algorithm.

Initialization:

- **Work:** Initialize **work = Available,** where Available is the array that stores the number of available instances for each resource type.
- **Finish:** Initialize the Finish array for each process such that Finish[i] = false; for I = 0, 1, 2, 3, 4… n - 1., indicating that no processes have finished at the start.

2. Check the availability status for each type of resource [i], such as:

1.
2. Need[i] <= Work
3. Finish[i] == false

If the i does not exist, go to step 4.

3. If a process i satisfies the conditions in step 2, simulate the allocation by updating the Work vector:

1.
2. WorkWork = Work +Allocation(i) // to get new resource allocation

This adds the resources currently allocated to process I back to the Work vector, stimulating the release of resources after the process has finished.

1.
2. Finish[i] = true

Go to step 2 to check the status of resource availability for the next process.

4. If Finish[i] == true, it means that the system is safe for all processes.

Resource Request Algorithm

A Resource Request Algorithm A description of how the system behaves when a process requests a resource needs to be made. The algorithm will check whether the requested resources can be allocated without causing deadlock. The request can then be portrayed as a request matrix.

A resource request algorithm checks how a system will behave when a process makes each type of resource request in a system as a request matrix.
Let create a resource request array R[i] for each process P[i]. If the Resource Request$_i$ [j] equal to 'K', which means the process P[i] requires 'k' instances of Resources type R[j] in the system.
Steps of the Resource Request Algorithm:

1. Check if the Request is within the Process's Maximum Claim:

At every request of resources done by a process P[i], the algorithm checks if the number of resources requested is less than or equal to the Need of that process:

1. If Request[i]≤Need[i](for all resources)

If the condition holds, then it proceeds to the next one. Otherwise, process P[i] has exceeded its maximum claim; the request is denied.
If the process exceeds its maximum claim, this is then a violation of the rules of the system; the process cannot be allowed to proceed.

2. Check if the Requested Resources are Available:

The system now checks the availability of the requested resources in sufficient quantity. This is done by comparing the request [i] with the Available resources:

1. If Request[i]≤Available(for all resources)

If this condition is satisfied, proceed to the next step. If not, then P[i] must wait until the required resources become available.
This makes sure the system does not allocate resources it does not currently have, which otherwise could lead to deadlock or unsafe states.

3. Simulate Resource Allocation:

If available, the system allocates temporary requested resources to process P[i] by modifying the Available, Allocation, and Need matrices.

1. AvailableAvailable=Available−Request[i]
2. Allocation[i]=Allocation[i]+Request[i]
3. Need[i]=Need[i]−Request[i]

4. Check System Safety:

- After the allocation of the requested resources, the system runs the Safety Algorithm to check whether the system continues to be in a safe state after that allocation.
- The resources are permanently allocated if the system is in a safe state to process P[i].
- Suppose the system is entering an unsafe state. In that case, process P[i] must wait, and the system must roll back to its earlier state by recovering its original values of Available, Allocation, and Need matrices.

Example: Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resources types: A has 10, B has 5 and the resource type C has 7 instances.

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

Answer the following questions using the banker's algorithm:

1. What is the reference of the need matrix?
2. Determine if the system is safe or not.

3. What will happen if the resource request (1, 0, 0) for process P1 can the system accept this request immediately?

Ans. 1: The context of the need matrix is as follows:

Need [i] = Max [i] - Allocation [i]
Need for P1: (7, 5, 3) - (0, 1, 0) = 7, 4, 3
Need for P2: (3, 2, 2) - (2, 0, 0) = 1, 2, 2
Need for P3: (9, 0, 2) - (3, 0, 2) = 6, 0, 0
Need for P4: (2, 2, 2) - (2, 1, 1) = 0, 1, 1
Need for P5: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

Process	Need		
	A	B	C
P1	7	4	3
P2	1	2	2
P3	6	0	0
P4	0	1	1
P5	4	3	1

Hence, we created the context of need matrix.
Ans. 2: Apply the Banker's Algorithm:
Available Resources of A, B and C are 3, 3, and 2.
Now we check if each type of resource request is available for each process.
Step 1: For Process P1:
Need <= Available
7, 4, 3 <= 3, 3, 2 condition is **false**.
So, we examine another process, P2.
Step 2: For Process P2:
Need <= Available
1, 2, 2 <= 3, 3, 2 condition **true**
New available = available + Allocation
(3, 3, 2) + (2, 0, 0) => 5, 3, 2

Similarly, we examine another process P3.
Step 3: For Process P3:
P3 Need <= Available
6, 0, 0 < = 5, 3, 2 condition is **false**.

Similarly, we examine another process, P4.
Step 4: For Process P4:
P4 Need <= Available
0, 1, 1 <= 5, 3, 2 condition is **true**
New Available resource = Available + Allocation
5, 3, 2 + 2, 1, 1 => 7, 4, 3

Similarly, we examine another process P5.
Step 5: For Process P5:
P5 Need <= Available
4, 3, 1 <= 7, 4, 3 condition is **true**

New available resource = Available + Allocation
7, 4, 3 + 0, 0, 2 => 7, 4, 5

Now, we again examine each type of resource request for processes P1 and P3.
Step 6: For Process P1:
P1 Need <= Available
7, 4, 3 <= 7, 4, 5 condition is **true**
New Available Resource = Available + Allocation
7, 4, 5 + 0, 1, 0 => 7, 5, 5

So, we examine another process P2.
Step 7: For Process P3:
P3 Need <= Available
6, 0, 0 <= 7, 5, 5 condition is true
New Available Resource = Available + Allocation
7, 5, 5 + 3, 0, 2 => 10, 5, 7

Hence, we execute the banker's algorithm to find the safe state and the safe sequence like P2, P4, P5, P1 and P3.

Ans. 3: Whether the system can accommodate the resource request (1,0,0)
for the process, P1 can be checked by using Banker's Algorithm as follows:
1. Checking if the request is within the maximum claim of P1
If the resources requested: (1,0,0) are less than or equal to P1's Need, then go ahead; otherwise, reject.
2. Check whether the system has enough available resources to handle the request
(1,0,0). If the resources are available, go; otherwise, P1 should wait.
3. Simulate resource allocation:
Allocate the resources to P1 temporarily and check if the system is safe or not using the Safety Algorithm. If the system is safe, the request is granted. Otherwise, cancel the allocation and let P1 wait.