1.Proccess Synchronization

1.1 Introduction

Process Synchronization is a mechanism in operating systems used to manage the execution of multiple processes that access shared resources. Its main purpose is to ensure data consistency, prevent race conditions and avoid deadlocks in a multi-process environment.

On the basis of synchronization, processes are categorized as one of the following two types:

Independent Process: The execution of one process does not affect the execution of other processes.

Cooperative Process: A process that can affect or be affected by other processes executing in the system.

Process Synchronization is the coordination of multiple cooperating processes in a system to ensure controlled access to shared resources, thereby preventing race conditions and other synchronization problems.

Improper Synchronization in Inter Process Communication Environment leads to following problems like

Inconsistency: When two or more processes access shared data at the same time without proper synchronization. This can lead to conflicting changes, where one process's update is overwritten by another, causing the data to become unreliable and incorrect.

Loss of Data: Loss of data occurs when multiple processes try to write or modify the same shared resource without coordination. If one process overwrites the data before another process finishes, important information can be lost, leading to incomplete or corrupted data.

Deadlock: Lack of proper Synchronization leads to Deadlock which means that two or more processes get stuck, each waiting for the other to release a resource. Because none of the processes can continue, the system becomes unresponsive and none of the processes can complete their tasks.

1.2 Race Condition

A race condition is a situation that may occur inside a critical section. This happens when the result Of multiple thread execution in critical section differs according to the order in which the threads execute.

Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

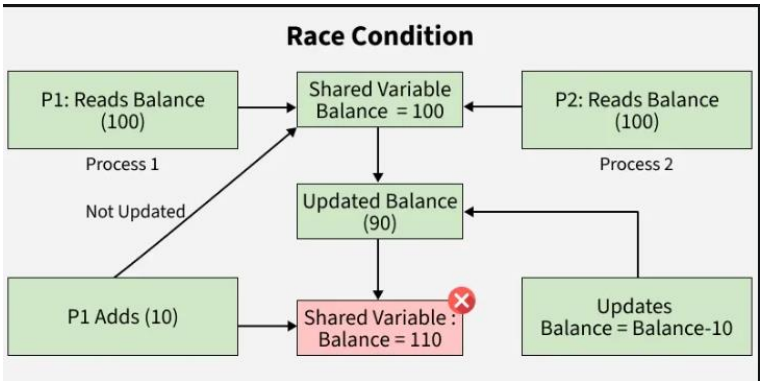
Causes of Race Conditions

- **Simultaneous Access**: when two or more processes try to read or write the same shared resource at the same time.
- **Non-Atomic Updates**: Operations like increment or decrement are not indivisible.
- **Lack of Synchronization**: No mechanisms like locks, semaphores, or monitors are used to control access.
- **Improper Scheduling**: OS scheduler interrupts processes at critical moments.

Example: Two Processes Updating a Shared Variable

Let's take a shared variable balance = 100 and two processes P1 and P2:
P1 wants to add 10 to balance.
P2 wants to subtract 10 from balance.
Scenario without synchronization:

Explanation:



- P1 reads balance = 100 and prepares to add 10.
- Before P1 updates the balance with the new value (110), it is interrupted by the process P2.
- P2, unaware of P1's action (of adding 10), reads the balance as 100 (incorrect) and prepares to subtract 10.
- After subtracting, P2 updates the balance to 90 and then P1 resumes and writes the balance as 110 which is incorrect now.

1.3 The Critical Section Problem

Consider a system consisting of n processes. Each process has a segment code, called a critical section, in which the process may be changing common variables, updating table, writing file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. The critical section problem is to design a protocol such that the processes can cooperate. Each process must request permission to enter its critical section. The section of the code that implements this request is the entry section. The critical section is may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process Pi is shown below:

```
do{
    // entry section
    Criticalsection
    //exit section
    Remaindersection
}while (true);
```

A solution to the critical section problem must satisfy the following three requirements.

1. **Mutual exclusion.** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next.
3. **Bounded Waiting:** whenever processes places their requests to enter their critical sections – the permission will be granted based on how many times the respective processes are allowed to enter their critical sections.

Peterson's Solution

Peterson's Solution is a classical software based solution to the critical section problem. In Peterson's solution, we have two shared variables:

- Boolean flag[i]: Initialized to FALSE, initially no one is interested in entering the critical section.
- turn :The process whose turn is to enter the critical section.

```
do{
        flag[i] =TRUE; turn = j;
        while (flag[j] && turn == j);
        //critical section
        flag[i] = FALSE;
        //remainder section
    }while (TRUE);
```

Peterson's Solution preserves all three conditions:

- Mutual Exclusion is assured as only one process can access the critical section at anytime.
- Progress is also assured, as a process out side the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's Solution

- It involves Busy waiting
- It is limited to 2 processes.

1.4 Semaphores

The hardware based solutions to the critical section problem are complicated for application programmers to use. To overcome this difficulty, we can use synchronization tool called a semaphore. A semaphore S is a synchronizing variable that can hold an integer value. It can be initialized to as specific integer. Semaphore as a control variable, can be accessed via to standard atomic operations namely **wait** and **signal**.

A semaphore works using two fundamental operations:

1. Wait (s)
   - Decreases the semaphore value.
   - If the value becomes negative, the process is blocked until the resource becomes available.

```
wait (S)
{
        while (S <= 0);
       // do nothing
        S--;
}
```

2. Signal (s)
   - Increases the semaphore value.
   - If there are waiting processes, one of them gets unblocked.

```
signal(S)
{
S++;
}
```

A semaphore does not permit the simultaneous modification of semaphore value S by more than one process, at a time. Therefore two or more processes can be synchronized by sharing a common semaphore variable say S.

The solution to critical section problem can be provided by implementing mutual exclusion with semaphore variables. The processes will share a common semaphore variable called mutex. The variable mutex is initialized to an integer value 1. Now the structure of each process Pi for mutual exclusion implementation can be given by the following code segment.

```
do {
        wait (mutex);
        // critical section signal
        (mutex);
        // remainder section
    }while(true);
```

Semaphore Types:

Semaphores can be classified into the following types:

1. **Counting Semaphores:** Counting semaphores can be used to control access to a given resource. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait () operation on the semaphore. When a process releases a resource, it performs, it performs a signal () operation. When the count becomes for semaphore goes to 0, all resources are in use. After that, processes that wish to use a resource will block until the count becomes greater than 0. Example: Managing access to a pool of 5 printers.

2. **Binary Semaphore:** it can store the binary values either 0 or 1. Works like a lock: either the resource is free (1) or busy (0). Example: Managing access to a single critical section.

1.5 Producer Consumer Solution using Semaphores

The Producer-Consumer problem is a classic example of a synchronization problem in operating systems. It demonstrates how processes or threads can safely share resources without conflicts. This problem belongs to the process synchronization domain, specifically dealing with coordination between multiple processes sharing a common buffer.

In this problem, we have:

- **Producers**: Generate data items and place them in a shared buffer.
- **Consumers**: Remove and process data items from the buffer.

The main challenge is to ensure:

- A producer does not add data to a full buffer.
- A consumer does not remove data from an empty buffer.
- Multiple producers and consumers do not access the buffer simultaneously, preventing race conditions.

Problem Statement

Consider a fixed-size buffer shared between a producer and a consumer.

- The producer generates an item and places it in the buffer.
- The consumer removes an item from the buffer.

The buffer is the critical section. At any moment:

- A producer cannot place an item if the buffer is full.
- A consumer cannot remove an item if the buffer is empty.

To manage this, we use three semaphores:

- mutex – ensures mutual exclusion when accessing the buffer.
- full – counts the number of filled slots in the buffer.
- empty – counts the number of empty slots in the buffer.

Semaphore Initialization

```
mutex = 1; // binary semaphore for mutual exclusion
full = 0; // initially no filled slots
empty = n; // buffer size
```

Producer

```
do {
    // Produce an item
    wait(empty); // Check for empty slot
    wait(mutex); // Enter critical section
    // Place item in buffer
    signal(mutex); // Exit critical section
    signal(full); // Increase number of full slots
} while (true);
```

Consumer

```
do {
    wait(full); // Check for filled slot
    wait(mutex); // Enter critical section
    // Remove item from buffer
    signal(mutex); // Exit critical section
    signal(empty); // Increase number of empty slots
} while (true);
```

- Empty ensures that producers don't overfill the buffer.
- Full ensures that consumers don't consume from an empty buffer.
- Mutex ensures mutual exclusion, so only o
- ne process accesses the buffer at a time.

1.6 The Dining–Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice and five plates across the table. The table surface also consists of five single chopsticks arranged adjacent to plates. Now the problem starts when philosophers became hungry and would like to eat rice.

When philosophers are busy in thinking about research etc. they never interact with each other. But they would attempt at least to pick up two chopsticks or spoons that are adjacent to their plate's
i.e. one chopstick to their left and the other to their right. It is assumed that eating with single chopstick is ruled out.

In other words, synchronization problem exit and in turn it may lead to deadlock and starvation. For instance, a philosopher may hold quickly one chopstick or spoon towards his right from his right hand. When he is looking for the next chopsticktowards his adjacent left, it might be picked up by another philosopher who is sitting aside to his left. Some of them may pick up both chopsticks at a time. Others may get only one chopstick

In this situation the dining philosopher problem must adopt a solution such that sufficient care will be ensured to overcome the possibility of philosopher's death due to starvation.

To conclude, dining philosopher problem could be solved by using semaphore class chopstick in the philosopher processes. This solution provides necessary synchronization and ensures that no two neighboring philosophers are eating at the same time.

```
semaphore chopstick[5]; do {
        wait(chopstick[i]);
        wait(chopstick[(i+1)%5]);
        . . .
        /*eat forawhile */
        . . .
        signal(chopstick[i]);
        signal(chopstick[(i+1)%5]);
        . . .
        /*think forawhile */
. . .
} while (true);
```

2.Deadlocks

A deadlock is a situation in which some processes wait for each other's actions in definitely. Processes involved in a deadlock remain blocked permanently.

2.1 System Model

In the context of resource usage,the computer resources can be categorized in two types:

1. Preemptable Resource.
2. Non–preemptable Resource.

A preemptable resource is a resource that can be taken away from the running process that holds the resource without causing system failure. For example: system memory is a preemptable resource.

Non – preemptable resource cannot be taken away from the running process that holds the resource without causing system failure. For example CD writer drives, Printers are non – preemptable.

Accordingly, a process utilizing the resource must follow the sequence of events as mentioned below:

1. Request: The process requests the resource. If the request cannot be granted immediately,then the request in g process must wait until it can acquire the resource.
2. Use:The process can operate on the resource(forexample,if the resource is a printer, the process can print on the printer).
3. Release:The process releases the resource.

The request and release of resource are system calls, examples are the request() and release() device,open() and close() file, and allocate() and free() memory system calls. Request and

Release of resources that are not managed by the operating system can be accomplished through wait() and signal() operations on semaphores. A system table records whether each resource is free or allocated. If a process requests a resource that is currently allocated to another process,it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

2.2 Deadlock Characterization

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

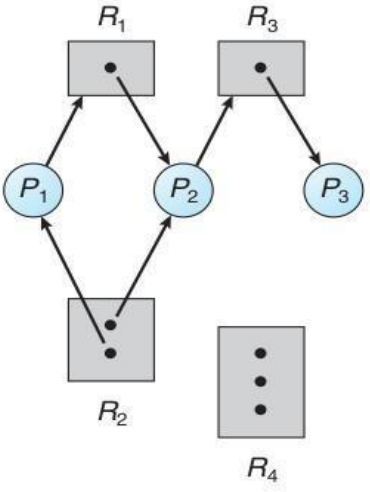
1. **Mutual Exclusion**: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests the same resource then it is forced to wait until that resource has been released.
2. **Hold and Wait**:A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No Preemption**: Resource cannot be preempted; that is a resource is released by the process, only after the completion of its task.
4. **Circular wait**: A set $\{P_0, P_1, \ldots P_n\}$ of waiting processes must exist such that $P_0$is waiting for a resource held by P_1, $P_1$is waiting for a resource held by P_2, ...$P_{n-1}$is waiting for aresource held by P_n, and $P_n$is waiting for a resource held by P_0.

2.3 Resource– Allocation Graph

Deadlock 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. the set of vertices V is partitioned into two different types of nodes: the set of all active processes in the system $P = \{P_0, P_1, \ldots P_n\}$ and a set consisting of all resource types in the system $R = \{R_0, R_1, \ldots R_m\}$.

A directed edge from process Pi to resource Rj is denoted as Pi → Rj and it indicates that process Pi has requested the resource Rj and is currently waiting for that resource. A directed edge from resource Rj to process Pi is denoted as Rj → Pi. It indicates that an instance of resource type Rj has been allocated to process Pi.

Pictorially, we represent each process Pi as a circle while each resource is represented by a rectangle. Since resource type Rj may have more than one instance, we represent each such instance as a dot within the rectangle. The resource allocation graph is shown in the following figure.



Resource-allocation graph.

From the above figure the process states are:

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an

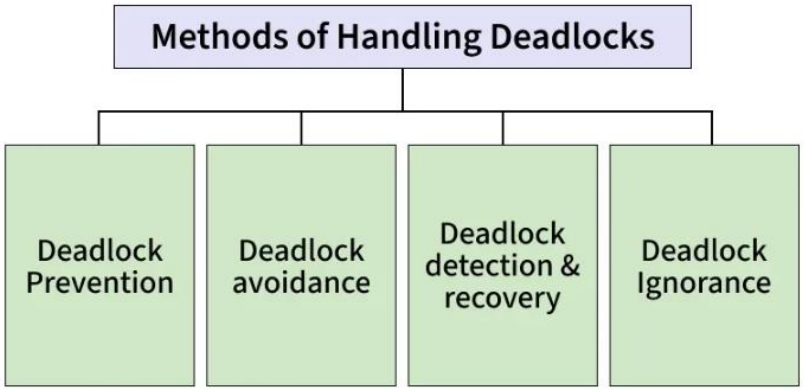
instance of R_3.

- Process P_3 is holding an instance of R_3.

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.

2.4 Methods of Handling Deadlocks

Deadlock handling methods are strategies used in operating systems to ensure processes do not remain permanently blocked, maintaining smooth execution and system reliability.
There are four approaches to dealing with deadlocks.



2.5 Deadlock Prevention

We know that, for a deadlock to occur in a system – all four conditions namely – Mutual Exclusion, Hold and wait, Circular Wait and No preemption must hold at the same time. Therefore Deadlock prevention strategies provide a set of methods to ensure that at least one of these conditions is always false.

Mutual Exclusion: The mutual exclusion condition must hold for non-sharable resources. For example Printer, CD WR etc. in such cases, mutual exclusion ensures that a resource can have an exclusive access by only one process at a time.

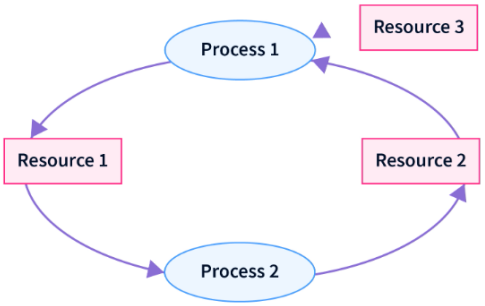
In general it is not possible to deny mutual exclusion for every resource type. Since some resources are intrinsically non-sharable, it is not worth while to invalidate mutual exclusion for prevention of deadlock.

Hold and Wait condition: It can be violated by employing the following two protocols:

1. A process must request all of its required resources when it is created and before it begins execution.
2. A process must place a request for resources if and only if it has not acquired any resources at present.

These two protocols introduce the following D**isadvantages:**

- Resource utilization may be low,since resources may be allocated but unused for a long period.
- 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.

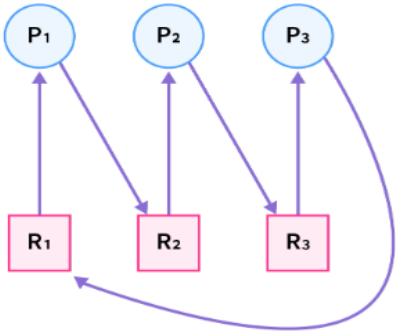


No Preemption: To ensure that this condition does not hold, we can use the following protocol.

1. If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are preempted. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it regains it old resources along with new ones that it is requesting.
2. If a process requests some resources, we first check whether they are available. If they are, we allocated them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources form the waiting process and allocated them to the requesting process. If the resources are neither available nor held by awaiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted provided that the other process might request the same.

Circular Wait

In circular wait, two or more processes wait for resources in a circular order. We can understand this better by the diagram given below:



To eliminate circular wait, we assign a priority to each resource. A process can only request resources in increasing order of priority. **In the example above,** process **P3** is requesting resource **R1**, which has a number lower than resource **R3** which is already allocated to process **P3**. So this request is invalid and cannot be made, as **R1** is already allocated to process **P1**.

2.6 Deadlock Avoidance

The deadlock prevention methods introduce the problems such as low device utilization and reduced system throughput as side effects of preventing deadlock. An alternative method for avoiding deadlocks is to get additional information about how resources are to be requested. Deadlock can be avoided by choosing algorithms that would determine whether allocating a resource is safe (deadlock will not occur) or unsafe (deadlock will occur).

A simplest and most useful algorithm for Deadlock Avoidance demands the following information in advance. That is, when process is created it must declare its maximum resource requirements. With this information, the algorithm can be developed such that the system will never encounter a deadlock situation. A deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that a circular wait condition can never exist.

2.6.1 Safe State

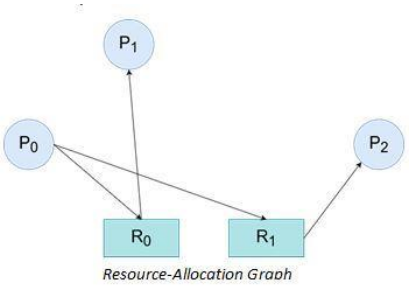
A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock. A sequence of processes$\{P_1, P_2, \text{---------} P_n\}$ is a safe sequence for the current allocation
state if for each P_i,the resource requests that P_i can be satisfied by the currently available resources

plus the resources held by all P_j,with j<i. In this situation,if the resources required by P_i are not available immediately, then $P_i$can wait until all $P_j$have finished. When they have finished, $P_i$can obtain all of its needed resources, complete its task, return its allocated resources and terminate. When P_i, terminates, P_{i+1}can obtain its needed resources, and so on. If no such sequence exists, then the system is said to be in unsafe.

A safe state is not a deadlocked state and not all unsafe states are deadlocks.

2.6.2 Resource-Allocation Graph Algorithm

Resource Allocation Graph (RAG) is a popular technique used for deadlock avoidance. It is a directed graph that represents the processes in the system, the resources available, and the relationships between them. A process node in the RAG has two types of edges, request edges, and assignment edges. A request edge represents a request by a process for a resource, while an assignment edge represents the assignment of a resource to a process.



Resource-Allocation Graph

To determine whether the system is in a safe state or not, the RAG is analyzed to check for cycles. If there is a cycle in the graph, it means that the system is in an unsafe state, and granting a resource request can lead to a deadlock. In contrast, if there are no cycles in the graph, it means that the system is in a safe state, and resource allocation can proceed without causing a deadlock.

The RAG technique is straightforward to implement and provides a clear visual representation of the processes and resources in the system. It is also an effective way to identify the cause of a deadlock if one occurs. However, one of the main limitations of the RAG technique is that it assumes that all resources in the system are allocated at the start of the analysis. This assumption can be unrealistic in practice, where resource allocation can change dynamically during system operation. Therefore, other techniques such as the Banker's Algorithm are used to overcome this limitation.

2.6.3 Banker's Algorithm

Banker algorithm used to **avoid deadlock** and **allocate resources** safely to each process in the computer system. The '**S-State'** examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes.

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.

Following are the important data structures terms applied in the banker's algorithm as follows:

Suppose n is the number of processes, and m is the number of each type of resource used in acomputer 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.WhentheNeed[i][j]=k,thenprocessP[i]mayrequireKmoreinstancesofresourcestype 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 the combination of the safety algorithm and the resource request algorithm to control the processes and avoid deadlock in a system:

Safety Algorithm

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.
   Initialize: Work= Available
   Finish[i] =false; for I=0, 1, 2,3, 4… n-1.

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

   Need[i]<=Work
   Finish[i]==false
   If the i does not exist, go to step 4.

3. Work=Work +Allocation(i)//to get new resource

   allocation 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 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 arrayR[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.

1. When the number of **requested resources** of each type is less than the **Need** resources, go to step 2 and if the condition fails, which means that the process P[i] exceeds its maximum claim for the resource. As the expression suggests:

   If Request(i)<=Need Go to step 2;

2. And when the number of requested resources of each type is less than the available resource for each process, go to step (3). As the expression suggests:

   If Request(i)<=Available
   Else ProcessP[i] must wait for the resources in ce it is not available for use.

3. When the requested resource is allocated to the process by changing state:

   Available = Available - Request
   Allocation(i)=Allocation(i)+Request(i)
   $Need_i = Need_i - Request_i$

When the resource allocation state is safe,its resources are allocated to the process P(i).And if the new state is unsafe, the Process P (i) has to wait for each type of Request R(i) and restore theold resource-allocation state.

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.

Question1.What will be the content of the Need matrix?

Question2.Is the system in a safe state ? If Yes, then what is the safe sequence?

Question3.What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?

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			

Solution:

1. 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 forP2: (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

Question 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.

Step1: 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,weexamineanotherprocessP3.

Step 3: For Process P3:

P3Need<=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<= 5,3,2conditionis **true**

New

Availableresource=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<= 7,4,3conditionis **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 ProcessP1: P1 Need <= Available

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:

P3Need<=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.

Question3:

For granting the Request(1,0,2), first we have to check that **Request <= Available**, that is (1, 0, 2) <=(3, 3, 2), since the condition is true. So the process P2 gets the request immediately.

E

A B C

$Request_1$ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm

Step 1

1, 0, 2 1, 2, 2

$Request_1$ < $Need_1$ ✔

Step 2

1, 0, 2 3, 3, 2

$Request_1$ < Available ✔

Step 3

Available = Available – $Request_1$

$Allocation_1$ = $Allocation_1$ + $Request_1$

$Need_1$ = $Need_1$ - $Request_1$

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

Allocation= Allocation + request

Allocation= 2,0,0,+ 1,0,2= 3,0,2

Need= need-request

Need= 1,2,2 – 1,0,2

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm onthe above data structures.

ApplytheBanker's Algorithm:

Available Resources of A,B and C are 2, 3,and 0.

Now we check if each type of resource request is available for each process.

Step1: For Process P1:

Need<= Available

7, 4, 3 <= 2, 3, 0 condition is **false**.**So,we examine another process, P2.**

Step 2: For Process P2:

Need<= Available

0,2,0<=2, 3, 0 condition**true**

New available=available+Allocation (2,3,0) + (3, 0, 2) => 5, 3, 2

Similarly, we examine another process P3. Step 3: For Process P3:

P3Need<= Available

6, 0, 0 < = 5, 3, 2 condition is **false**. **Similarly,we examine another process,P4. Step 4:** For Process P4:

P4Need<= Available

0, 1, 1 <=5, 3, 2 condition is**true**

NewAvailableresource=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.

Step6: For Process P1:

P1Need<= 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.

Step7: 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.