ASSIGNMENT

By

Nishant patyad

2022a1r021

3rd

Computer Science

Model Institute of Engineering & Technology (Autonomous)

(Permanently Affiliated to the University of Jammu, Accredited by NAAC with “A” Grade) Jammu, India

2023

ASSIGNMENT

Subject Code: com-302

Subject: Operating system

Due Date: 4 november,2023

Question Number	**Course Outcomes**	**Blooms' Level**	**Maximum Marks**	**Marks Obtain**
Q1	CO 4	3-6	10	
Q2	CO 5	3-6	10	
Total Marks			20	
Faculty:Dr.Mekhla Sharma Signature.Email:mekhla.cse@mietjammu.in				

TASK 1:

Analyze the differences between mutex locks and semaphores in terms of functionality and use cases for synchronization. Explain situations in which you would choose one over the other and provide specific examples to support your analysis.

Mutex is a specific kind of binary semaphore that is used to provide a locking mechanism. It stands for Mutual Exclusion Object. Mutex is mainly used to provide mutual exclusion to a specific portion of the code so that the process can execute and work with a particular section of the code at a particular time.

A semaphore is a non-negative integer variable that is shared between various threads. Semaphore works upon signaling mechanism, in this a thread can be signaled by another thread.

Mutex locks and semaphores are both synchronization mechanisms used in concurrent programming to control access to shared resources. While they serve similar purposes, there are key differences in terms of functionality and use cases.

Mutex Locks:

1. **Functionality:**
    1. **Binary Semaphores:** Mutex locks are often implemented as binary semaphores with a value of 1 or 0. This means that only one thread can acquire the lock at a time.
    2. **Exclusive Ownership:** Mutexes provide exclusive ownership, meaning that only the thread that acquired the lock can release it.
2. **Use Cases:**
    1. **Mutual Exclusion:** Mutex locks are primarily used for ensuring mutual exclusion, allowing only one thread to access a critical section of code or a shared resource at a time.
    2. **Short-Term Locking:** Mutexes are suitable for scenarios where the locking duration is expected to be short, and a thread needs to quickly acquire and release the lock.

3. **Example:**

Critical Sections: Protecting a critical section of code where data integrity must be maintained. For instance, in a multithreaded program, if multiple threads are updating a shared data structure, a mutex can be used to ensure that only one thread accesses it at any given time.

Semaphores:

1. **Functionality:**

    1. **Counting Semaphores:** Unlike mutex locks, semaphores can have values greater than 1. They are more flexible and can be used to control access by multiple threads simultaneously.

    2. **General Synchronization:** Semaphores can be used for more general synchronization purposes, not limited to mutual exclusion.

2. **Use Cases:**

    1. **Resource Management:** Semaphores are often used to manage resources where there are a limited number of available slots. For example, a semaphore with a count of 5 could represent five available resources that multiple threads are competing for.

    2. **Producer-Consumer Problem:** Semaphores are useful in scenarios where there are multiple producers and consumers. They can be used to control access to a shared buffer.

3. **Example:**

Printers and Permits: Consider a scenario where there are multiple printers, and a semaphore is used to control access to them. If the semaphore count is the number of available printers, each thread (representing a print job) will acquire a permit (decrementing the semaphore count) before printing and release it (incrementing the count) after finishing.

Choose Mutex Locks When:

- In a database management system, when multiple threads or processes attempt to modify a critical data structure, you want to ensure that only one thread has exclusive access to that data at a time. A mutex lock can be used to enforce this exclusive access within the critical section of code.

- Consider a multithreaded program where threads are updating a shared variable. If the critical section is short and involves minimal computation, a mutex lock is a lightweight option. It allows threads to quickly acquire and release the lock, minimizing contention.

- In situations where you have a small number of resources and a simple resource hierarchy, a mutex may be preferred to avoid the complexities associated with semaphores. For instance, managing access to a single shared file can be efficiently handled with a mutex.

Choose Semaphores When:

- In a scenario where multiple threads need simultaneous access to a shared resource, such as a fixed-size buffer, a semaphore can be employed. Each thread, upon entering the critical section, decrements the semaphore count, and upon leaving, it increments the count. This allows for controlled concurrency.

- Consider a scenario where you have a pool of worker threads and a pool of tasks. You want to limit the number of concurrent tasks being processed. A semaphore with the count representing the number of available worker threads allows you to manage and control the resource usage dynamically.

- In scenarios involving complex synchronization requirements, such as avoiding both deadlock and starvation, a semaphore provides greater flexibility. For instance, in a dining philosophers problem where philosophers represent threads and forks represent resources, a semaphore can be used to control access to the limited resource of forks.

Choosing Based on Use Case:

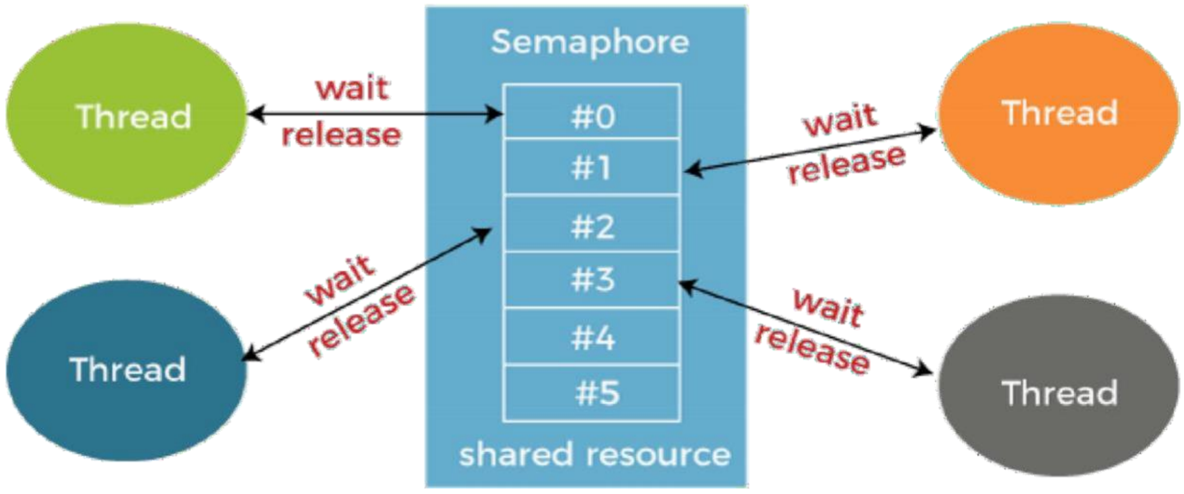
- If the primary concern is exclusive access and simplicity, choose a mutex.
- If there is a need for more sophisticated resource management or concurrent access, opt for semaphores.

Choosing Between Mutex Locks and Semaphores:

- **Mutual Exclusion vs. Resource Management:** If the goal is to protect a critical section of code, use a mutex. If you need to control access to a finite set of resources, a semaphore is more appropriate.

Binary vs. Counting: If you only need a binary (0 or 1) state to control access, a mutex is simpler. If you need to manage multiple resources or permit multiple threads to access a shared resource concurrently, a semaphore is more suitable.

It's essential to consider the specific requirements and characteristics of your application when making a decision, as the choice between mutex locks and semaphores depends on the nuances of the synchronization needs in your particular scenario.



TASK 2:

Write a program that implements the Banker's algorithm for deadlock avoidance.

Simulate multiple processes making resource requests and releases.

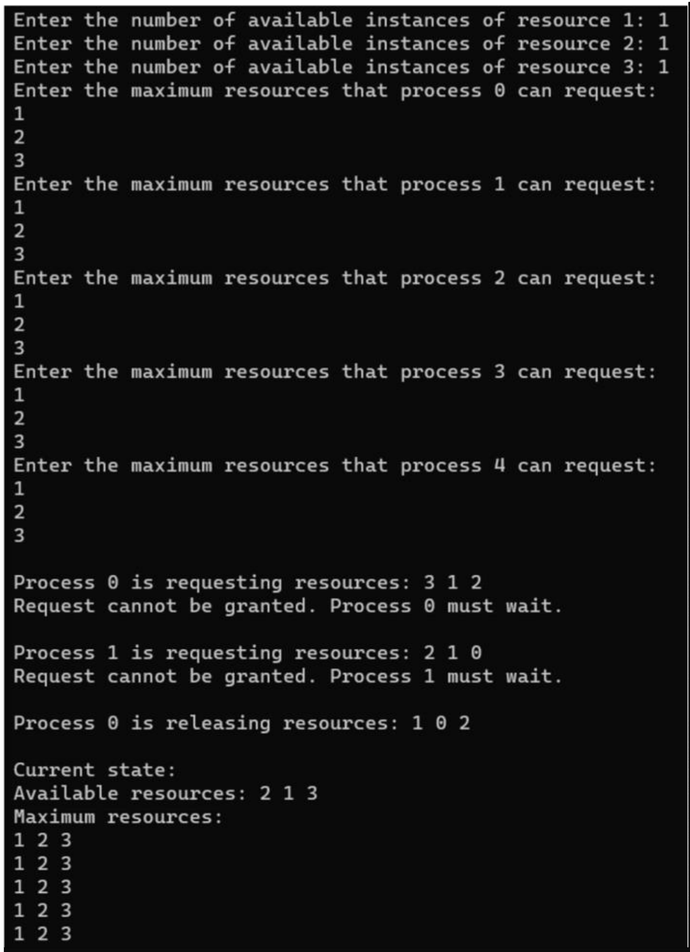
Demonstrate how the algorithm ensures safe states and prevents deadlocks.

Discuss the advantages and limitations of the Banker's algorithm.

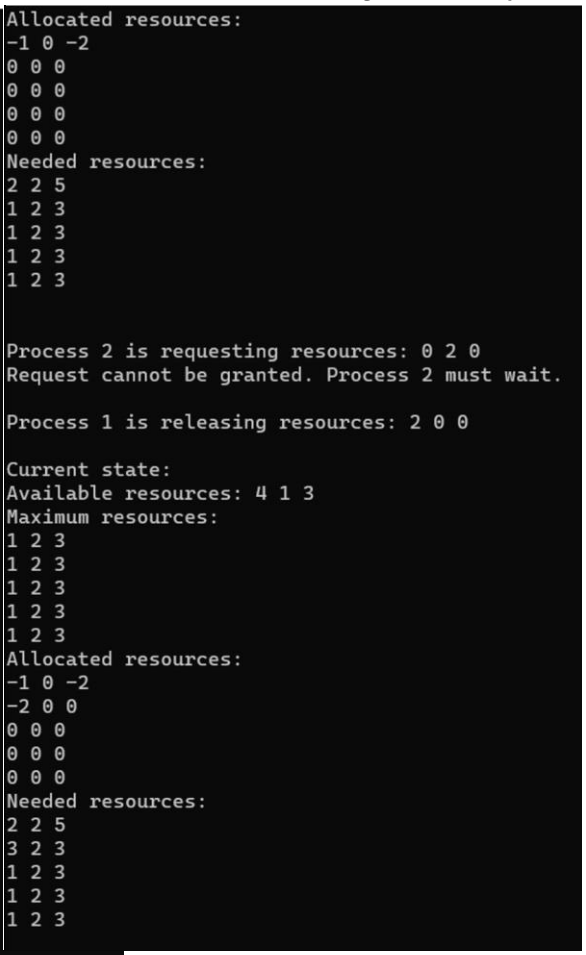
```
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

int available[MAX_RESOURCES];
int maximum[MAX_PROCESSES][MAX_RESOURCES]; int
allocated[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];

// Function prototypes void initialize(); void
requestResources(int process, int request[]); void
releaseResources(int process, int release[]); bool
isSafeState();
void printState();

int main() {
   initialize();

   // Simulate processes making resource requests and releases
requestResources(0, (int[]){3, 1, 2});     requestResources(1,
(int[]){2, 1, 0});     releaseResources(0, (int[]){1, 0, 2});
requestResources(2, (int[]){0, 2, 0});     releaseResources(1,
(int[]){2, 0, 0});     releaseResources(2, (int[]){0, 2, 0});

   return 0;
```

```
}

void initialize() {
    // Set the available resources
    for (int i = 0; i < MAX_RESOURCES; i++) {
        printf("Enter the number of available instances of resource %d: ", i + 1);
scanf("%d", &available[i]);
    }

    // Set the maximum resources that each process can request
for (int i = 0; i < MAX_PROCESSES; i++) {
        printf("Enter the maximum resources that process %d can request:\n", i);
for (int j = 0; j < MAX_RESOURCES; j++) {
            scanf("%d", &maximum[i][j]);
        }
    }

    // Initialize allocated and need matrices     for
(int i = 0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
            allocated[i][j] = 0;
need[i][j] = maximum[i][j];
        }
    }
}

void requestResources(int process, int request[]) {
printf("\nProcess %d is requesting resources: ", process);
for (int i = 0; i < MAX_RESOURCES; i++) {
printf("%d ", request[i]);
    }
    printf("\n");

    // Check if the request can be granted
    for (int i = 0; i < MAX_RESOURCES; i++) {
        if (request[i] > need[process][i] || request[i] > available[i]) {
printf("Request cannot be granted. Process %d must wait.\n", process);
return;
        }
    }

    // Temporarily allocate resources     for (int i =
0; i < MAX_RESOURCES; i++) {
available[i] -= request[i];
allocated[process][i] += request[i];
        need[process][i] -= request[i];
    }

    // Check if the system is in a safe state after allocation
if (isSafeState()) {
        printf("Request granted. Process %d is now allocated resources.\n", process);     }
else {
        // Rollback the allocation
        for (int i = 0; i < MAX_RESOURCES; i++) {
            available[i] += request[i];
allocated[process][i] -= request[i];
            need[process][i] += request[i];
        }
        printf("Request denied. Process %d must wait to avoid deadlock.\n", process);
    }

    printState();
}

void releaseResources(int process, int release[]) {
printf("\nProcess %d is releasing resources: ", process);
    for (int i = 0; i < MAX_RESOURCES; i++) {
printf("%d ", release[i]);
    }
    printf("\n");

    // Release resources
    for (int i = 0; i < MAX_RESOURCES; i++) {
available[i] += release[i];         allocated[process][i]
-= release[i];
        need[process][i] += release[i];
    }

    printState();
}

bool isSafeState() {
    int work[MAX_RESOURCES];
    bool finish[MAX_PROCESSES];

    // Initialize work and finish arrays     for (int i
= 0; i < MAX_RESOURCES; i++) {
work[i] = available[i];
    }

    for (int i = 0; i < MAX_PROCESSES; i++) {
        finish[i] = false;
    }

    // Find a process to satisfy the resource request
    for (int i = 0; i < MAX_PROCESSES; i++) {
if (!finish[i]) {             int j;
            for (j = 0; j < MAX_RESOURCES; j++) {
if (need[i][j] > work[j]) {
                    break;
                }
            }

            // If all resources are available, allocate them to the process
            if (j == MAX_RESOURCES) {
                finish[i] = true;
                for (int k = 0; k < MAX_RESOURCES; k++) {
                    work[k] += allocated[i][k];
                }
                i = -1;  // Start from the beginning to check for more processes
            }
        }
    }

    // If all processes finish, the system is in a safe state
    for (int i = 0; i < MAX_PROCESSES; i++) {
        if (!finish[i]) {
return false;
        }
    }

    return true;
}

void printState() {     printf("\nCurrent state:\n");
printf("Available resources: ");     for (int i = 0; i
< MAX_RESOURCES; i++) {         printf("%d
", available[i]);
    }
printf("\n");

    printf("Maximum resources:\n");     for (int i = 0;
i < MAX_PROCESSES; i++) {         for (int j = 0; j
< MAX_RESOURCES; j++) {
            printf("%d ", maximum[i][j]);
        }
        printf("\n");
    }

    printf("Allocated resources:\n");     for (int i =
0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
printf("%d ", allocated[i][j]);
        }
        printf("\n");
    }

    printf("Needed resources:\n");     for (int i =
0; i < MAX_PROCESSES; i++) {
        for (int j = 0; j < MAX_RESOURCES; j++) {
printf("%d ", need[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

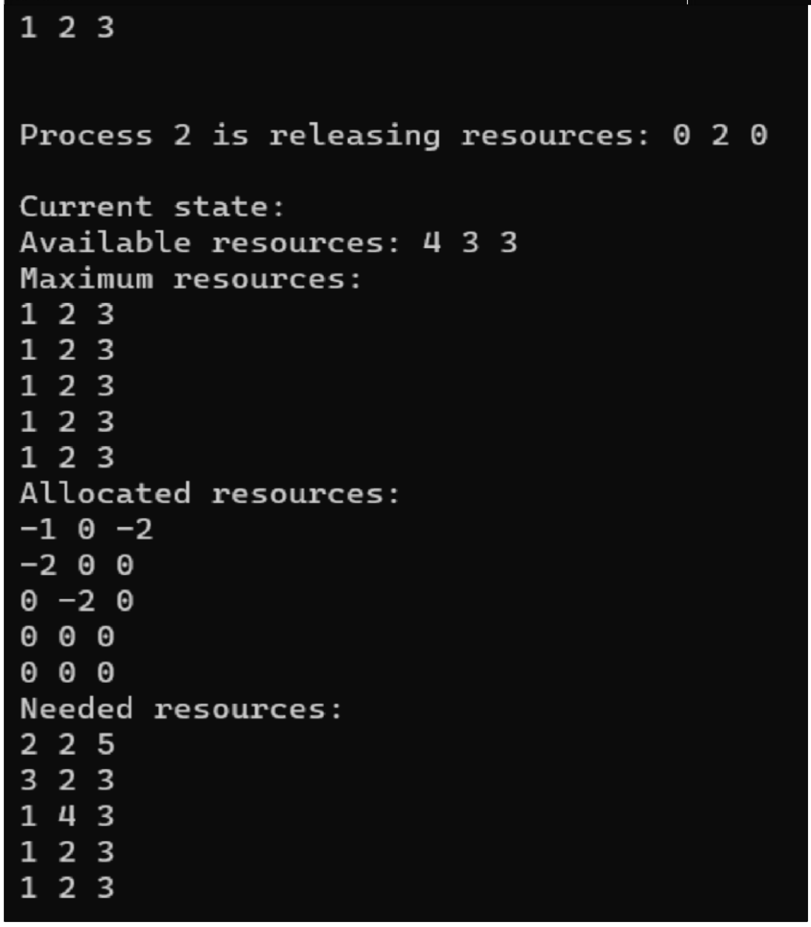
Outputs of the above C program



```
Enter the number of available instances of resource 1: 1
Enter the number of available instances of resource 2: 1
Enter the number of available instances of resource 3: 1
Enter the maximum resources that process 0 can request:
1
2
3
Enter the maximum resources that process 1 can request:
1
2
3
Enter the maximum resources that process 2 can request:
1
2
3
Enter the maximum resources that process 3 can request:
1
2
3
Enter the maximum resources that process 4 can request:
1
2
3

Process 0 is requesting resources: 3 1 2
Request cannot be granted. Process 0 must wait.

Process 1 is requesting resources: 2 1 0
Request cannot be granted. Process 1 must wait.

Process 0 is releasing resources: 1 0 2

Current state:
Available resources: 2 1 3
Maximum resources:
1 2 3
1 2 3
1 2 3
1 2 3
1 2 3
```



```
Allocated resources:
-1 0 -2
0 0 0
0 0 0
0 0 0
0 0 0
Needed resources:
2 2 5
1 2 3
1 2 3
1 2 3
1 2 3


Process 2 is requesting resources: 0 2 0
Request cannot be granted. Process 2 must wait.

Process 1 is releasing resources: 2 0 0

Current state:
Available resources: 4 1 3
Maximum resources:
1 2 3
1 2 3
1 2 3
1 2 3
1 2 3
Allocated resources:
-1 0 -2
-2 0 0
0 0 0
0 0 0
0 0 0
Needed resources:
2 2 5
3 2 3
1 2 3
1 2 3
1 2 3
```



```
1 2 3


Process 2 is releasing resources: 0 2 0

Current state:
Available resources: 4 3 3
Maximum resources:
1 2 3
1 2 3
1 2 3
1 2 3
1 2 3
Allocated resources:
-1 0 -2
-2 0 0
0 -2 0
0 0 0
0 0 0
Needed resources:
2 2 5
3 2 3
1 4 3
1 2 3
1 2 3
```

STEP BY STEP EXPLANATION OF THE CODE:

1. Banker's Algorithm Class:

- The constructor initializes the BankerAlgorithm class with the number of processes, number of resources, available resources, maximum claim matrix, and allocation matrix.
- It calculates the initial need matrix based on the difference between the maximum claim and allocation for each process. **2. Is_safe_state method:**
- **is_safe_state** checks if granting a resource request will result in a safe state.
- It first checks if the request is within the bounds of the need matrix.
- If the request is valid, it checks if the system has enough available resources to fulfill the request.
- If both conditions are met, it simulates the allocation by updating the available, allocation, and need matrices.
- It then checks if the system is still in a safe state using the **check_safety** method.
- If the system is safe, it returns **True**; otherwise, it rolls back the allocation and returns **False**.

3. Check_safety Method:

- **check_safety** simulates the resource allocation to check if the system is in a safe state.
- It uses the Banker's algorithm safety check logic.
- It maintains a temporary sequence of processes and updates the available resources accordingly.
- The process continues until all processes can finish or no further progress is possible.
- If all processes can finish, it returns **True**; otherwise, it returns **False**.

4. Request_resources Method:

- **request_resources** is responsible for processing resource requests for a specific process.
- It calls **is_safe_state** to check if the request can be granted safely.
- If the request is granted, it updates the state, prints a success message, and displays the current state.

- If the request is denied, it prints a message and displays the current state.

5. Print_state Method:

- **print_state** prints the current state of the system, including the available resources, allocation matrix, maximum claim matrix, need matrix, and safe sequence.

6. Main Function:

- The **main** function serves as the entry point for the program.
- It takes user input for the number of processes, the number of resource types, available resources, maximum claim matrix, and allocation matrix.
- It creates an instance of the **BankerAlgorithm** class and enters a loop to interactively process resource requests until the user chooses to exit.

TO DEMONSTRATE HOW BANKER'S ALGO ENSURES SAFE STATE AND PREVENTS DEADLOCKS:

To demonstrate how the Banker's algorithm ensures safe states a prevent deadlocks, Let's consider a scenario with three processes (P0, P1, P2) and three resource types (A, B, C).

We will go through the steps of the Banker's algorithm, showing the initial state, a resource request, and a resource release.

Initial State:

Processes:

```
P0   P1   P2
```

Max Resources:

```
A B C   A B C   A B C
7 5 3   3 2 2   9 0 2
```

Allocated Resources:

```
A B C   A B C   A B C
0 1 0   2 0 0   3 0 2
```

Available Resources:

```
A B C
3 3 2
```

Step 1:

Resource Request (P1 requests 1 unit of A, 0 units of B, 0 units of C):

Processes:

```
P0   P1   P2
```

Max Resources:

A B C A B C A B C
7 5 3 3 2 2 9 0 2

Allocated Resources:

A B C A B C A B C
0 1 0 3 0 0 3 0 2

Need Resources:

A B C A B C A B C
7 4 3 0 2 2 6 0 0

Available Resources:

A B C 2 3
2

The system checks if the resource request can be granted without leading to an unsafe state. In this case, the request is granted, and the system is still in a safe state.

Step 2:

Resource Release (P0 releases 0 units of A, 1 unit of B, 0 units of C):

Processes:

P0 P1 P2

Max Resources:

A B C A B C A B C
7 5 3 3 2 2 9 0 2

Allocated Resources:

A B C A B C A B C
0 0 0 3 0 0 3 0 2

Need Resources:

A B C A B C A B C
7 5 3 0 2 2 6 0 0

Available Resources:

A B C 2 4
2

The system checks if the resource release maintains a safe state. In this case, the release is valid, and the system is still in a safe state.

The Banker's algorithm ensures that resource requests are only granted if they do not lead to an unsafe state, and it checks the system's safety after each resource allocation or release. This prevents deadlocks by avoiding resource allocations that could potentially lead to a situation where no further progress is possible.

Advantages of Banker's Algorithm:

- One of the primary advantages of the Banker's algorithm is its ability to prevent deadlocks by carefully assessing and granting resource requests. This proactive approach ensures that processes can only request resources in a way that guarantees a safe state.

- The algorithm promotes efficient resource utilization by allowing processes to request only the resources they truly need and can safely use. This prevents unnecessary resource blocking and contributes to overall system efficiency.

- Banker's algorithm ensures fairness in resource allocation by considering the needs of each process. It prevents scenarios where a single process could indefinitely monopolize resources, leading to a more equitable distribution.

- It supports dynamic changes in resource requirements by dynamically adjusting the available resources and needs of processes. This adaptability is crucial for systems with varying workloads and resource demands.

- The careful verification of resource requests reduces the risk of deadlock occurrence. By granting requests only if they lead to a safe state, the algorithm significantly lowers the probability of processes being deadlocked.

Limitations of Banker's Algorithm:

- One significant limitation is its assumption of a static environment, with a fixed number of processes and resources. In dynamic systems where these parameters can change over time, the algorithm may be less suitable.

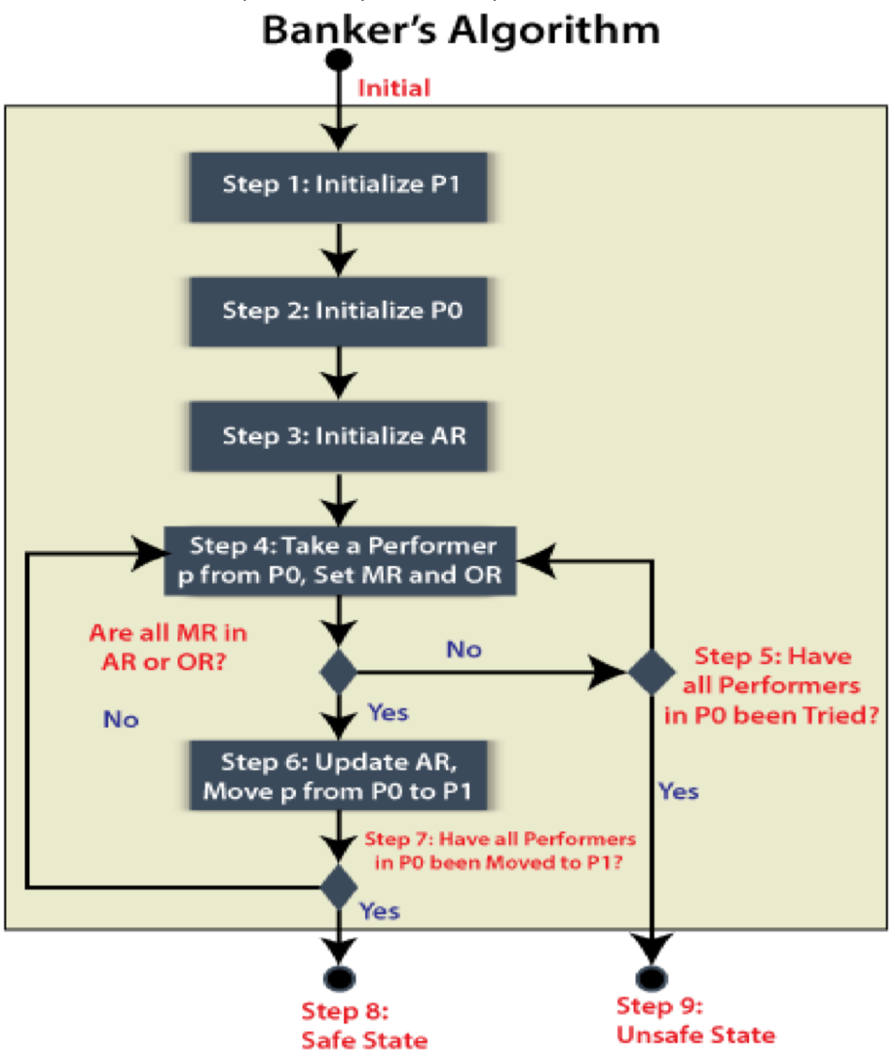
- The algorithm relies on having complete and accurate information about the maximum resource needs of each process in advance. In practice, obtaining such precise information might be challenging or unrealistic.

- Banker's algorithm faces the hold-and-wait problem, where a process holding resources can still request additional resources. This can lead to underutilization of resources and potential inefficiencies.

- The algorithm lacks support for resource preemption, meaning that once a process is allocated resources, it cannot be forced to release them. This limitation might impact the system's ability to adapt to changing priorities.

- Implementing the Banker's algorithm can be complex, especially in large systems. The need for meticulous bookkeeping and constant safety checks adds to the overall system complexity.

- The continuous safety checks required by the algorithm can introduce performance overhead, particularly in systems with a large number of processes. This overhead might impact the system's responsiveness.


Banker's Algorithm
Initial
Step 1: Initialize P1
Step 2: Initialize P0
Step 3: Initialize AR
Step 4: Take a Performer p from P0, Set MR and OR
Are all MR in AR or OR?
No
Yes
No
Step 5: Have all Performers in P0 been Tried?
Yes
Step 6: Update AR, Move p from P0 to P1
Step 7: Have all Performers in P0 been Moved to P1?
Yes
Step 8: Safe State
Step 9: Unsafe State

Group picture